Introduction to Programming Languages

Anthony A. Aaby

HTML Style Guide | To Do | Miscellenous (possible content) | Figures | Definitions

Short Table of Contents

Supplementary Material

Code
Answers

Long Table of Contents

HTML Style Guide | To Do

Preface

Syntax, translation, semantics and pragmatics

Models of Computation

Pragmatics

Supplementary Material

Disclamer and Copyright

This information is provided in good faith but no warranty can be made for its accuracy. Opinions expressed are entirely those of myself and cannot be taken to represent views of past, present or future employers.

If you have questions or comments feel free to send mail to me.
Anthony A. Aaby

Anthony Aaby

Office: 328 Kretchmar Hall
Phone: 1-(509)-527-2067
FAX: 1-(509)-527-2253
email: **aabyan@wwc.edu**

Professor of Computer Science
BA *Mathematics* Loma Linda University
MA *Mathematics* The Pennsylvania State University
Ph.D. *Computer Science* The Pennsylvania State University

Course Syllabi

CPTR 235 System Software and Programming
CPTR 352 Operating Systems
CPTR 415 Introduction to Databases
CPTR 425 Introduction to Networking
CPTR 435 Software Engineering
CPTR nn Software Project Management
CPTR 460 Parallel and Distributed Programming
CPTR 464 Compiler Design
CPTR 496,7,8 Seminar

INFO 150 Software Application
INFO 250 System Software

Instructions for students

Research Interests

Parallel and Distributed Computing
Programming Languages
Natural Language Processing
Theory of Programming
Automated Reasoning
Systems Research Group

Work in Progress

C Family of Languages
Essays on Ethics
Computing Curricula 2001 1991
CS Labs
Hardware
Logic Programming -- tiger
Logic resources
The Logic of Fact, Fiction, Fantasy, & Physics
Programming Languages

- Compiler Construction using Flex and Bison (Lecture Notes)
- Multiparadigm Programming Language
- Textbook Introduction Programming Languages
- Tutorials [Pascal | Prolog | Gödel | Scheme | SML | Haskell | PCN]

Software Engineering

- Introduction to Software Engineering (Lecture Notes)
- Programming Patterns
- Design
- Temporal logic for specification

Colloquia

Index

Stand alone lectures

Prolog and AI
OOP
VR & Limits of Computation

Old Syllabi

INFO 105 Personal Computing
CPTR 141 Introduction to Programming
CPTR 215 Assembly Language Programming
CPTR 221-222 Programming Languages
CPTR 350 Computer Architecture
CPTR 351 Memory and I/O Systems

Local

Computing at WWC
Software Engineering at WWC
BS, BA, AS - CS
BS-SE; BS-SE alt
BSE-SE vs BS-SE

Last Modified - .
Send comments to aabyan@wwc.edu

CPTR235 System Software & Programming - 4

CAUTION: subject to change

Description:

Introduction to Unix, system adminstration, system software including database managment systems and web severs, and system programming. Prerequisites: CPTR 141 or CIS 130.

This course has a significant project componenent that follows a *project-based organization* rather than the traditional lecture-lab organization. All class members are expected to spend 120 hours (12 hours/week) on this course.

Goals:

Upon completion of the course students will have a variety of skills in the following areas:

- Unix operating system
- Standard Unix C libraries
- Development tools
- Documentation
- Data management (database)
- Scripting languages (e.g., shell, Tcl, Perl)
- Concurrency and distributed applications
- GUI programming (e.g., Tk, GTK+, Qt)
- Web programming

Consistent with peer review practice in academia, the source code for software developed for this course must be available to all class members and if desired may be protected by one of the approved open source licenses unless prior arrangement is made for a more restricted copyright protected by an NDA.

Resources

Lecture notes and schedule
Skill set
Projects
Forms References (recommended in bold face)

- Application server
  - Charles Au *Linux Apache Web Server Administration* (Linux Library) Sybex 2000

- Ben Laurie, Peter Laurie, Robert Denn (Editor) *Apache : The Definitive Guide* O'Reilly 1999
- Amos Latteier, Michel Pelletier *The Zope Book* New Riders Publishing 2001.
- Martina Brockmann, Katrin Kirchner, Sebastian Luhnsdorf, Mark Pratt *Zope Web Application Construction Kit* Sams 2001

- Database
  - Stucky, Matthew. *MySQL: Building User Interfaces* New Riders 2001.
- GUI
  - GTK
    - Harlow, Eric. *Developing Linux Applications with GTK+ and GDK* (Feb.1999)
    - Stucky, Matthew. *MySQL: Building User Interfaces* New Riders 2001.
  - TCL/TK
  - Qt
    - Matthias Kalle Dalheimer *Programming With Qt* O'Reilly & Associates 1999.
- Unix (& Linux) Programming
  - **Matthew, et. al *Professional Linux Programming* Wrox Press Ltd. 2000**
  - **Stones and Matthew *Beginning Linux Programming* Wrox Press Ltd. 1999**
  - Chan, Terrence *Unix System Programming Using C++* Prentice-Hall PTR 1997
  - Haviland, Gray & Salama *Unix System Programming* Addison-Wesley 1999
  - Mitchell, Oldham, Samuel & Oldham *Advanced Linux Programming* New Riders 2001
  - Robbins & Robbins *Practical Unix Programming* Prentice-Hall PTR 1996
- Unix (& Linux)
  - Sarwar, Koretsky, & Sarwar. *Linux: The Textbook* Addison-Wesley 2002.
  - **IBM Developer Resource IBM**

Internet
Fortuitous.com's Linux Fundamentals
Linux Documentation Project - Rute Users Tutorial and Exposition
Aaby's Unix notes
WWC CS Department HowTo pages
WWC IS Unix FAQ
Usenet:
Technical Journals:

CACM, Computing Surveys, JACM,

Evaluation

The course grade is determined by the quantity and quality of work completed on laboratory assignments, homework, and tests. The grade expectations document helps to explain the different grades.

GRADING WEIGHTS		LETTER GRADES	
Labs	50%	As	90 - 100%
Homework	-%	Bs	80 - 89%
Tests	50%	Cs	70 - 79%
		Ds	60 - 69%

Study Hints

- **Ask questions in class (you are paying for it).**
- **At the *first* sign of difficulty, talk to your teacher.**
- **Form a study group and meet regularly.**
- **Construct chapter summaries noting concepts, definitions, & procedures.**

Last Modified - . Comments and content invited aabyan@wwc.edu

---

CPTR 235 System Software and Programming - 4

Lecture notes and schedule

These schedules are subject to change.

Lecture notes and schedule

Week	Topic	Text/Resources	Assignments
1	Orientation Introduction	Fortuitous.com's Linux Fundamentals Linux Documentation Project - Rute Users Tutorial and Exposition Aaby's Unix notes WWC CS Department HowTo pages WWC IS Unix FAQ	Become familiar with Unix/Linux
2	More Unix		
3	Development tools	Bison and Flex (yacc and lex)	1. Modify a compiler 2. Create an e-commerce website. • The textbook site • A book exchange • A car pooling site
4	Data management		
5			

6	Concurrency		1. Create a web based user interface which allows the user to obtain the current time from several on and off campus machines. 2. Construct a program which permits multiple users on multiple machines to act as consumers and producers - adding and deleting items from a bounded queue.
7	GUI		Create user interfaces for your website with GTK+
8			
9	Web		
10			
	Project Due & Final Exam		

Last Modified - . Comments and content invited aabyan@wwc.edu

CPTR 235 System Software and Programming

Server-side include
Imbedded PHP

Skills Checklist

Name: **Date:**

	BLP	PLP	Have	Demo
Unix/Linux enviornment				
Introduction to Unix				
C/C++	1			
Shell programming	2			
Files	3			
Unix environment	4			
Terminals	5			
Curses	6			
Security		12		
Multimedia		19		
Diskless systems		22		
Beowulf Clusters		24		
Device drivers	21	26		
Development tools				
Software engineering		1		
Internationalization		28		
make	8			
RCS/CVS	8	2		
debugging - gdb	9	6		
Testing		11		
Flex & Bison		10		
Distribution				
tar, patches	8	27		

packages configure autoconf automake		27		
Documentation				
Man pages, troff, info files	8	25		
Command line help TeX, LaTeX DocBook Literate programming		25		
Data management				
Memory, files, & dbm	7			
PostgreSQL		3, 4		
MySQL		5		
LDAP		7		
Concurrency and distributed applications				
Processes & Signals	10			
POSIX threads	11			
Pipes	12			
Semaphores	13			
Sockets	14			
RPC		18		
CORBA		20, 21		
User interface				
Tcl	15			
X & Tk	16			
Gnome/GTK+	17	8, 9		
KDE/Qt		13, 14		
The Web				
Perl	18			
HTML	19			
CGI programming	20			
PHP		16		
XML		23		
Python		15, 17		

Last Modified - Wed Oct 27 09:42:00 1999. Comments and content invited aabyan@wwc.edu

---

CPTR 235 System Software and Programming

The Labs

The laboratory exercises are open labs meaning that they are unscheduled but are expected to require 30-40 hours of activity. The programming oriented labs should be done in pairs. More ambitious projects will require a team effort which will provide an opportunity for a variety of roles - excellent resume material.

Several projects are available to cater to the goals and/or major of the student. Students are expected to commit to a project early in the quarter. Neither the textbook nor the lectures may cover enough material to complete the project. You are expected to determine what additional materials are needed and obtain them in sufficient time to complete the project.

Students who would like to work on a different problem may propose an alternative project at any point in the quarter. The project must be well-defined, approved by the instructor, and involve roughly the same amount of work as the remaining assignments.

Major(s)	Project	Short Description	Grading
CIS, CS	Database		
CIS, CS, CpE, SwE			
CS, CpE, SwE	Embeded system		

Embedded systems projects:

- Develop a small application for a palm device
  - Palm OS: Palm, Handspring
  - Windows: Pocket PC or
  - Linux: Sharp's Zaurus SL-5000, GMate's Yopy, Tuxia's iPaq
  - Compaq http://www.compaq.com/.
  - Sharp http://www.sharp.co.jp/.
  - Gmate http://www.gmate.co.kr/.
  - Tuxia http://www.tuxia.com
- Develop a small application for Sun's Java Card (a smart card)

Other projects

- Online course evaluation form
-

Past projects

- Book exchange

Last Modified - Wed Oct 27 09:42:00 1999. Comments and content invited aabyan@wwc.edu

---

Software Process Forms

Meeting Documents

- Meeting agenda
- Meeting minutes

IEEE Project Documents (for details see Software Engineering Standards Committee (SESC) *IEEE Standards Software Engineering* Vols 1-4 IEEE 1999. - in WWC library)

Customer and Terminology Standards

IEEE Std. #	Title
610.12-1990	IEEE Standard Glossary of Software Engineering Terminology
1062 1998 Edition	IEEE Recommended Practice for Software Acquisition
1233, 1998 Edition	IEEE Guide for System Requirements Specifications
12207.0-1996	Software Life Cycle Processes
I12207.1-1997	Software Life Cycle Processes--Life cycle date
12207.2-1997	Software Life Cycle Processes--Implementation considerations

Process Standards

IEEE Std. #	Title
828-1998	IEEE Standard for Software Configuration Management Plans
1042-1987	IEEE Guide to Software Configuration Management
1058-1998	IEEE Standard for Software Project Management Plans (SPMP)
1074-1997	IEEE Standard for Developing Life Cycle Processes
I490-1998	IEEE Guide - Adoption of PMI Standard - A Guide to Project Management Body of Knowledge

Product Standards

IEEE Std. #	Title

Resource and Techniques Standards

IEEE Std. #	Title
830-1998	IEEE Recommended Practice for Software Requirements Specifications
1016-1998	Recommended Practice for Software Design Descriptions

Additional process standards

- Problem statement
- Project Agreement
- Requirements Analysis Document (RAD)
- System Design Rationale Document (SDRD)
- Change Proposal Form (CPF)

Personnel Evaluation

- Time card
- Performance Reviews (Self and Peer)
- Short forms
  - Performance review - performed by the instructor.
  - Peer review - performed by a peer.

    Self evaluation

Lifecycle		Management
Project initiation	Problem statement	
Project steady state	Requirements Analysis Document (RAD)	

Last Modified - . Comments and content invited aabyan@wwc.edu

Introduction to Unix

Topic/Lecture Notes

Introduction

Text editing (vi, emacs, textedit)

The X window system

Electronic mail

Commands and Filters

The **csh** Shell

Shell Scripts

File System Managment

Networking and Internet

Unix and C Programming (dbx)

System Programming (system calls)

System Programming (pipes and sockets)

Program maintenance with `make` and `rcs`

Document preparation using LaTeX

Guide to the CS Department Computing Environment

CS account information

Please review the Policies for Responsible Computing at Walla Walla College and the CS Department Computer User Policies and Procedures.

For help, email mundda@cs.wwc.edu

HOWTO pages for CS and Moonbase

The Basics

- HOWTO: Getting Started logging in, dialup, services
- HOWTO: Use UNIX/Linux
- HOWTO: Use email
- HOWTO: Print in UNIX
- HOWTO: Use floppies and CD's
- HOWTO: Setup a personal web page
- HOWTO: Use StarOffice
- HOWTO: Dialin PPP
- HOWTO: WWW, Browsers, HTML,etc
- HOWTO: Use vi. vim and related editors
- HOWTO: Use emacs and related editors

Applications

- HOWTO: Amaya W3C browser/editor
- HOWTO: Cygwin
- HOWTO: Haskell
- HOWTO: Java (man page)
- HOWTO: JLex

- HOWTO: Make
- HOWTO: Use Make
- HOWTO: Microsoft Visual C++
- HOWTO: MPI - Setup, compile, and run programs
- HOWTO: LAM - compile, and run programs
- HOWTO: Prolog
- HOWTO: Use Oracle

Information

- HOWTO: Install Linux on your own computer using an NFS server.
- HOWTO: Samba file & print server for MS Windows
- HOWTO: Dealing with Microsoft Windows NT
- HOWTO: Install Client32 on Win95/NT
- HOWTO: XML
- HOWTO: Our Hardware

Administration

- HOWTO: Install and run stow
- HOWTO: Install SSH 1&2
- HOWTO: Sendmail (CS Specific)
- HOWTO: Cfengine (CS Specific)
- HOWTO: PPP-Server (CS Specific)
- HOWTO: KBackup (CS Specific)
- HOWTO: Install Apache + PHP + SSL + ndsauth
- HOWTO: Install BigBrother
- HOWTO: Install Zope

Help us: Suggest changes, other HOWTOs, create some of your own and share them with us -- use our template.
Additional Resources Usenet comp FAQs

Programming Languages

Pascal
Perl
Tcl/Tk

C Family

C, C++
Java

Logic Programming

Gödel

Functional Programming

Lisp
Scheme
SML

Parallel/Distributed Programming

Fortran M
Java
MPI
PCN
PVM
SR

Assembler

LinuxAssembly.org
Neveln, Bob. Linux Assembly Language Programming P-H PTR 2000
MASM
Hal
x86
SPIM
Orion
MIPS
TASM

Graphics

XLIB
Postscript
Unix/HPLJ4M

Last Modified October 18, 2000

Send comments to webmaster@cs.wwc.edu

CPTR352 Operating Systems - 4

Description

History, evolution, and philosophies; tasking and processes; process coordination and synchronization; scheduling and dispatch; physical and virtual memory organization; device management; file systems and naming; security and protection; communications and networking; distributed operating systems and real-time concerns. Laboratory work required. Prerequisite: CPTR 215 Assembly Language Programming.

Course contents cover subtopics OS1-8 of the topic OS Operating Systems of the The Computer Science Body of Knowledge described in the Computing Curricula 2001.

Please read: Integrity, Computing, & Disability Polices

Goals

Satisfactory completion of this course requires demonstration of the following skills:

- Overview of operating systems (2)
- Operating system principles (2)
- Concurrency
- Scheduling and dispatch (3)
- Memory management (5)
- Device management
- Security and protection
- File systems

These skills may be structured into the following:

- be familiar with how operating systems manage processes
- be familiar with how operating systems manage storage
- be familiar with how operating systems provide protection and security
- be familiar with the basic concepts of distributed systems.

Evaluation

The course grade is determined by the quantity and quality of work completed on laboratory

assignments, homework, and tests. The grade expectations document helps to explain the different grades. Attach a completed work summary sheet with your work.

WEIGHT, %, & GRADES

Labs	40%	90 - 100%	As
Homework	20%	80 - 89%	Bs
Paper/report	10%	70 - 79%	Cs
Tests	30%	60 - 69%	Ds

You need good C, C++, or Java programming skills and can expect to put in 9-12 hours per week for the class (including lectures) and an additional 3-4 hours per week for the lab/project.

Resources

Lecture notes and schedule

Operating System Labs: Several projects are available to cater to the goals and/or major of the student.

Textbook (in bold face):

Bacon, Jean, *Concurrent Systems* Addison-Wesley 1998

Crowley, Charles, *Operating Systems: A Design-oriented Approach* Irwin 1997

Nutt, Gary. *Operating Systems: A Modern Perspective, Lab Update 2e* Addison-Wesley 2002 ISBN 0-201-74196-2

Nutt, Gary. *Kernel Projects for Linux* Addison-Wesley 2000 ISBN 0-201-61243-7

Stallings, William, *Operating Systems: Internals and Design Principles 3rd ed.* Prentice-Hall 1998

Tanenbaum, *Operating Systems: Design and Implementation 2nd ed.* Prentice-Hall 1997

Tanenbaum, *Modern Operating Systems 2nd ed*. Prentice-Hall 2001 ISBN 0-13-031358-0

Silberschatz & Galvin, *Operating System Concepts 5th ed.* John Wiley 1998

Other Books:

Linux Kernel Hacker's Guide

Bar, Moshe. *Linux Internals* McGraw-Hill 2000.

Bovet & Cesati *Understanding the Linux Kernel* O'Reilly & Co. 2000

Mitchell, Oldham, Samuel & Oldham *Advanced Linux Programming* New Riders 2001

Robbins & Robbins *Practical Unix Programming* Prentice-Hall PTR 1996

Stones & Matthew *Beginning Linux Programming* WROX Press

Vahalia, Uresh *Unix Internals* Prentice-Hall 1996

Other References

Buhr et al., 1995. Monitor Classification. *ACM Computing Surveys 27*, 1 (March) 63-108.

Halfhill, T. R., 1996. Unix vs NT. *Byte*, Vol 21 No 5 (May 1996) 42-52.

Usenet:

comp.os.*, comp.sources.unix, comp.unix.*, comp.windows.x

Technical Journals:

CACM, Computing Surveys, JACM, TOCS, SigOPS

Miscellaneous Items

SimOS NachOS OSP

Study Hints

- Ask questions in class (you are paying for it).
- At the *first* sign of difficulty, talk to your teacher.
- Form a study group and meet regularly.
- Construct chapter summaries noting concepts, definitions, & procedures.

95.6.5 a.aaby

CPTR 352 Operating System Design

The Labs

The OS laboratory exercises are open labs meaning that they are unscheduled but are expected to require 30-40 hours of activity. The programming oriented labs should be done in pairs. More ambitious projects will require a team effort which will provide an opportunity for a variety of roles - excellent resume material.

Several projects are available to cater to the goals and/or major of the student. Students are expected to commit to a project early in the quarter. Neither the textbook nor the lectures may cover enough material to complete the project. You are expected to determine what additional materials are needed and obtain them in sufficient time to complete the project.

Students who would like to work on a different problem may propose an alternative project at any point in the quarter. The project must be well-defined, approved by the instructor, and involve roughly the same amount of work as the remaining assignments.

Major(s)	Project	Short Description	Grading
CIS, CS	Operating system administration	Setup and administer a Linux, Solaris, and/or MS-Windows2000 Server • Kaplenk, Joe *Unix System Administrator's Interactive Workbook* Prentice-Hall PTR 1999 • Helmick, Jason *Preparing for MCSE Certification (Windows 2000 Server)* DDC Publishing 2000	Grade sheet

CIS, CS, CpE, SwE	OS programming interface (API)	Gain experience in systems programming using the OS API • Chan, Terrence *Unix System Programming Using C++* Prentice-Hall PTR 1997 • Haviland, Gray & Salama *Unix System Programming* Addison-Wesley 1999 • Nutt, Gary *Operating System Projects for Windows NT* Addison Wesley 2000 • Stones and Matthew *Beginning Linux Programming* Wrox Press Ltd. 1999	
CS, CpE, SwE	OS internals	Gain experience with OS internals Linux source code browser or Linux Kernel Cross Ref • Textbook projects or • OS internals	Grade sheet
CS, CpE, SwE	OS Simulation	Simulators: SimOS, NachOS, OSP, Crowley's sossim, jsos Prolog Code from CC1991 • Tasking and Processes - Design and implementation of a simple context switcher and multiple tasks, using a timer to cause context switch. Done either in a high-level language or on available simulator or machine. Student gains understanding of process context and the idea of context switch. • Process Coordination and Synchronization - Using a simulator or an actual system, explore the consequences of shared access under different timings. Develop mechanisms to synchronize access	

and prove lack of conflict. Observe a deadlock and decide how to prevent it from happening. Students gain an appreciation of problems of synchronization and race conditions.

- Scheduling and Dispatch - Run various job mixes in a simulator or actual system under various kinds of scheduling, and then analyze the results. Students learn how to analyze a scheduling policy, and the effects of different scheduling policies.
- Physical and Virtual Memory Organization - Analysis of access times, delays, I/O operations to manage various job mixes under various algorithms, largely with a simulator. Adjustment of page size, page-ahead, victim policies, penalties, etc. to observe behavior of system Students gain an understanding of memory management schemes.
- Device Management
- File Systems and Naming - Experiments (possibly with a simulator) on the effect of file size and transfer latencies, to gain an impression of how file systems behave. By examining the amount of disk space used and the number of accesses, students learn to retrieve and evaluate performance data that occurs out of various possible organizations of files and directories.

Last Modified - Wed Oct 27 09:42:00 1999. Comments and content invited aabyan@wwc.edu

---

ACADEMIC, COMPUTING, & DISABILITY POLICIES

Academic Integrity

An integral part of the mission of Walla Walla College is to prepare its students to be responsible individuals with Christian values. The College expects all members of its community to have integrity, including a steadfast adherence to honesty. Faculty have a responsibility to foster integrity by example and instruction. Students have a responsibility to learn, respect, and practice integrity.

All acts of dishonesty are unacceptable, including cheating, plagiarism, forgery, misrepresentation, falsification, prohibited collaboration, and prohibited use of files. Departments or schools may have specific criteria for behavior and skills suitable to their disciplines which will be communicated to students, typically in course syllabi.

Computer Science Department Addendum All submitted work should be the student's own work. Where there is a deviation from this ideal, the source of ideas and assistance must be properly credited including fellow students, teachers, and reference material. Academic dishonesty may result in course failure and notice will be given to the Academic Administration of the college.

Responsible Computing

Computer users

- may log in only to their own computer accounts.
- must insure that their work does not interfere with others.
- may not examine, copy, modify, or delete files belonging to others without their consent.
- must not waste computer resources.
- must not use WWC computer facilities to gain unauthorized access to remote networks or systems or violate the use policies of any remote system.
- *Computer Science Department Addendum* Use of the Computer Science Department computing environment is governed by the CS Department Computer User Policies and Procedures

See also the Policies for Responsible Computing at Walla Walla College

Software Engineering Code of Ethics

- Browsable HTML format
- Printable PDF format

Disabilities

If you have a physical and/or learning disability and require accommodations, please contact your instructor or the Disabilities Support Services office (basement of Village Hall; 527-2366). Syllabi are available in alternative print formats upon request. Please ask your instructor.

Final Exam

All students are expected to take the final exam as scheduled. Special administrations are arranged by petition to the Associate Vice President for Academic Administration three weeks prior to the close of the quarter. See the Walla Walla College class schedulue for date and time.

Study Hints

- Ask questions in class (you are paying for it).
- At the *first* sign of difficulty, talk to your teacher.
- Form a study group and meet regularly.
- Construct chapter summaries noting concepts, definitions, & algorithms.
- Keep a course/project/lab journal dating and summarizing all ideas, all design decisions, all code modifications, and all problems encountered and the solutions found. The journal should be complete enough to allow someone else to reproduce your sequence of activities.

Maintained by WWC CS Department
Last Modified
Send comments to webmaster@cs.wwc.edu

Operating systems: Work Summary

Name: **Grade:**

Instructions: Enter data from each area (as a percent) into the table below, multiply by the factor placing the result in the last column. Sum the last column, the result is the total. Include this sheet with your work.

	Score	x factor	=
Labs		.4	
Homework		.2	
Paper/report		.1	
Tests		.3	
		Total	

CPTR 352 Operating System Design - 4

Lecture notes and schedule

These schedules are subject to change. **OS internals lab schedule.**

Lecture notes and schedule

Week	Topic	Nutt	Assignments
	OPERATING SYSTEM PRINCIPLES	1 - 4	
1	OS1. Operating system principles (core -- 2 hours)		p. 50; 4, 5, 6, 7 p. 105; TBA Lab 1: Shell program Lab 2: Kernel timers
	PROCESS MANGEMENT	6 - 10	
2, 3	OS2. Concurrency (core -- 6 hours)		Lab 3: Observing OS behavior Lab 4: Bounded Buffer Problem Lab 5: Refining the Shell
4	OS3. Scheduling and dispatch (core -- 3 hours)		
	Test (Process Management)		
	MEMORY MANAGEMENT	11, 12	
5	OS4. Virtual memory (core -- 3 hours)		
	DEVICE MANAGEMENT	5	
6	OS5. Device Managment (core -- 2 hours)		Lab 6: A floppy disk driver
	FILE SYSTEM MANAGEMENT	13	
7	OS7. File systems and naming (core -- 3 hours)		Lab 7: A simple file manager
	Test (Memory & File System Management)		

8	OS6. Security and protection (core -- 3 hours)	14	
	OS8. Real-time systems		
9, 10	DISTRIBUTED SYSTEMS		
	Network Structures	15	Lab 8: Using TCP/IP
	Distributed-System Structures	17	
	Distributed-File Systems	16	
	Distributed Coordination	17	
	Test (Protection and Security, Distributed Systems)		

Design notes

Last Modified - . Comments and content invited aabyan@wwc.edu

Operating System Paper & Report

Write a report (term paper 3-5 pages) and make a presentation in class on one of the following topics:

- Multimedia operating systems
- Multiple processor systems
- Case study of
  - Unix and Linux
  - Windows 2000/XP
  - Be OS
  - ...
- Embedded & real-time systems
- Distributed systems
  - Operating System Directions for the Next Millennium: http://research.microsoft.com/research/sn/Millennium/mgoals.html
- Networks
- Client-server
- Any course topic in greater depth
- OS design principles (see Tanenbaum and Crowley's texts)

Last Modified - . Comments and content invited aabyan@wwc.edu

CPTR 352 Operating System Design Exams

- Process managment
- Memory and file system management
- Protection and security
- Comprehensive Final Exam

CPTR 415 Introduction to Databases - 4

Description

Introduction to the design, use, and programming of database systems, and database management system implementation. Prerequisite: CPTR 143 Data structures and algorithms, MATH 250 Discrete mathematics.

You can expect to put in 9-12 hours per week for the class (including lectures) and an additional 3-4 hours per week for the lab/project.

Goals

Course contents cover subtopics IM1-IM6 of the topic IM. Information Management of the The Computer Science Body of Knowledge described by the Computing Curricula 2001.

Satisfactory completion of this course requires demonstration of the following skills:

- The relational model
  - Be able to construct a simple relational database.
  - Be able to formulate simple queries using the relational algebra.
- Database Query Languages
  - SQL
    - Be able to use the data definition language (DDL).
    - Be able to use the data manipulation language (DML).
- Database Design (The resulting database design must contain at least 15 relations.)
  - Modeling
    - Be able to construct either an E/R diagram or an UML diagram for a database.
    - Be able to use a tool for drawing a diagram.
  - Mapping
    - Be able to map a conceptual model (diagram) to a relational schema.
  - Functional Dependency
    - Be able to determine all functional dependencies of a relation.
    - Be able to determine all candidate keys.
  - Normalization
    - Be able to achieve the desirable state of 3NF by progressing through the intermediate states of 1NF and 2NF if needed.
    - Be able to normalize to the BCNF.
    - Be aware of the 4NF and the 5NF.
- Database Implementation

- Create a database using DDL.
- Identify different classes of users and create appropriate views of the database.
- Create a web interface to the database for each class of user utilizing a programming language interface (e.g. Perl, PHP, Java, C/C++) to the DBMS.

Evaluation

The course grade is determined by the quantity and quality of work completed on homework assignments, the project, and the tests. The grade expectations document helps to explain the different grades. Attach a completed work summary sheet with your work.

	WEIGHT %	& GRADES	
Project	50%	90 - 100%	As
Homework	10%	80 - 89%	Bs
Paper/report	10%	70 - 79%	Cs
Tests	30%	60 - 69%	Ds

Resources

Lecture notes and schedule
Database Labs: Several different projects are available to cater to the goals and/or major of the student.

Software Tools

E/R or UML Diagramming tool: Alloy, Argo/UML, Dia, Rose, Visio
DBMS: Oracle or PostgreSQL
Web page description: HTML; Netscape Composer
Webserver: Apache
Operating System: Unix (Linux)
Scripting language: PHP or Perl
Programming language: C/C++ or Java

Textbook:

Elmasri & Navathe *Fundamentals of Database Systems 3 ed* Addison Wesley 2000 ISBN 0-8053-1755-4
Ullman & Windom *A First course in Database Systems* Prentice-Hall 1997 (ISBN 0-13-861337-0) Text resources
Yarger, Reese, & King *MySQL & mSQL* O'Reilly 1999 (ISBN 1-56592-434-7)
Laurie & Laurie *Apache The Definitive Guide* 2nd ed. O'Reilly 1999 (ISBN 1-56592-528-9)
Gundavaram, Shishir *CGI Programming on the World Wide Web* O'Reilly 1996 (ISBN 1-56592-168-2)

WWW:
Usenet News Groups:
Technical Journals:
CORBA

Comments and content invited aabyan@wwc.edu

Grade Expectations

The "A" Students -- Outstanding Students

Attendance
: Virtually perfect attendance. Their commitment to the class resembles that of the teacher.

Preparation
: Always prepared for class. They always read the assignment. Their attention to detail is such that they occasionally catch the teacher in a mistake.

Curiosity
: Show an interest in the class and in the subject. They look up or dig out what they don't understand. They often ask interesting questions or make thoughtful comments.

Retention
: Have retentive minds. They are able to connect past learning with the present. They bring a background with them to the class.

Attitude
: Have a winning attitude. They have both the determination and the self-discipline necessary for success. They show initiative. They do things they have not been told to do.

Talent
: Have something special. It may be exceptional intelligence and insight. It may be unusual creativity, organizational skills, commitment -- or a combination thereof. These gifts are evident to the teacher and usually to the other students as well.

Results
: Make high grades on tests -- usually the highest in the class. Their work is a pleasure to grade.

The "C" Students -- Average or Typical Students

Attendance
: Miss class frequently. They put other priorities ahead of academic work. In some cases, their health or constant fatigue renders them physically unable to keep up with the demands of high-level performance.

Preparation
: Prepare their assignments consistently but in a perfunctory manner. Their work may be sloppy or careless. At times, it is incomplete or late.

Attitude
: Not visibly committed to the class. They participate without enthusiasm. Their body language often expresses boredom.

Talent
: They vary enormously in talent. Some have exceptional ability but show undeniable signs of poor self-management or bad attitudes. Others are diligent but simply average in academic

ability.

Results

Obtain mediocre or inconsistent results on tests. They have some concept of what is going on but clearly have not mastered the material.

From *Clarifying Grade Expectations* by John H. Williams in **The Teaching Professor**

CPTR 415 Introduction to Databases

The Labs

The database laboratory exercises are open labs meaning that they are unscheduled but are expected to require 80-90 hours of activity. The programming oriented labs should be done in pairs. More ambitious projects will require a team effort which will provide an opportunity for a variety of roles - excellent resume material.

Several projects are available to cater to the goals and/or major of the student. Students are expected to commit to a project and a specific RDBMS early in the quarter. Neither the textbook nor the lectures may cover enough material to complete the project. You are expected to determine what additional materials are needed and obtain them in sufficient time to complete the project. You may need to obtain

1. DBMS specific guides,
2. GUI interface programming guides, and
3. other materials.

Students who would like to work on a different problem may propose an alternative project at any point in the quarter. The project must be well-defined, approved by the instructor, and involve roughly the same amount of work as the remaining assignments.

Major(s)	**Project**	**Short Description**
CIS, CS	Database adminstration	Setup and administer a DBMS such as Oracle or MS-SQLServer
CIS, CS, CpE, SE	DB design & implementation Grade form RDBMS: Oracle or PostgreSQL Language: C or Java	Design and implement a database using a RDB. Suggested database projects include: • Airport • Art gallery/museum • Congressional voting • CS Department Students and Alumni DB • E-commerce web site (a group project?) • Embedded database • Human resources • Inventory: auto parts, department store, super market • Library

		• Medical practice • Multimedia database • Music Store • Pharmacy • Reservation system - airline, hotel, etc • Sports teams • University (WWCs?) Your solution must include • A problem statement • Requirements analysis • A conceptual design (using either UML or ER diagrams prepared using a tool such as Visio or Dia) • A logical design (for a RDBMS or an ODBMS) • A refined schema (elimination of redundencies) • A physical implementation (using ORACLE or PosgreSQL) with installation scripts. • A user interface using VB, Java, browser, etc. See the Music Store project description for a sample project description. Submit your entire project in a tar file with installation instructions and scripts.
CS, CpE, SE	DBMS implementation	Implement and/or modify a DBMS such as MySQL, PostgreSQL, InterBase or other opensource RDBMS or OODBMS.

Last Modified - Wed Oct 27 09:42:00 1999. Comments and content invited aabyan@wwc.edu

CPTR 415 Introduction to Databases -- 4

IM. Information Management (10 core hours)

Information Management (IM) plays a critical role in almost all areas where computers are used. This area includes the capture, digitization, representation, organization, transformation, and presentation of information; algorithms for efficient and effective access and updating of stored information, data modeling and abstraction, and physical file storage techniques. It also encompasses information security, privacy, integrity, and protection in a shared environment. The student needs to be able to develop conceptual and physical data models, determine what IM methods and techniques are appropriate for a given problem, and be able to select and implement an appropriate IM solution that reflects all suitable constraints, including scalability and usability.

Week	Topic	Text	Assignments
1	IM1. Information models and systems (core 3 hours)	1	Lab: requirements
2	IM2. Database systems (core -- 3 hours)	2	Software tools
Database design & programming			
3	Database LifeCycle IM3. Data modeling (core -- 4 hours) Classroom activity: Conceptual design Classroom activity: Logical design	3, 4, 16.1, 16.2	Lab: conceptual design Modeling tools Lab: logical design
4-6	IM4. Relational Databases (8 hours) **Mid-term Test -** Conceptual/logical design	7	
7-8	IM5. Database query languages (5 hours)	8	
8-10	IM6. Relational database design (5 hours) Programming with SQL	9, 10, 14, 15	Lab: refined schema Lab: physical implementation Lab: User interface implementation
11	**Final Test**		
Advanced topics			
	IM7. Transaction processing Transactions Failure and Recovery Concurrency Control IM8. Distributed databases IM9. Physical database design		

IM10. Data mining
IM11. Information storage and retrieval
IM12. Hypertext and hypermedia
IM13. Multimedia information and systems
IM14. Digital libraries

Last Modified - . Comments and content invited aabyan@wwc.edu

Database Paper & Report

Write a report (term paper ~5 pages) and make a presentation in class on one of the following topics:

- Record storage, file organization and index structures.
- Object-oriented database technology
- Transaction processing
- Concurrency control
- Deductive databases
- Data warehousing and data mining
- Emerging database technologies
    - Active database concepts
    - Temporal database concepts
    - Spatial and multimedia databases
    - Distributed databases
    - Geographic information systems

Last Modified - . Comments and content invited aabyan@wwc.edu

CPTR 415 Introduction to Database

Sample exams; Topics

- Exam1
- Exam2
- Comprehensive Final Exam

CPTR425 Introduction to Networking - 4

Description

A study of networking algorithms, network architecture, network elements, data link, switching and routing, end-to-end protocols, data security, and programming. Prerequisite: CPTR 352 Operating System Design.

You can expect to put in 9-12 hours per week for the class (including lectures) and an additional 3-4 hours per week for the labs.

Course contents cover subtopics AR 9, NC1-4 of the topic NC Net-Centric Computing of the The Computer Science Body of Knowledge described in the Computing Curricula 2001.

Please read: Integrity, Computing, & Disability Polices

Goals

Upon completion of the course you will be familiar with

- the OSI network architecture,
- TCP/IP network architecture,
- the basic algorithms for computer networks, and
- how to program network applications.

AR9 Architecture for networks and distributed systems

NC. Net-Centric Computing (15 core hours)

NC1. Introduction to net-centric computing (2)
NC2. Communication and networking (7)
NC3. Network security (3)
NC4. The web as an example of client-server computing (3)
NC5. Building web applications
NC6. Network management
NC7. Compression and decompression
NC8. Multimedia data technologies
NC9. Wireless and mobile computing

Evaluation

The course grade is determined by the quantity and quality of work completed on laboratory assignments, homework, and tests. The grade expectations document helps to explain the different grades.

	WEIGHT % & GRADES		
Labs	40%	90 - 100%	As
Homework	20%	80 - 89%	Bs
Paper/report	10%	70 - 79%	Cs
Tests	30%	60 - 69%	Ds

You need good C, C++, or Java programming skills and can expect to put in 9-12 hours per week for the class (including lectures) and an additional 3-4 hours per week for the lab/project.

Resources

Lecture notes and schedule
Networking Labs
Books (Textbook in bold face):

Comer, Douglas E. *Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture*, 4/e PrenticeHall 2000. ISBN 0-13-018380-6
Peterson & Davie *Computer Networks 2nd ed.* Morgan Kaufman 2000
Mann, Scott *Linux TCP/IP Network Administration* Prentice Hall PTR 2000
Kurose & Ross *Computer Networking* Addison-Wesley 2001
Stallings, William *Data & Computer Communications 6th ed.* Prentice-Hall 2000
Tanenbaum, Andrew *Computer Networks* 3rd ed. Prentice-Hall 1996
Stevens, W. Richard *TCP/IP Illustrated, Volume 1 The Protocols* Addison-Wesley 1994
Wright & Stevens *TCP/IP Illustrated, Volume 2 The Implementation* Addison-Wesley 1995
Donahoo & Calvert *The Pocket Guide to TCP/IP Sockets C version* Morgan Kaufman 2001
Information theory

Other References

- Iren, Amer, & Conrad. (1999) The Transport Layer: Tutorial and Survey in ACM Computing Surveys Vol 31 # 4 Dec 1999 pp. 360-405.

Usenet:
Internet Engineering Taskforce (for RFCs)
Technical Journals:

CACM,

Computing Surveys,
JACM,
TOCS,
SigOPS,
ACM SIGCOMM

Study Hints

- Ask questions in class (you are paying for it).
- At the *first* sign of difficulty, talk to your teacher.
- Form a study group and meet regularly.
- Construct chapter summaries noting concepts, definitions, & procedures.

Last Modified - . Comments and content invited aabyan@wwc.edu

CPTR 425 Introduction Networking - 4

The Networking Laboratories

The networking laboratory exercises are open labs meaning that they are unscheduled but are expected to require 30-40 hours of activity. The programming oriented labs should be done in pairs. More ambitious projects will require a team effort which will provide an opportunity for a variety of roles - excellent resume material.

Several projects are available to cater to the goals and/or major of the student. Students are expected to commit to a project early in the quarter. Neither the textbook nor the lectures may cover enough material to complete the project. You are expected to determine what additional materials are needed and obtain them in sufficient time to complete the project.

Students who would like to work on a different problem may propose an alternative project at any point in the quarter. The project must be well-defined, approved by the instructor, and involve roughly the same amount of work as the remaining assignments.

Major(s)	**Project**	**Short Description**	**Grading**
CS, CpE, SwE	*SPECIAL PROJECT SPRING 2001*	Work with the compiler design class on the implmentation of Tel, a language for writing networking protocols.	
CIS, Technology	Network design & implementation	Gain experience in networking technologies	
CIS, CS	Unix Network administration Microsoft Network administration	Gain experience in network administration	
CS, CpE, SwE	Networking APIs • Sockets • TCP/IP API • Java Networking API • Perl Networking API	Gain exerience in programming with various networking APIs.	

CS, CpE, SwE	Networking internals Simulation tools • *cnet* network simulator • The Network Simulator - ns-2	Design and implementation of networking protocals.	

Last Modified

Send comments to aabyan@wwc.edu

CPTR 425 Introduction Networking - 4

Schedule and lecture notes

Week	Lecture Notes	Reading	Assignment
1	Preliminaries Foundation	Chapter 1	p. 60 1. 5, 6 2. 11-13 3. 15-21 4. 28-30 Sockets programming
2	Direct Link Networks (physical layer) Frohne - Information theory Thompson - error detection & correction	Chapter 2 except 2.2, 2.9	p. 156 • 1 of #s 1-3 • 1 of #s 4-8 • 2 of #s 9-32 • 2 of #s 33-45 Alternate assignment: provide complete documentation for one of the Linux ethernet drivers
3	Packet Switching (data link layer)	Chapter 3 except 3.4	switching p. 235 # 1, 2, 3, 4, 12, 13
4	Internetworking IP (network layer) Anderson - Internet telephony	Chapter 4 except 4.4	router p. 354 # 12, 13,
5	End-to-end protocols (transport layer)	Chapter 5	UDP TCP RPC p. 433 # 49
6	Congestion control and resource allocation	Chapter 6 optional	
7	End-to-end data (presentation layer) **7.1** Presentation formatting	Chapter 7	

8	Network security **8.4 Firewalls**	Chapter 8	p. 619 #s 22-26 alternate assignment OpenSSH attacks monitoring firewall
9	Applications (application layer)	Chapter 9	DNS SMTP MIME HTTP SNMP RTP

Last Modified

Send comments to aabyan@wwc.edu

CPTR435 Software Engineering - 4

Invitation to submit projects

Bulletin Description: Study of the issues involved in building large software systems. Topics include the methods, languages, and tools used in contemporary software development, including software process models, project management, software metrics, software analysis and design, verification and validation, object-oriented concepts, professionalism and ethics. Prerequisites: CPTR 143, 221.

This course follows a *project-based organization* rather than the traditional lecture-lab organization. It is immersion style in that students learn software engineering concepts by participating in all phases of a software engineering project and are assigned a variety of roles. All class members are expected to spend 120 hours (12 hours/week) on this course.

Goals:

- The primary goal of the course is to experience the types of written communication that are found in large software engineering projects.
- The secondary goal is that upon completion of the course students will
  - have been exposed to important issues in software engineering by
    - solving real problems
    - described by real clients
    - with real tools
    - under real constraints;
  - have clocked 120 hours (12 hours/week) on the project.

Consistent with peer review practice in academia, the source code for software developed for this course must be available to all class members and if desired may be protected by one of the approved open source licenses unless prior arrangement is made for a more restricted copyright protected by an NDA.

Resources

Lecture notes and schedule
Projects

Books: (Textbook in bold face):

Bruegge & Dutoit, *Object-Oriented Software Engineering* Prentice-Hall 2000
Beck, Kent *Extreme Programming Explained: Embrace Change* Addison Wesley Longman 2000

Beck & Fowler *Planning Extreme Programming* Addison-Wesley 2000
Moore, James W. *Software Engineering Standards: A User's Road Map* IEEE Computer Society Press 1997
Software Engineering Standards Committee (SESC) *IEEE Standards Software Engineering Vols 1-4* IEEE 1999
Jefferies, Anderson, & Hendrickson *Extreme Programming Installed* Addison-Wesley 2000
Hunt, Thomas, Cunningham *The Pragmatic Programmer: From Journeyman to Master* Addison-Wesley 1999
Page-Jones, Meilir *Fundamentals of object-oriented design in UML* Addison-Wesley Longman Pub Co 2000
Sandred, Jan. (2001) *Managing Open Source Projects: A Wiley Tech Brief* Wiley 2001.
Sommerville, Ian *Software Engineering 6e* Addison-Wesley 2000
Schach, Stephen, R. *Classical and Object-Oriented Software Engineering* McGraw-Hill 1999
Humphrey, Watts *Introduction to the Personal Software Process* Addison-Wesley 1997.) PSP Scripts and forms are available.
Humphrey, Watts *Introduction to the Team Software Process* Addison-Wesley 2000.) see Supplements. TSP Scripts and forms are available.

Forms

Internet

- UML Zone
- Rational
- SWEBOK
- Extreme Programming and The Agile Alliance
- Aspect Oriented Software Development

Articles

- Louridas & Loucopoulos (2000) A Generic Model for Reflective Design ACM Transactions on Software Engineering and Methodology Vol 9 # 2 April 2000 pp. 199-237.

Usenet:

comp.se

Technical Journals:

CACM, Computing Surveys, JACM,

Evaluation

Each student is expected to construct a portfolio containing

- his/her resume
- copies of all weekly time cards and activity logs
- copies of all performance reviews
- copies of a software and documentation
- a self review

The course grade is determined by the quantity and quality of work completed on assigned deliverables, complete time card (with date, time, and activity), and performance reviews.

GRADING WEIGHTS		LETTER GRADES	
Time cards	10%	As	90 - 100%
Performance Reviews	40%	Bs	80 - 89%
Deliverables	50%	Cs	70 - 79%
		Ds	60 - 69%
		F	Fired

Study Hints

- Ask questions in class (you are paying for it).
- At the *first* sign of difficulty, talk to your teacher.
- Form a study group and meet regularly.
- Construct chapter summaries noting concepts, definitions, & procedures.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Software Engineering - Student Projects

Students undertake team projects at various points in the course of their studies. Clients for these projects come from a variety of external and internal sources. The team must identify the requirements and write the specification for the software, design and code the solution for the system, test and install it, and provide all relevant documentation.

You can find out more information about having *your* project taken on by our students. You must complete a project proposal form.

CPTR435 Software Engineering - 4

Lecture notes and schedule

This schedule is subject to change.

Week	Milestones	Assignments	Training
	Course initiation		
1	Course Orientation	Initial Assignment Textbook Project Management Requirements Engineering Realtime SE	SE Overview Software Life Cycle Knowledge Areas Project Management
	Course steady state		
2	Requirements Elicitation Project Communication CASE Tools	The Project Alternative Projects	
3	Collaboration Env System Design Modeling & UML		Software Configuration Management
4			
5			
6			
7			Object Design
8	Report:		Rationale Management
9	Report:		Testing
10	Report:		
	Course termination		
11	Collection of student portfolios Course evaluation discussion		

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Group Projects

Projects are designed to give students some experience of group work. Ideally groups should have a mix of personalities, abilities, and sexes. Students may be assigned to groups rather than being left to choose their own groups. Groups will have to find time for group meetings and collaborative work.

One goal of group projects is to illustrate the problems of group working and, sometimes, personality clashes arise. Depending on the individuals involved, the instructor may step in and resolve outstanding issues or may simply leave the group to sort out its own problems.

An aspect of software engineering which students find unsettling is the fact that system users and procurers usually have a vague and potentially contradictory set of requirements. The approach adopted by this instructor is to act as a user and deliberately be vague, contradictory and present impossible requirements.

Individual assessment is more difficult for group projects. Students should realize that education is more important than assessment. Individuals within groups will be assessed independently of the group project and assessment may include input from group members. The final individual grade will be based both on individual assessment and the project grade.

Project deliverables

Design projects

Design projects are very general projects which are not implementable in the time and with the resources available. They may be large applications, require access to special hardware, or may require detailed domain knowledge to complete the implementation. It is intended that students construct high-level specifications and designs of such systems. The aim of the work is to illustrate the problems of writing specifications and designs. It may be possible to prototype parts of the system.

The documents which might be produced are:

- A requirements definition and (partial) specification.
- An outline architectural design.
- A project plan and schedule.
- A prototype of part of the system user interface.

Term projects

Term projects are smaller scale projects that a student or group might take through from initial specification to implementation.

The documents which might be produced are:

- A requirements specification which expands the outline below in more detail.
- A formal specification for part of the project.
- An outline architectural design.
- A detailed design specification.
- A test specification.
- A user manual and associated help frames.
- A project plan and schedule setting out milestones, resource usage and estimated costs.
- A quality plan setting out quality assurance procedures
- An implementation.
- A source tree to support iterative development.
- A baseline release package with configuration files, data files, an installation program and complete documentation.

So that students in later years can understand the standard of work that is expected of them, each group must produce

1. a poster presentation describing their work,
2. a web site which describes and contains their work and, time permitting,
3. a downloadable package of the product and an installation guide.

The department will display the posters and host the web site.

Design Projects

Term Projects

- E-commerce web site.
- Web based voting system for the WWC governance system.(Winter 2001)
- Collaboration Portal/ASP (Winter 2000)
    - Distance learning
    - SE project
- Development Environment for WWC
- WWC student records database
- Test item banking/generation/scoring/analysis with support for multiple teachers and classes. Suggestion, modify the web based voting system for the WWC governance system. Should include:
    - support for graphics and mathML

    - online testing and scoring
- Browser based email tool
- Pattern database, catalogue & browser
- web based network monitor - performance, conectivity, etc.
- Report on an article in IEEE Transactions on Software Engineering.

Project suggestions

- Compiler design
- Database
- Operating systems
- Application for a mobile computing platform
- Networking
- Software engineering
- System software and programming
- Senior seminar
- See also Sommerville's Instructor's Guide

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Personal Software Process (PSP)

Personal Software Processes (for details see Humphrey, Watts *Introduction to the Personal Software Process* Addison-Wesley 1997.)

Forms	Instructions & Scripts
Time Management	
Time Recording Log	
Weekly Activity Summary	
Job Number Log	Job Number Log Instructions
Software Quality Management	
	PSP Process Script
PSP Project Plan Summary	PSP Project Plan Instructions
	Code Review Script
C++ Code Review Guidelines and Checklist	C++ Coding Standard
	Defect Type Standard
Defect Recording Log	Defect Recording Log Instructions

Last Modified - . Comments and content invited aabyan@wwc.edu

Software Development Script

Purpose		**To guide a team though developing a software product.**
Entry Criteria		• A needs statement • Materials, facilities, and resources for team support. • A development team
General		
Step	**Acitivities**	**Description**
0	Launch	• Assign teams and roles.
	Strategy	• Produce the conceptual design, establish the development strategy, make size estimates, and assess risk.
1	Planning	• Produce the team and engineer plans.
2	Requirements	• Define and inspect the requirements. • Produce the system test plan and support materials.
3	Design	• Produce and inspect the high-level design. • Produce the integration test plan and support materials
4	Implement	• Produce the unit test plan and support materials. • Implement and inspect the code.
5	Test	• Build, integrate, and system test. • Produce user documentation
6	Postmortem	• Conduct a postmortem and write final report. • Produce role and team evaluations
Exit criteria		• Completed product or product element and user documentation • Completed and updated project note book. • Documented team evaluations and cycle reports.

CAUTION: an internal study document, is not intended to represent future plans.

Software engineering tracks & a new course in project management

Document status: internal planning

Last Modified - . Comments and content invited aabyan@wwc.edu

This is a study of the possibility of an introductory course in software engineering to meet the needs of CIS, SE, & CpE majors. It is based on selected chapters Sommerville, Ian. Software Engineering 6th ed. (text currently in use for CPTR 435 and one of the leading software engineering texts) and the IS '97 curricular guidelines. The course would include projects tailored to the needs of the different majors. Note: the Sommerville text is designed to support a variety of courses including a year long sequence.

	SE Tracks			SE Project Management
	CIS	SE	CpE	
Overview				
Introduction	a	x	x	x
Computer-based system engineering				x
Software processes	a	x	x	
Project management	a			x
Requirements				
Software requirements	a	x	x	
Requirements engineering processes	a, d			6.4
System models				
Software prototyping				
Formal specification		x		
Design				
Architectural design	a	x	x	
Distributed systems architectures		x		

Object-oriented system design	a	x		
Real-time software design			x	
Design with reuse		x		
User interface design		x		
Critical systems				
Dependability			x	
Critical systems specification			x	
Critical systems development			x	
Verification and validation				
Verification and validation		x		x
Software testing		x	x	
Critical systems validation			x	
Management				
Managing people	d			x
Software cost estimation	d			x
Quality management	d			x
Process improvement	d			x
Evolution				
Legacy systems				x
Software change				x
Software re-engineering				x
Configuration management	d	x		x
Project Implementation		x	x	

The CIS track

- IS '97 calls for a four course sequence
    1. IS '97.7 Analysis and Logical Design (prerequisite IS '97.3 Information Systems Theory and Practice)
    2. IS '97.8 Physical Design and Implementation with DBMS (prerequisite IS '97.7)
    3. IS '97.9 Physical Design and Implementation with Programming Environments (prerequisite IS '97.8 and IS '97.5 Programming, Data and Object Structures)
    4. IS '97.10 Project Management and Practice (prerequiste IS '97.7, corequisites IS '97.8, IS '97.9)
- a - designates topics for IS '97.7
- d - designates topics for IS '97.10

Observations

- Sommerville text, with supplementation with a DBMS text, could cover the IS '97.7-10 sequence.
- CPTR 235 System Software and Programming covers much of the same *technical material* as IS '97.7, IS '97.8 and IS '97.9 but includes much additional material *assumed* to be of little interest to CIS majors. After six courses, (end of sophomore year) CS majors have a broad range of entry level programming skills.
- CPTR 415 Introduction to Databases covers the DBMS material of IS '97.7, IS '97.8 and IS '97.9 at a greater depth and would be an appropriate course for CIS majors wanting additional theoretical background or for MBA students with an undergraduate CIS concentration.
- CPTR 435 Software Engineering covers much the same management and human factors material as IS '97.7, IS '97.8 and IS '97.9 however, it lacks the business and DBMS focus of the IS sequence. For CE and CpE majors, it provides a systematic description of the discipline of software engineering.
- IS '97.10 Project Management, is not available at WWC.

Conclusions

- The proposed BS-SE major could include the IS sequence as an option. However, a common software engineering course does not seem practical for CIS, SE, & CpE majors for the following reasons.
    - The IS sequence is best suited to meeting the needs of CIS majors.
    - The IS sequence has too specific a focus for SE and CpE majors.
    - Conversely, the CPTR courses may be too general for the typical CIS major.
- A project management course should be made available to CIS and SE majors. However, a common course for CIS and SE majors does not seem possible given that the IS sequence is tightly integrated. Scheduling a common course may be difficult. A project management course for SE majors that would not require additional instructional resources could be developed as follows:
    - It would be offered at the same time as CPTR 435 with the same instructor.
    - Its students would be required to manage the CPTR 435 project.
    - It should have limited enrollment (projects should have fewer managers than developers).
    - It should require a term paper.
    - Concurrent enrollment with CPTR 435 should be permitted only if it is possible to prevent conflict between the roles of developer and project manager i.e. there should be at least two projects to permit separation of roles.

DRAFT - Last update:

BS - Software Engineering

A curriculum proposal based on the
IEEE-CS/ACM Education Task Force
Accreditation Guidelines

STATUS
- Added an internship requirement 5/5/2000
- Circulated for comment to EE, CS, BUS, Tech 4/21/2000
- Dropped internship requirement
- Elaborated math and science requirements 11/30/2000
- Reviewed math-science requirements 1/23/2001
- Approved by CS faculty -
- Approved by EE faculty -

Mission Statement

The mission of the software engineering program is to produce graduates which know, understand, and can use the theories, methods and tools which are needed to develop high quality, large and complex software in a cost effective way on a predictable schedule and are prepared to participate in the development of a broad range of software products.

Proposed curriculum

Senior students are required to take the MFAT exam in Computer Science.

SE major - BS degree 192 hours		CrHr
Computer Science & Engineering - 37 hours		
CPTR 141	Introduction to Programming	4
CPTR 142, 143	Data Structures and Algorithms	4,4
CPTR 215	Assembly Language Programming	3
CPTR 316	Programming Paradigms	4
CPTR 352	Design and Analysis of Algorithms	4
CPTR 425	Introduction to Networking	4
CPTR 454	Operating System Design	4
ENGR 121-123	Introduction to Engineering	6
Software engineering - 34 hours		

CPTR 235	System Software & Programming	4
CPTR 245	Object-Oriented System Design	4
CPTR 415	Introduction to Databases	4
CPTR 435	Software Engineering	4
	Software engineering electives	10
ENGR 326	Engineering Economy	3
ENGR 345	Contracts and Specifications	2
ENGR 396	Seminar	0
ENGR 496-498	Seminar	3
ENGR 495	Colloquium	0

Applications and Advanced materials - 36 hours

	Math & science electives	8
	Zero or more hours	0-12
	CPTR, ENGR, INFO electives	
	One or more area (of 12+ hours each)	12-24
	For example:	
	Computer science (beyond requirement)	
	Engineering (beyond requirement)	
	Mathematics (beyond requirement)	
	Science (beyond requirement)	
	COMM 275 Communication Theory 2	
	PSYC 425 Cognitive Psychology 4	

Supporting Areas - 39 hours

ENGL 121-2	College Writing	6
ENGL 323	Writing for Engineers	3
SPCH 101	Fund. of Speech Communications	4
SPCH 207	Small Group Communications	3
MATH 206	Applied Statistics	4
MATH 250	Discrete Mathematics	4
MATH 181	Analytic Geom & Calc I, II	8
MATH 289	Linear Algebra and Applications	3
PHIL 206	Intro to Logic	4

General studies - 50 hours

PSYC 130	H&PE electives	2
	History electives	8
	General Psychology	4
PHYS	Humanities electives	8
	Religion electives	16
	General or Prin of Physics	12
		192

Courses may not be used to satisfy multiple requirements.

Electives and application areas

Students in consultation with their advisors will select 10 hours of software engineering electives, 8 hours of math and science electives and 12-24 hours of application area electives. The details remain to be worked out but some obvious choices include numerical applications in science and engineering, embedded systems (especially the exploding market for wireless devices), and computing infrastructure (compilers, computer networks, operating systems and other system software).

All of the electives and application area courses will be selected from current courses, internships (co-op credit) and team projects.

Math-science requirement

ABET requires one year of mathematics and science i.e., 48 quarter hours. The proposed implementation is as follows:

Area	Classes	Rationale
Mathematics	Discrete, Applied Statistics	ABET
(23 hours)	Calculus I, II, Linear Algebra, Logic	Curricular support
Science	12 hours of General or Principles of Physics	Traditional bias
(16 hours)	4 hours of General Psychology	To support HCI
Electives (8 hours)	Science electives: Astronomy, Biology, Chemistry, Physics, Psychology	
	Math electives: any college level mathematics course	ABET

Differences with current programs

1. BS-SE, BS-CIS differences
    - BS-SE requires a maximum of 24 hours of non CIS business courses and permits upto another 12 hours for a total of 36 hours while BS-CIS requires 59 hours. Note that event the BA-BA requires 59 hours of non CIS business courses.
    - BS-SE requires at least 64 hours of computing course work while BS-CIS requires 48 hours.
    - BS-SE degree requires 48 hours of math and science while the BS-CIS requires 20 hours.
    - Informal data collected over nine years suggests that students migrate from the BS-CS (software option) toward CIS and not vice versa.
2. BS-SE, BS-CS differences
    - BS-SE has no free electives. BS-CS has 33 hours of free electives and there is more elective choice in general studies.
3. BS-SE, BSE-CpE differences
    - BS-SE requires 55 hours of general studies hours of general studies while the BSE-CpE

requires 44.

- BSE-CpE requires 29 hours of engineering courses not required in the BS-SE.
- BS-SE requires 48 hours of math and science while the BSE-CpE requires 55.

Send comments to aabyan@wwc.edu

CPTR 460 Parallel and Distributed Computation -- 4

Check back frequently, this page is under construction.

Syllabus

Topics

Week	Topic/Lecture Notes	Reading	Assignment	Due
	PART I			
1	Introduction Hardware Performance Andrews	PP 1 PL: CP	p. 36 1.1-1.5, 1.7, 1.12, 1.13	1/14
2	Message Passing Introduction to MPI Debugging Your Program	PP 2 PL: CP	p. 80 2,2 or 2.3, 2.4-2.7	1/21
3	Independent parallelism	PP 3	any one p. 102 3.1-3.5, 3.7-3.10, 3.12, 3.14	1/28
4	Partioning & Divide&Conquer Strategies	PP 4	2, one integration, one n-body p. 133 4.8-4.21, 4.23	
5	Pipelined computations	PP 5	any one p. 158 5.1-5.6, 5.8-5.10, 5.12, 5.13	2/11
6	Synchronous computations	PP 6	any one p. 191 6.13-6.20, 6.22, 6.23	2/18
7	Load balancing & termination detection	PP 7		

			Hand in fully functioning hard copies of • 1 problem from 3 • 2 problems from 4: one integration, one n-body • 1 problem from 5 • 1 problem from 6 with full documentation and full attribution of source and assistance. Be prepared to explain each line of code and alternative designs. Final	2/21
	Shared memory **PART II**			
8-10	Project	PP 8-12	Choose any three chapters and convince us that you understand and can apply the content.	
	Communication Patterns Grouping Data Communicators and Topologies I/O Design and Coding **Appendix** Parallel Patterns **Obsolete**			
	Principles of Concurrency	FOPP 2		
	Axioms of Flow-correctness	FOPP 3		

Additional concepts

- Protocols
- Fundamental Algorithms
- Fault Tolerance

Resources

- UNCC Web pages
- MPI Forum, MPI at ANL
- HPF - need to find a free version

- BLAS - need to find a source
- LAPACK - need to find a source
- NAG
- PETSc

Outdated resources

- PCN (Parallel composition notation)
- SR (Synchronizing resources)

Last Modified
Send comments to aabyan@wwc.edu

CPTR 460 Parallel and Distributed Computation - 4

Description

Concurrency and synchronization; architectural support; programming language constructs for parallel computing; parallel algorithms and computability; messages vs. remote procedure calls vs. shared memory models, structural alternatives (e.g., master-slave, client-server, fully distributed, cooperating objects); coupling (tight vs. loose); naming and winding; verification, validation, and maintenance issues; fault tolerance and reliability; replication and avoidability; security; standards and protocol; temporal concerns (persistence, serializability); data coherence; load balancing and scheduling; appropriate applications. Prerequisites: CPTR 143, MATH 289. Offered odd years only.

The course will be conducted in a study group setting with group discussion of the reading assignment and presentations of solutions to the design problems posed by the assigned problems. The laboratories explore the implementation of parallel algorithms using MPI

Goals

The goals for this course include:

- understanding the various models of parallelism
- knowing how to design parallel algorithms
- being able to analyze the performance of parallel algorithms
- be proficient in parallel programming in at least one environment.
- ...

Resources

Textbooks:

Wilkinson & Allen (1999) *Parallel Programming* Prentice-Hall

Additional material

Thomas L. Sterling, John Salmon, Donald J. Becker, Savarese, Daniel F. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters* MITPress

Gregory R. Andrews (2000) Foundations of Multithreaded, Parallel, and Distributed Programming Addison-Wesley

Pacheco, Peter (1996) *Parallel Programming with MPI* Morgan Kaufmann

Pfister, Gregory F (1998) *In Search of Clusters* Prentice-Hall PTR

Other Books:

- Akl, S. G. (1989) *The Design and Analysis of Parallel Algorithms* Prentice-Hall -- Focus is on algorithms
- Ben-Ari, M., (1990) *Principles of Concurrent and Distributed Programming* Prentice-Hall
- Chandy, K.M. & Taylor, S., (1992) *An Introduction to Parallel Programming with PCN* Jones and Bartlett -- Design methodology
- Cosnard, M. & Trystram, D., (1995) *Parallel Algorithms and Architectures* International Thomson Computer Press
- East, Ian, (1995) *Parallel Processing with Communicating Process Architecture* UCL Press
- Hartly, S.J., (1995) *Operating Systems Programming* Oxford Univ. Press
- Lester, B.P. (1993) *The Art of Parallel Programming* Prentice-Hall
- Lewis, T.G. & El-Rewini, H. (1992) *Introduction to Parallel Computing* Prentice-Hall
- Quinn (1994) *Parallel Computing: Theory and Practice* McGraw-Hill, New York, New York
- Foster, Ian (1995) *Designing and Building Parallel Programs* Addison-Wesley

Language Manual:

Foster & Tuecke *Parallel Programming with PCN*

Andrews, G.R. & Olsson, R.A., (1993) *The SR Programming Language* Benjamin Cummings

Reading List:

Languages

HPF: High performance Fortran

MPI: the Message-Passing Interface standard

WWW:

http://remarque.berkeley.edu/~muir/free-compilers/

Usenet News Groups:

comp.parallel, comp.parallel.pvm, comp.lang.hermes

Technical Journals:

ACM: TOCS, TOMACS, TOMS, TOPLAS, TOSEAM, Computing Surveys, Communications of the ACM, Journal of the ACM

Reading List

- Kormicki et al (1997) Parallel Logic Simulation on a Network of Workstations Using a Parallel Virtual Machine A*CM DAES* 2, 2 (April 1997), 123-134.

Grading

Last Modified

Send comments to aabyan@wwc.edu

Introduction/Motivation

Black type indicates current course content.
Red type indicates optional material.

Motivational Examples

- Grand challenge problems
  - Global weather forecasting
  - Modeling DNA structures
  - Astrophysical *N*-body simulation
- Idea: With *n* computers a problem could be computed in 1/*n* th the time.
- **Fox's Wall** -- How fast can we build a brick wall?

What is a parallel computer?

A *parallel computer* is either a single computer with multiple internal processors or multiple interconnected computers.

What is parallel programming?

A *parallel program* is a program written to take advantage of a parallel computer. In execution a paralle program is a collection of processes connected to one another through either message-passing or access to shared data.

- Trivially parallel: processes operate independently.
- Control-flow: more than one thread of control (different operations in parallel)
- Data-parallel -- (Example: brick laying)

Why study parallel programming?

- Computer architecture: pipelining (multiple steps), super-scalar (multiple instructions)
- Compiler design
- Parallelism is natural and sequential programming is artificial.
- Quest for speed

How can we build parallel programs?

- "Genius compiler" but can it replace a sequential binary search with a parallel linear search?
- Rewrite all code from scratch

- Write new code.

At what levels can we study parallel and distributed programming?

- Design of parallel programs (fundamental concepts).
- Analysis of parallel algorithms (analytic measures of performance).
- Implementation of parallel constructs (hardware).

What is this course is about?

The design and construction of parallel programs.

1. Fundamental constructs for the expression of parallelism.
2. Analytical measures of performance.
3. Machine independence (because hardware is so variable). N processors and an interconnection network.
4. Language independence (because language design has not stabilized).
5. Concepts:
    - fine (statement level) and large (procedure level) grained parallelism
    - data distribution
    - synchronization
    - tasking
    - allocation of tasks to processors
    - trade-off between communication and computation

How do we do parallel problem solving?

1. Understand Parallel Hardware
    1. Hardware: interconnect processors and memory modules
    2. System Software: design and implement system software
2. Problem Solving
    1. Problem: Design algorithms and data structures
    2. Partition the algorithms and data structures into subproblems
    3. Identify the communication requirements
    4. Assign subproblems to processors and memory modules.

Distributed Systems

A distributed system is an interconnected collection of autonomous computers, processes, or processors. The computers, processes, or processors are referred to as the *nodes* of the distributed system.

The characteristics of a distributed system include

- Resource sharing
- Information exchange
- Increased reliability through replication
- Increased performance through parallelization
- Simplification of design through specialization

Computer Networks

Differences between wide-area and local-area networks

- Reliability parameters (low, high)
- Communication time (slow, fast)
- Homogeneity (low, high)
- Mutual trust (low, high)

Wide-area networks (point to point): Algorithmical issues

- Reliability of point-to-point data exchange
- Selection of communication paths (routing)
- Congestion control
- Deadlock prevention
- Security

Local-area networks: Algorithmical issues

- Broadcasting and synchronization
- Election
- Termination detection
- Resource allocation
- Mutual exclusion
- Deadlock detection and resolution
- Distributed file maintenance

Multiprocessor computers: Algorithmical issues

- Implementation of a message-passing system
- Implementation of a virtual shared memory
- Load balancing
- Robustness against undetectable failures

Cooperating Processes (shared memory)

- Atomicity of memory operations
- The producer-consumer problem
- Garbage collection

Primitives for shared memory system

- Semaphores
- Monitors
- Pipes
- Message passing

Software Architecture

Distributed Algorithms

Differences between distributed and centralized algorithms

- Lack of knowledge of global state
- Lack of global time-frame
- Non-determinism

Last Modified
Send comments to aabyan@wwc.edu

Parallel Computers

A parallel computer is either a single computer with multiple internal processors or multiple interconnected computers.

Black type indicates current course content.
Red type indicates optional material.

Fox's Wall

- Sequential
- Pipeline
- ...

Processes

- Process: single flow of control through a set of instructions
- Processor: hardware device for executing
- Parallel computer: two or more processors connected through an interconnection network.

Flynn's Taxonomy

System classification by number of instruction and data streams.

- SISD: classical sequential von Neumann machine. Inherently sequential. Parallelism may be simulated by interleaving instructions & multiprogramming.
- Pipelining and vector architectures
- SIMD: synchronous since there is a single instruction stream, each processor has its own data stream. Matrix operations are a good example. Thinking Machines - CM, Maspar Computer Corp -- MP (single sequencing units)
- MISD: does not seem to be useful
- MIMD/SPMD: asynchronous processes but with occasional pauses to synchronize; Intel iPSC, nCUBE, Sequent Symmetry, SGI Onyx, SUN MP system
    - shared-memory (sometimes called multiprocessors) locking and protection mechanism
    - distributed-memory (sometimes called multicomputers) message passing
- SPMD - *single program multiple data* - the program may be partitioned so that some parts are executed by certain computers and not others.

Shared Memory Multiprocessor System (MIMD)

Shared memory multiprocessor systems have multiple processors but memory is a single address

space.

- SMP - symmetric multiprocessing
- Bus-based architectures
- Cache coherence -- for bus based systems use the snoopy protocol
- Switch-based architectures, crossbar switch
- NUMA - nonuniform memory access

Distributed-Memory (MIMD) - message passing

Key issues in network design

- bandwidth - bits/second
- latency - total time to send a message
- cost -
- diameter - minimum number of links between the two farthest nodes in the network
- bisection width - number of links (or wires) that must be cut to divide the network in two halves. Used to determine minimum number of messages that must be transmitted.

Connection schemes - often related to the structure of an algorithm

- Static interconnection networks - *direct physical links between computers*.
    - completely connected - impractical for engineering and economic reasons when n is large.
    - array
        - linear array, ring - pipelined computations
        - 2D, torus & 3D mesh - scientific and engineering problems with a natural mesh structure
    - hypercube - diameter is $\log_2 n$
    - bus
    - *embedding* - a mapping of node of one network onto another network.
- Dynamic interconnection networks
- Clusters
- NOW - networks of workstations

Communication and routing

- message transmission
    - *circuit switching* - establish a path and maintain the links until the communication is complete
    - *packet switching* - break message into packets and packets flow through the network
    - *wormhole routing*
- livelock - message circulates without reaching destination
- deadlock - cycle of waiting packets

I/O

Network of Workstations (NOWs) and Clusters of Workstations (COWs)

1. Very high performance at low cost
2. Easy to upgrade
3. Based on existing software

Interconnection

- Performance issues
- Ethernet bus - thick net, thin net, hubs
- Switched ethernet
- Multiple ethernets

Software Issues

- process creation -- static, dynamic
- Programming paradigms
  - Shared-memory programming
    - critical section
    - mutual exclusion
    - binary semaphore
    - barrier
  - Message passing
    - send, receive
    - synchronous
    - asynchronous, buffered
    - blocking and nonblocking communication
- Data parallelism
- RPC, client-server
- Data mapping and load balancing
  - block mapping
  - cyclic mapping
  - block-cyclic mapping

Resources

MPI Complete Reference
Using MPI
Designing and Building Parallel Programs

PETSc

- ScaLAPACK

Algorithms

- Parallel - centralized control
- Distributed - distributed control/intelligence
  - termination detection

Computational Environment

- MPI - compiler & references
- HPF - compiler & references
- Algorithms
  - Parallel
  - Distributed
- Programming/Software Engineering
- Libraries
- Courses

Fault Tolerant, Client-server, Network, sockets

Last Modified

Send comments to aabyan@wwc.edu

Performance Analysis (Complexity)

Background

The running time of this program is T(n) units if the input has size n.

Problems

1. Clock time -- OS load dependent
2. Execution time -- hardware dependent
3. Number of statements -- language and compiler dependent
4. Size of input -- may be order dependent (sorting)
5. Communication -- I/O is orders of magnitude slower

Asymptotic analysis (order statistics - input has size n)

- average-case
- worst-case
- f(n) is an *upper bound* on running time
- T(n) is O(f(n)) iff for c and n_0, T(n) <= cf(n) whenever n >= n_0.
- g(n) is a *lower bound* on running time:
- T(n) is Omega(g(n)) iff there exists a constant c such that T(n) >= cg(n) infinitely often.

Common functions

Name	**Running Time Function**	**Order**	**Example:** n=256 (instructions) 1 microsec/instruction $1x10^{-6}$ sec/instruction
Constant time	c	O(1)	
Log Log N time	a log log n + b	O(log log n)	0.000003 sec
Log N time	a log n + b	O(log n)	0.000008 sec
Linear time	an + b	O(n)	0.0025 sec
N Log N time	a n log n + b n + c	O(n log n)	0.002 sec
Quadratic time	a n^2 + b n + c	O(n^2)	0.065 sec
Polynomial time	a n^k + ...	O(n^k)	17 sec (k=3)

Exponential	a k^n + ...	O(k^n)	$3.67x10^{61}$ centuries (k=2)

Processes

Granularity

- = the size of a process (lines of code, number of instructions),
- = the size of the computation time between commmunications/synchronizations
- large granularity minimizes process startup time and communication times and reduces paralellism

A granularity metric :

Computation/Communication ratio = Computation time / communication time = t_{comp} / t_{comm}
Maximize ratio while maintaining an acceptable amount of parallelism.

Speedup Factor - S(n)

The measure of relative performance between multiprocessor system and a single processor system.

t_1 = time on one processor
t_n = time on *n* processors
S(*n*) = speedup
S(*n*) = Time on one processor / Time on *n*-processors; (S(*n*) = t_1/t_n)
S(*n*) = Time on one processor / Time on *n*-processor parallel system: (S(*n*) = t_1/t_p)
S(*n*) = runtime of fastest serial algorithm / runtime of parallel algorithm on *n*-processors
S(*n*) = O(sequential algorithm) / O(parallel algorithm)
Linear speedup: S(*n*) = $t_1/(t_1/n)$ = *n* is the maximum speedup.
Superlinear speedup: S(*n*) > O(*n*) occurs where

- serial algorithm is not optimal or
- there is a special feature of the multiprocessor system and
- can occur in search algorithms.

Overhead

1. Processor idle time
2. Extra computations appearing in parallel version, duplicate computations
3. Communication time

Amdahl's Law of Speedup

- constant problem size scaling

- f_s fraction of program that is inherently serial
- $f_p = 1-f_s$ fraction of program that is inherently parallel
- t_s = time of serial fraction
- t_p = time of parallel fraction on *n* processors
- Time to run on n processors = serial time + parallel time i.e.,
  - $t_n = t_s + t_p$
  - $t_n = f_s t_1 + t_1 f_p/n$
- Speedup
  - $S(n) = t_1/t_n$
  - $S(n) = t_1/(f_s t_1 + f_p t_1/n)$
  - $S(n) = t_1/(f_s t_1 + (1-f_s)t_1/n)$
  - $S(n) = n/(1 + (n-1)f_s)$
- Maximum speedup $S(n)_{n->inf} = 1/f_s$
  - for $f_s = 1/2$, speedup is 2
  - for $f_s = 5\%$, speedup is 20

Therefore we should work on providing fast single processor machines.

Efficiency

- Efficiency = $t_1/nt_n = 100xS(n) / n$
  - if E = 1 then linear speedup
  - if E = 1/N then slowdown

Cost

- Cost = (execution time)x(total number of processors used)
- Cost of sequential execution = t_1
- Cost of parallel execution = $nt_1/S(n) = t_1/E$

Scalability

Scalable algorithm: Speedup = O(n)

Gustafson-Barsis' Law

- time constrained scaling

- Amdahl's law does not model scalable algorithms since Speedup = O(1).
- Gustafson-Barsis
  - Recall $f_s + f_p = 1$
  - $S_s(n) = (f_s + nf_p)/(f_s + f_p) = f_s + nf_p = n + (1 - n)f_s$
    - for f_s = 50% and 20 processors, speedup is 10.5
    - for f_s = 5% and 20 processors, speedup is 19.05
  - Normalize T_N to 1; B as before

- T_1 = BT_N + N(1-B)T_N = B + (1-B)N -- serial + parallel times
- Speed-up = N - (N - 1)B -- substitution and rearrangement
- For B = 1/2, speedup is (N + 1)/2

Gustafson-Barsis' law models scalable algorithms since Speed-up = O(N)

Evaluating Parallel Programs

- Parallel execution time
  - $t_{parallel\ time} = t_{computation} + t_{communication}$
  - $t_{startup}$ = time to send a message without data
  - t_{data} = time to send one data word
  - $t_{communication} = t_{startup} + n\ t_{data}$
- Time complexity
  - *Upper bound* O(g(x)): f(x) = O(g(x)) iff there exists c > 0 & x_0 > 0, such that 0 <= f(x) <= cg(x) for all x >= x_0
  - Theta(g(x)) --- f(x) = O(g(x)) iff there exists c_0 > 0, c_1 > 0 & x_0 > 0, such that 0 <= c_0g(x) <= f(x) <= c_1g(x) for all x >= x_0
  - *Lower bound* Omega(g(x)): f(x) = O(g(x)) iff there exists c > 0 & x_0 > 0, such that 0 <= cg(x) <= f(x) for all x >= x_0
- Cost-optimal algorithms
  - n = number of processors
  - cost = $nt_{parallel\ algorithm} = kt_{sequential\ algorithm}$
  - *Cost optimal* if $nt_{parallel\ algorithm} = O(t_{sequential\ algorithm})$
- Time complexity of broadcast/gather
  - Hypercube
  - Tree
  - Mesh
  - Workstation cluster - ethernet bus

Empirical methods

- Elapsed time
- Communication time measurement
    - time(x)

      send(...
      recv(...
      time(y)
      et = (y-x)/2
- Profiling - histogram showing time spent on different parts of the program and is used to identify "hot spots"
- Optimizations

Definitions

- Linear speedup: Speedup = O(N) (N is number of processors); isoefficiency -- E = O(1))

- Overhead W(N)
    - Amdahl's law: Speedup = N/(BN + (1-B) + W(N)) = O(1/W(N))
    - Gustafson-Barsis' law: Speedup = [N - (N - 1)B]/W(N) = O(N/W(N))
- Scalable: Speedup >= O(N)
- Parallel-computable: Speedup = O(N)
- Quasi-scalable: Speedup >= 1

	Amdahl Law	**Gustafson-Barsis Law**
Scalable	W(N) <= O(1/N)	W(N)<= O(1)
Parallel-computable	W(N) = O(1/N)	W(N) = O(1)
Quasi-scalable	W(N) <= O(1)	W(N) <=O(N)

Last Modified
Send comments to aabyan@wwc.edu

Threads, Distributed Systems, and Parallelism

Concurrency

is the

- interleaved execution of atomic actions on a single processor or the
- parallel execution of atomic actions on multiple processors.

Concurrency is a property of execution.

Communication

is the exchange of information using

- shared variables and/or
- message passing.

Synchronization

is the coordination of activity using

- mutual exclusion (& critical sections), and/or
- conditional synchronization.

Paradigms of concurrency

- Multithreaded systems - shared memory with more processes than processors
  - Pthreads
  - Java
  - OpenMP
  - Cilk
- Distributed systems - distributed memory and processors
  - MPI
  - Java
  - Orca
- Parallel systems - data parallel applications where speedup is the primary goal

- HPF

Concurrency in Programming Languages: *Recall: a program is a specification of a computation. A programming language is a notation for specifying computations.*

- Imperative languages require ***explicit*** constructs to specify concurrency, communication, and synchronization.
- Declarative languages provide ***implicit*** concurrency, communication, and synchronization so concurrency is a property of execution not of notation.

Programming patterns/paradigms for concurrency

- General patterns of concurrency
  1. data parallel (same task, different data)
  2. task parallel (different tasks, same or different data)
- Application patterns
  1. Iterative parallelism; e.g. matrix multiplication
  2. Recursive parallelism; e.g. adaptive quadrature

     Note: the difference between iterative and recursive parallelism is one of style.
  3. Producers and consumers (pipeline); e.g. unix pipes

     Note: sequential programs are producers and consumers whose stream consists of a single element.
  4. Clients and servers; e.g. file systems
  5. Interacting peers; e.g. distributed matrix multiplication

	Data Parallel	Task Parallel
Iterative Parallelism	x	
Recursive Parallelism		x
Producers & Consumers		x
Clients and servers		x
Interacting peers	x	x

Multithreaded, distributed, and parallel models

System	**Hardware**	**Application**
Multithreaded	shared memory; # processes > # processors	data parallel iterative parallelism recursive parallelism task parallel

Distributed	distributed memory and processors	producer-consumer client-server interacting peers task parallel
Parallel	multiprocessor with shared memory	data parallel iterative parallelism recursive parallelism

Questions

1. Classify each application pattern in the appropriate general pattern.

Concurrent Programming

The root of all successful human organization is co-operation not competition. Concurrent programming is characterized by programming with more than one process.

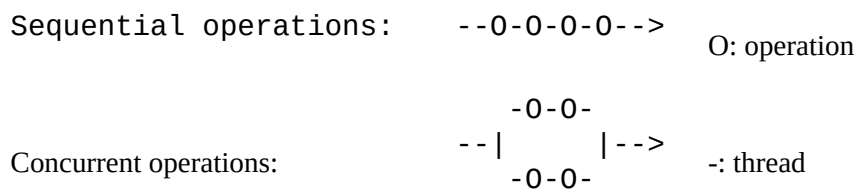
Keywords and phrases Pipelines, parallel processes, message passing, monitors, concurrent programming, safety, liveness, deadlock, live-lock, fairness, communication, synchronization producer-consumer, dining philosophers.

There are several reasons for a programmer to be interested in concurrency:

1. To better understand computer architecture (it has a great deal of concurrency with pipelining (multiple steps) and super-scalar (multiple instructions)) and
2. compiler design,
3. some problems are most naturally solved by using a set of co-operating processes,
4. A sequential solution constitutes over specification, and
5. to reduce the execution time.

At the machine level, operations are *sequential*, if they occur one after the other, ordered in time. Operations are *concurrent*, if they overlap in time. In Figure 1, sequential operations are connected by a single thread of control while concurrent operations have multiple threads of control.

Figure 1: **Sequential and Concurrent Operations**



Operations in the source text of a program are *concurrent* if they could be, but need not be, executed in parallel. Thus concurrency occurs in a programming language when two or more operations could be but need not be executed in parallel. In Figure 2a the second assignment depends on the outcome of the first assignment while in Figure 2b neither assignment depends on the other and may be executed concurrently.

Figure 2: **Sequential and Concurrent Code**

a. not concurrent	b. concurrent
X := 5;	X := A*B + C;
Y := 3*X + 4	Y := 3*A + 7;

Concurrent programming involves the notations for expressing potential parallelism so that operations may be executed in parallel

and the techniques for solving the resulting synchronization and communication problems. Notations for explicit concurrency are a program structuring technique while parallelism is mode of execution provided by the underlying hardware. Thus we can have parallel execution without explicit concurrency in the language. We can have concurrency in a language without parallel execution. This is the case when a program (with or without explicit concurrent sections) is executed on a single processor. In this case, the program is executed by interleaving executions of the concurrent operations in the source text.

> **Aside.** The terms, *concurrent*, *distributed* and *parallel* have a been used at various times to describe various types of concurrent programming. Multiple processors and disjoint or shared store are implementation concepts and are not important from the programming language point of view. What matters is the notation used to indicate concurrent execution, communication and synchronization.
>
> Functional and logic programming languages do not necessarily need explicit specification of concurrency and, with a parallelizing compiler, may be executed on parallel hardware. It is important to note that the notion of processes is orthogonal to that of inference, functions and assignments.

The two fundamental concepts in concurrent programming are processes and resources. A *process* corresponds to a sequential computation with its own *thread* of control. Concurrent programs are distinguished from sequential programs in that, unlike sequential programs, concurrent programs permit multiple processes. Processes may share resources. Shared *resources* include program resources -- data structures and hardware resources -- CPU, memory, & I/O devices.

> **Aside.** Processes which share an address space are called *threads* or *light-weight processes*. For some programming languages (C, C++) there are threads packages to permit concurrent programming. In other cases, the operating system (Microsoft Windows NT, Sun Solaris) provides system calls for threads. Processes which do not share an address space are called *heavy-weight processes*. The Unix family of operating systems provide a system call to allow programmers to create heavy-weight processes.

The Concurrent Nature of Systems

Co-operation

The Bakery. busy waiting, fairness, liveness

The Secretary. scheduling, priority, co-operative multitasking, interrupts, competitive multitasking, pre-emptive multitasking

The Secretarial Pool parallel tasking

Geoffrey Fox's wall. the construction of a brick wall by a number of workers.

Questions:

- How do we break down the task to extract maximum parallelism?
- Wow do we get the task done in the shortest possible time with a given number of workers.
- What is the minimum amount of supervision needed?
- Can all workers be kept equally busy?
- Does the task demand specialized workers?
- Can we maintain efficiency as either the size of the problem or the number of workers grows?

The Nature of Concurrent Systems

Abstraction

Performance

Communication

In the previous solution, it was assumed that the processes shared the address space and that synchronization was achieved by the use of monitor and condition queues. If the address spaces are disjoint, then both communication and synchronization must be achieved through message passing. There are two choices, message passing can be synchronous or asynchronous. When message passing is asynchronous, synchronization can be obtained by requiring a reply to a synchronizing message. In the examples that follow, synchronized message passing is assumed.

Behavior

Synchronization and Communication

Two processes are said to communicate if an action of one process must entirely precede an action of a second process. *Synchronization* is related to communication.

Live-lock may result if there are more than one waiting process and when the signal is received access is not granted fairly.

Starvation: (live-lock) multiple processes waiting for access but access is not provided in a fair manner

Coroutines.

Real-time Programming language issues

When message passing is asynchronous, synchronization can be obtained by requiring a reply to a synchronizing message. In the examples that follow, synchronized message passing is assumed.

Communication commands in the guards. Most communication based programming languages permit input commands in the guards but not output commands. The asymmetry is due to the resulting complexity required to implement output commands in the guards.

```
process Q;
const qsize = 10;
var head, tail : integer;
   queue : array[0..qsize-1] of integer;

begin
    head,tail := 0,0;
    *[ head != tail, C?X --> C!queue[head]; head := (head + 1) mod qsize
      [] head != (tail+1) mod qsize, P?X --> queue[tail],tail := X, (tail + 1) mod
qsize]
end;

process P;
begin
   *[ true --> produce(X); Q!X]
end;

process C;
begin
   *[ true --> Q!X, Q?X; consume(X)]
end;


begin
[ P || C || Q ]
end.
```

Nondeterminism

A program is *deterministic* if its evaluations on the same input it always produce the same output. The evaluation strategy might not always be unique.

A program is *nondeterministic* if it has more than one allowable evaluation strategy and different evaluation strategies lead to different results.

A concept related to nondeterminism is *parallel evaluation* Parallel evaluation that does not involve interaction on the part of its subparts is called *noninterfering parallelism*. Processes which have disjoint address spaces cannot interfere with each other and thus can operate without fear of corrupting each other. For example, the two processes in

[|| i:=1, j:=2]

do not share an address space therefore, the assignments may take place in parallel.

Another example of non-interfering processes is found in matrix multiplication. When two matrices are multiplied, each entry in the product matrix is the result of multiplying a row times a column and summing the products. This is called an inner product. Each inner produce can be computed independently of the others. Figure~\ref{cp:mm}

Figure M.N: **Matrix Multiplication**

```
# multiply n by n matrices a and b in parallel
# place result in matrix c
# all matrices are global to multiply
process multiply( i := 1 to n, := 1 to n)
   var inner_prod := 0
   fa k := 1 to n ->
      inner_prod := inner_prod + a[i,k]*b[k,j]
   af
   c[i,j] := inner_prod
end
```

---

is an example of a matrix multiplication routine written in the SR programming language. This particular example also illustrated dynamic process creation in that {\tt n^2} processes are created to perform the multiplication.

In interfering parallelism, there is interaction and the relative speeds of the subparts can affect the final result.

Processes that access a common address space may interfere with each other. In this program,

[i:=1 || i:=2]

the resulting value of i could be either 1 or 2 depending on which process executed last and in this program,

[i:=0;i:=i+1 || i:=2]

the resulting value of i could be either 1, 2 or 3.

A language is *concurrent* if it uses interfering parallelism.

Sequential programs are nearly always deterministic. A deterministic program follows a sequence of step that can be predicted in advance. Its behavior is reproducible and thus, deterministic programs are testable. Concurrent programs are likely to be nondeterministic because the order and speed of execution of the processes is unpredictable. This makes testing of concurrent programs a difficult task.

The requirement for disjoint address space may be too severe a requirement. What is required is that shared resources may need to be protected so that only one process is permitted access to the resourse at a time. This permits processes to cooperate, sharing the resource but maintaining the integrity of the resource.

Mutual Exclusion

Often a process must have exclusive access to a resource. For example, when a process is updating a data structure, no other process should have access to the same data structure otherwise the accuracy of the data may be in doubt. The necessity to restrict access is termed *mutual exclusion* and involves the following:

- At most one process has access
- If there are multiple requests for a resource, it must be granted to one of the processes in finite time.
- When a process has exclusive access to a shared resource it release it in finite time.
- When a process requests a resource it must obtain the resource in finite time.
- A process should not consume processing time while waiting for a resource.

There are several solutions to the mutual exclusion problem. Among the solutions are semaphores, critical regions and monitors.

Deadlock

Deadlock is a liveness problem; it is a situation in which a set of processes are prevented from making any further progress by their mutually incompatible demands for additional resources. For example, in the dining philosophers problem, deadlock occurs if each philosopher picks up his/her left fork. No philosopher can make further progress.

Deadlock can occur in a system of processes and resources if, and only if, the following conditions all hold together.

- *Mutual exclusion*: processes have exclusive access to the resources.
- *Wait and hold*: processes continue to hold a resource while waiting for a new resource request to be granted.
- *No preemption*: resources cannot be removed from a process.
- *Circular wait*: there is a cycle of processes, each is awaiting a resource held by the next process in the cycle.

There are several approaches to the problem of deadlock. A common approach is to *ignore* deadlock and hope that it will not happen. If deadlock occurs, (much as when a program enters an infinite loop) the system's operators abort the program. This is not an adequate solution in highly concurrent systems where reliability is required.

A second approach is to allow deadlocks to occur but *detect* and *recover* automatically. Once deadlock is detected, processes are selectively aborted or one or more processes are *rolled back* to an earlier state and temporarily suspended until the danger point is passed. This might not an acceptable solution in real-time systems.

A third approach is to *prevent* deadlock by weakening one or more of the conditions. The wait-and-hold condition may be modified to require a process to request all needed resources at one time. The circular-wait condition may be modified by imposing a total ordering on resources and insisting that they be requested in that order.

Another example of a liveness problem is *live-lock* (or lockout or starvation). Live-lock occurs when a process is prevented from making progress (other processes are running). This is an issue of *fairness*.

Scheduling

When there are active requests for a resource there must be a mechanism for granting the requests. Often a solution is to grant access on a first-come, first-served basis. This may not always be desirable since there may be processes whose progress is more important. Such processes may be given a higher *priority* and their requests are processed first. When processes are prioritized, some processes may be prevented from making progress (such a process is *live-locked*). A *fair* scheduler insures that all processes eventually make progress thus preventing *live-lock*.

Semantics

Parallel processes must be... \begin{enumerate}
- Synchronization-coordination of tasks which are not completely independent.
- Communication-exchange of information
- Scheduling-priority,
- Nondeterminism-arbitrary selection of execution path \end{enumerate} Explicit Parallelism (message passing, semaphores, monitors) Languages which have been designed for concurrent execution include Concurrent Pascal, Ada and Occam. Application areas are typically operating systems and distributed processing. Ensemble activity

Concurrency in Programming Languages

Threads/Communication/Metaphor

From the programmer's point of view, concurrent programming notations allow programs to be structured as a set of possibly interactive processes. Such an organization is particularly useful for operating systems, real-time control systems, simulation studies, and combinatorial search applications.

To permit the effective use of multiple processes, concurrent programming languages must provide notations for:

1. *Concurrent execution*: A notation that denotes operations that could be, but need not be, executed in parallel.

   PCN: [|| P_1, P_2, ..., P_n]

   Occam:
   ```
   PAR
      P1
      ...
      Pn
   ```

2. *Communication:* A notation that permits processes to exchange information either through shared variables (visible to each process) or a message passing mechanism.

   Shared Memory
   - *Assignment*: X := E

   Message Passing
   - *Synchronous* P_i!E, P_j?X
   - *Asynchronous* P_i!E, P_j?X

   Remote procedure call

3. *Synchronization*: A notation to require a process to *wait* for a *signal* from another process. In general processes are not independent. Often a process depends on data produced by another process. If the data is not available the process must wait until the data is available.

   wait(P_i), signal(P_j)

   A process can change its state to Blocked (waiting for some condition to change) and can signal Blocked processes so that they can continue.

   In this case, the OS must provide the system calls BLOCK and WAKEUP. cking version of a semaphore

```
type semaphore = record
                     value : integer;
                     L : list of processes; // or queue blocked waiting for
                 end;                       // the signal

down(S): S.value := S.value - 1;  // wait
         if S.value < 0 then
            add this process to S.L;
            block;
         end;

up(S): S.value := S.value + 1;    // signal
       if S.value <= 0 then
          remove a process P from S.L;
          wakeup(P);
       end;
```

Implementation
- *Single processor:* The normal way is to implement the semaphore operations (up and down) as system calls with the OS disabling the interrupts while executing the code.
- *Multiprocessor:* Each semaphore should be protected by a lock variable, with the TSL instruction used to be sure that only one CPU at a time examines the semaphore. Using the TSL instruction to prevent several CPUs from accessing the semaphore at the same time is different from busy waiting.

In many applications it is necessary to order the actions of a set of processes as well as interleave their access to shared resources. common address space, critical section protected by a monitor, synchronization provided through wait and signal.

Some alternative synchronization primitives are

- Semaphores
- Critical Regions
- Monitors
- Synchronized Message Passing

4. *Mutual exclusion*: A notation to synchronize access to shared resources.
semaphores
Monitors: One approach is to protect the critical section by a monitor. The monitor approach requires that only one process at a time may execute in the monitor.

```
monitor Queue_ADT
const qsize = 10;
var head, tail : integer;
   queue : array[0..qsize-1] of integer;
   notempty, notfull : condition;
procedure enqueue (x : integer);
   begin
      [ head=(tail+1) mod qsize --> wait(notfull)
      [] head!=(tail+1) mod qsize --> skip];
      queue[tail],tail := x, (tail + 1) mod qsize
      signal(notempty)
   end;
procedure dequeue (var x : integer);
   begin
      [ head=tail --> wait(notempty)
      [] head!=tail --> skip];
      x,head := queue[head],(head + 1) mod qsize;
      signal(notfull)
   end;
begin
    head,tail := 0,0;
end;
begin
 [ produce(x); enqueue(x) || dequeue(y); consume(y) || dequeue(y); consume(y)]
end.
```

- Correctness (safety and liveness)
- Performance
- Architecture
- Implementation

Aside.

concurrency: Fork (P) & Join (P)

combined notation for communication and synchronization C, Scheme, Ada, PVM, PCN, SR, Java and Occam are just some of the programming languages that provide for processes.

Producer-Consumer

In the following program there is a producer and a consumer process. The producer process adds items to the queue and the consumer process removes items from the queue. The safety condition that must be satisfied is that the head and tail of the queue must not over run each other. The liveness condition that must be satisfied is that when the queue contains an item, the consumer process must be able to access the queue and when the queue contains space for another item, the producer process must be able to access the queue.

```
const qsize = 10;
var count:integer;
   queue : array[0..qsize-1] of integer;
procedure enqueue (x : integer);
   begin
       *[ head=(tail+1) mod qsize --> skip];
      queue[tail],tail := x, (tail + 1) mod qsize
   end;
procedure dequeue (var x : integer);
   begin
       *[ head=tail --> skip];
      x,head := queue[head],(head + 1) mod qsize
   end;
begin
 head,tail := 0,0;
 [ *[produce(x); enqueue(x)] || *[dequeue(y); consume(y)]]
end.
```

Since the processes access different portions of the queue and test for the presence or absence of items in the queue before accessing the queue, the desired safety properties are satisfied. Note however, that busy waiting is involved.

Shared Memory Model

Process Creation

- Static
- Dynamic

Process Identification

- Named
- Anonymous

Synchronization

- Semaphore
- Monitor

In many applications it is necessary to order the actions of a set of processes as well as interleave their access to shared resources. common address space, critical section protected by a monitor, synchronization provided through wait and signal.

Some alternative synchronization primitives are

- Semaphores
- Critical Regions
- Monitors
- Synchronized Message Passing

If in the previous example another process where to be added, either a producer or a consumer process, an unsafe condition could result. Two processes could compete for access to the same item in the queue. The solution is to permit only one process at a time to access the enqueue or dequeue routines. One approach is to protect the critical section by a monitor. The monitor approach requires that only one process at a time may execute in the monitor. The following monitor solution is incorrect.

```
monitor Queue_ADT
   const qsize = 10;
   var count:integer;
      queue : array[0..qsize-1] of integer;
   procedure enqueue (x : integer);
      begin
          *[ head=(tail+1) mod qsize -> skip];
         queue[tail],tail := x, (tail + 1) mod qsize
      end;
   procedure dequeue (var x : integer);
      beg\=in
          *[ head=tail -> skip];
         x,head := queue[head],(head + 1) mod qsize
      end;
   begin
       head,tail := 0,0;
   end;
begin
    [ produce(x); enqueue(x) $\parallel$ dequeue(y); consume(y) $\parallel$
dequeue(y); consume(y)]
end.
```

Note that busy waiting is still involved and further once a process is in the monitor and is waiting, no other process can get in and the program is {\it deadlocked}.

Message Passing Model

Process Creation

- Static
- Dynamic

Process Identification

- Named
- Anonymous

Message Passing

- Synchronous
- Asynchronous

Data Flow

- Unidirectional
- Bidirectional

MPI

Hardware

Processes

- Process: single flow of control through a set of instructions
- Processor: hardware device for executing
- Parallel computer: two or more processors connected through an interconnection network.

Flynn's Taxonomy

System classification by number of instruction and data streams.

- SISD: classical sequential von Neumann machine. Inherently sequential. Parallelism may be simulated by interleaving instructions & multiprogramming.
- Pipelining and vector architectures
- SIMD: synchronous since there is a single instruction stream, each processor has its own data stream. Matrix operations are a good example. Thinking Machines - CM, Maspar Computer Corp -- MP (single sequencing units)
- MISD: does not seem to be useful
- MIMD/SPMD: asynchronous processes but with occasional pauses to synchronize; Intel iPSC, nCUBE, Sequent Symmetry, SGI Onyx, SUN MP system
    - shared-memory (sometimes called multiprocessors) locking and protection mechanism
    - distributed-memory (sometimes called multicomputers) message passing

Shared-Memory MIMD

- Bus-based architectures
- Cache coherence -- for bus based systems use the snoopy protocol
- Switch-based architectures, crossbar switch
- NUMA - nonuniform memory access

Distributed-Memory MIMD

- Dynamic interconnection networks
- Static interconnection networks
    - linear array, 2D mesh, 3D mesh
    - ring, torus
    - hypercube
    - bus

Communication and routing

- routing
- store-and-forward

  cut-through

The Engineering of Concurrent Programs

A parallel programming environment must support the following three phases of system behavior specification.

- *Programming* Behavior of processes and their interconnection
- *Network description* Processors and their interconnection
- *Configuration* Mapping of software onto hardware

Programming

> The way to design parallel software is to begin with the most parallel algorithm possible and then gradually render it more sequential ... until it suits the machine on which it is to run.
> *East (1995)*

Chandy and Taylor (1992) define an elegant parallel programming language PCN (Program Composition Notation) based on:

- Shared definition variables (single assignment) -- X=Exp,
- Parallel composition -- [||P_0,...,P_n],

- Choice composition -- [? G_0 -> P_0,..., G_n -> P_n],
- Sequential composition -- [; S_0,...,S_n], and
- Recursion -- *name*(*parameters*) *composition expression*

The definition variable eliminates the indeterminacy problems. Communication is through shared variables which may be streams. Synchronization is achieved by requiring a process that references an undefined variable to wait until it is defined by some other process before continuing. Recursion with parallel composition permits dynamic process creation.

> If a program that uses only parallel and choice composition and definition variables does not have adequate efficiency, ...
> We use the following steps in the introduction of mutables and sequencing into a parallel block.
>
> 1. We order the statements in a parallel block so that all variables that appear on the right-hand sides of definition statements reduce to ground values or tuples, and all guards reduce to the ground values **true** or **false**, give only the definitions established by statements earlier in the ordering. In other words, we order statements in the direction of data flow; statements that write a variable appear earlier than statements that read that variable. Then we convert the parallel block into a sequential block by replacing "||" by ";" retaining the data-flow order of statements.
> 2. Next, we introduce mutables, add assignment statements to our program, and show that the mutable **m** has the same value as the definition variable **x** it is to replace, at every point in the program in which **x** is read - i.e., where **x** appears on the right-hand side of a definition statement or assignment or guard.
> 3. Finally, we remove the definition variables that are replaced by mutables, secure in the knowledge that the mutables have the same value as the definition variables in the statements in which they are read. We must, of course, be sure that mutables shared by constituent blocks of a parallel block are not modified within the parallel block.
>
> *Chandy and Taylor (1992)*

Decomposition

Function decomposition

Break down the task so that each worker performs a distinct function.

Advantages
Disadvantages

- Fewer tasks than workers
- Some tasks are easier than others

Domain decomposition

Divide the domain by the number of workers available.

- Horizontal domain decomposition: group is responsible for the entire project.
- Vertical domain decomposition: Assembly line, pipelining

Communication and Synchronization

Co-operation requires communication. Communication requires a protocol.

Alternation and Competition

Allocate time to multiple tasks.

- Scheduling
- Co-operative multitasking: multi-person game, using the copy machine

- Priority: telephone vs email
- Competitive multitasking: time slice
- Client-server: bakery
- Busy waiting
- Fairness

Correctness

Partial correctness, Total correctness, satisfaction of specifications...

Chandy & Taylor (1992) require

1. Shared mutable variables remain constant during parallel composition.
2. Mutable variables to copied when used in definitions.
3. When defined, definition variables act as constants in assignment.

Lewis (1993) develops a theory of program correctness called *flow-correctness*. Lewis requires for each shared variable:

1. it must be defined before it is referenced,
2. it must be referenced before it is updated, and
3. only one process at a time may (re)define it.

These rules apply only to the dependencies among variables and do not include either total correctness (termination) or logical correctness (satisfaction of specifications).

Correctness issues in the design of concurrent programs fall in one of two categories: *safety* and *liveness*.

- *Safety*: nothing bad will happen. For example, access to a shared resource like a printer requires that the user process have exclusive access to the resource. So there must be a mechanism to provide *mutual exclusion*.
- *Liveness*: something good will happen. On the other hand, no process should prevent other processes from eventual access to the printer. Thus any process which wants the printer must eventually have access to the printer.

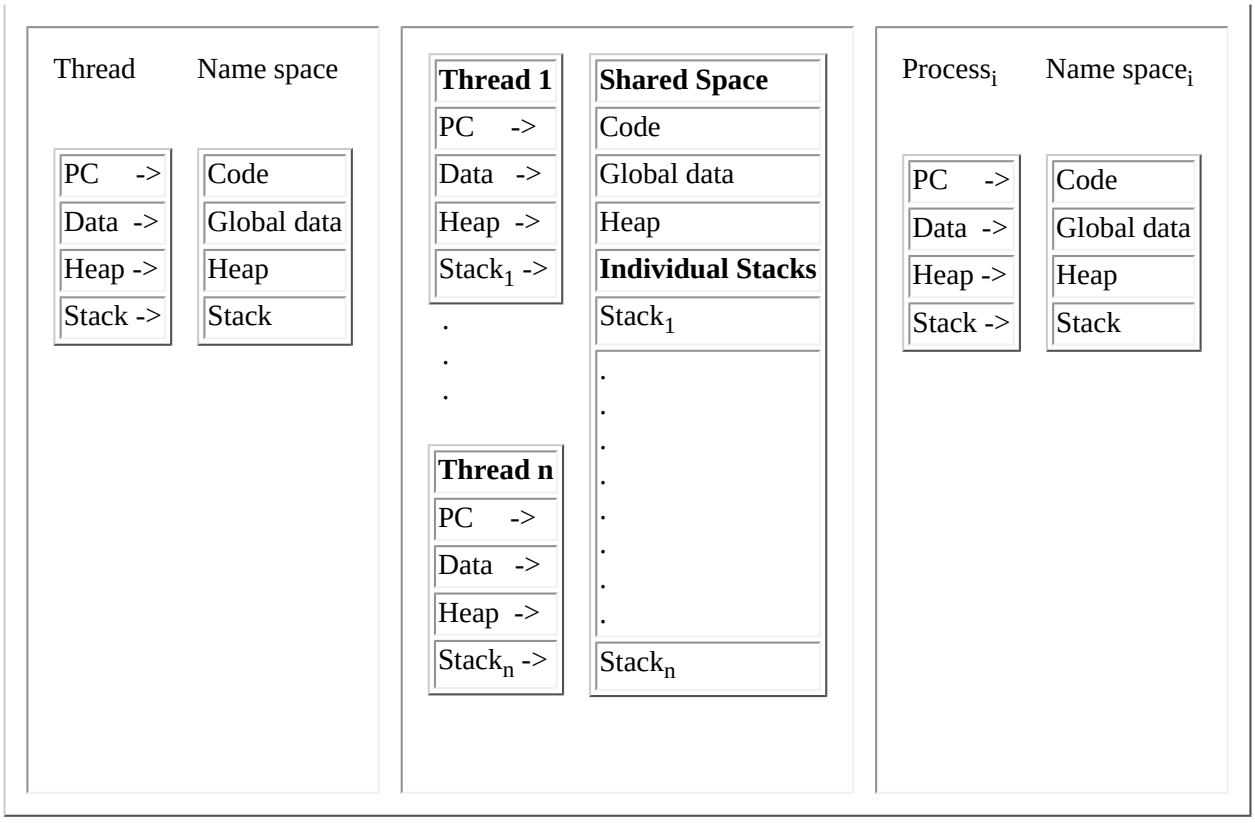
Safety is related to the concept of a loop invariant. A program should produce the ``right" answer. Liveness is related to the concept of a loop variant. A program is expected to make progress. Termination is an example of a liveness property when a program is expected to terminate.

Network description

Configuration

Implementation

Sequential Program	Concurrent Program	
single process/thread	**multiple threads**	**multiple processes**



Historical Perspectives and Further Reading

Related issues: Lazy evaluation vs Parallel execution; Backtracking vs Parallel execution

Concurrency occurs in hardware when two or more operations overlap in time. Concurrency in hardware dates back to the 1950s when special-purpose processors were developed for controlling input/output devices. This permitted the overlapping of CPU instructions with I/O actions. For example, the execution of an I/O instruction no longer delayed the execution of the next instruction. The programmer was insulated from this concurrency by the operating system. The problems presented to the operating systems by this concurrency and the resulting solutions form the basis for constructs supporting concurrency in programming languages. Hardware signals called *interrupts* provided the synchronization between the CPU and the I/O devices.

Other advances in hardware have lead to the the development of alternative architectures. *Pipeline processors* which fetch the next instruction while the first instruction is being decoded. *Super scalar* processors combine multiple pipelines to provide an even greater level of concurrency. *Array processors* provide a large number of identical processors that operate simultaneously on different parts of the same data structure. *Data flow* computers aim at extracting maximum concurrency from a computation by performing as much of the computation in parallel as possible. *Connectionism* based hardware models provide concurrency by modeling computation after the neural networks found in the brain.

Interrupts together with a hardware clock made it possible to implement *multiprogramming* systems which are designed to maximize the utilization of the the computer systems resources (CPU, store, I/O devices) by running two or more jobs concurrently. When one job was performing I/O another job could be executing using the CPU.

Interrupts and the hardware clock also made possible the development of *interactive* systems where multiple users have simultaneous access to the system resources. Such a system must provide for a large number of jobs whose combined demands on the system may exceed the system resources. Various techniques of swapping and paging meet this need by moving jobs in and out of the store to the larger capacity of backing store devices. With the increase of jobs comes the need to increase the capacity of the CPU. The solution was to develop *multiprocessor* systems in which several CPUs are available and simultaneously operate on separate jobs in the shared store.

An alternate solution is to develop *distributed* systems consisting of several complete computers (each containing both CPU and an associated store) that can operate independently but also communicate efficiently. Such systems of local area networks permit the efficient use of shared resources (such as printers and large backing store via file servers) and increase the computational throughput of the system.

Andrews, Gregory R. and Olsson, Ronald A. (1993)
The SR Programming Language, Benjamin/Cummings, Redwood City, CA.
Ben-Ari, M. (1990)
Principles of Concurrent and Distributed Programming, Prentice Hall International, Hemel Hempstead, Hertfordshire.
Chandy, K. Mani and Taylor, Stephen (1992)
An Introduction to Parallel Programming Jones and Bartlett, Boston.
East, Ian. (1995)
Parallel Processing with Communicating Process Architecture, UCL Press, London, England.
Foster, I. (1996)
Compositional Parallel Programming Languages *TOPLAS* Vol 18 No. 4 (July 1996): pp. 454-476.
Hehner, Eric C. R. (1993)
A Practical Theory of Programming Springer-Verlag, New York.
Lewis, Ted G. (1993)
Foundations of Parallel Programming: A Machine Independent Approach IEEE Computer Society Press, Los Alamitos, CA.
Pacheco, Peter S. (1997)
Parallel Programming with MPI Morgan Kaufmann Publishers Inc., San Francisco, CA.
Watt, David A. (1990)
Programming Language Concepts and Paradigms, Prentice-Hall International, Hemel Hempstead, Hertfordshire.

Exercises

For each of the following problems identify the potential for concurrent execution and the synchronization and communication requirements. Define appropriate safety and liveness invariants. Construct solutions using ...

- **Producer-Consumer/Bounded Buffer** (Models race conditions) Producers create data elements which are placed in a buffer. The consumers remove data elements from the buffer and perform some internal computation. The problem is to keep the producer from overwriting full buffers and the consumer from rereading empty buffers.
- **Readers and Writers** (Models access to a database) A data object is shared among several concurrent processes. Some of which only want to read the content of the shared object, whereas others want to update (read and write) the shared object. The problem is insure that only one writer at a time has access to the object. Readers are processes which are not required to exclude one another. Writers are required to exclude every other process, readers and writers alike.
- **The Dining Philosophers.** (Models exclusive access to limited resources) N philosophers spend their lives seated around a circular table thinking and eating. Each philosopher has a plate of spaghetti and, on each side, shares a fork his/her neighbor. To eat, a philosopher must first acquire the forks to its immediate left and right. After eating, a philosopher places the forks back on the table. The problem is to write a program that lets each philosopher eat and think.

  The philosophers correspond to processes and the forks correspond to resources.

  A safety property for this problem is that a fork is held by one and only one philosopher at a time. A desireable liveness property is that whenever a philosopher wants to eat, eventually the philosopher will get to eat.

  1. Solve the dining philosophers problem using a central fork manager (centralized).
  2. Solve the dining philosophers problem where there is a manager for each fork(distributed).
  3. Solve the dining philosophers problem where the philosophers handle their own forks (decentralized).
  4. Solve the dining philosophers problem if the philosophers must acquire all the forks in order to eat (distributed mutual exclusion).
- **Sleeping Barber** The barber shop has one barber, a barber chair, and *n* chairs for waiting customers. The problem is to construct an appropriate simulation.
- **Searching**
  1. Find the largest element in an unordered list
- **Sorting**
  1. Merge sort: Your program should break the list into two halves and sort each half concurrently. While sorting, the two halves should be concurrently merged.
  2. Parallel merge of sorted lists -- if X[i] should just precede Y[j], then X[i] should appear at Z[i+j-1].
  3. Rank sort: X[i] has rank k if X has exactly k items less than X[i] i.e., X[i] should be placed in position k.
  4. Insertion sort: value is placed into its place in the sorted list.
  5. Exchange/Bubble sort: small values flow left and large values flow right.

6. Quicksort
7. Bitonic sort

- **The N-body problem.** The N-body problem is used in astrophysics to calculate the dynamics of the solar system and galaxies. Each mass in this problem experiences a gravitational attraction by every other mass, in proportion to the inverse square of the distance between the objects.
- **The sieve of Eratosthenes.** The sieve of Eratosthenes is a method of generating prime numbers by deleting composite numbers. This is done by the following beginning with two as the first prime:
    1. Delete all multiples of the prime number other than the prime number.
    2. Iterate with the next remaining number which is prime.
- **Polynomial Multiplication** -- initialize, form cross-product, sort by power, combine like powers
- **The quadrature problem.** The quadrature problem is to approximate the area under a curve, i.e., to approximate the integral of a function. Given a continuous, non-negative function f(x) and two endpoints l and r, the problem is to compute the area of the region bounded by f(x) the x axis, and the vertical lines through l and r. The typical way to solve the problem is to subdivide the regions into a number of smaller ones, using something like a trapezoid to approximate the area of each smaller region, and them sum the areas of the smaller regions.
- **Matrix Operations.**
    1. Multiplication: AB = C where A is a p \times q matrix, B a q \times r matrix, C a p \times r matrix and C[i,j] = \sum_{k=1}^m A[i,k]B[k,j]
    2. Triangularization: Triangularization is a method for reducing a real matrix to upper-triangular form. It involves iterating across the columns and zeroing out the elements in the column below the diagonal element. This is done by performing the following step for each column.
        1. For each row *r* below the diagonal row *d*, subtract a multiple of row *d* from row *r*. The multiple is m[r,d]/m[d,d]; subtracting this multiple of row d has the effect of setting m[r,d] to zero.
    3. Backsubstitution:
    4. Gaussian elimination: Gaussian elimination with partial pivoting is a method for reducing a real matrix to upper-triangular form. It involves iterating across the columns and zeroing out the elements in the column below the diagonal element. This is done by performing the following three steps for each column.
        1. Select a pivot element, which is the element in column *d* having the largest absolute value.
        2. Swap row *d* and the row containing the pivot element.
        3. For each row *r* below the new diagonal row, subtract a multiple of row *d* from row *r*. The multiple is m[r,d]/m[d,d]; subtracting this multiple of row d has the effect of setting m[r,d] to zero.

    Assume the matrix is non-singular (the divisor is non-zero).
- **Shortest Path** between two vertices of a graph (edges are weighted).
- **Traveling salesman problem.** Find the shortest tour that visits each city exactly once.
- **Dutch national flag.** A collection of colored balls is distributed among N processes. There are at most N different colors of balls. The goal is for the processes to exchange balls so that eventually, for all i, process i holds all balls of color i. The number of balls in the collection is unknown to the processes.
- **Distributed Synchronization**
    1. Write a program that polls N processes for yes or no votes and terminates when at least N/2 responses have been received. Assume N is even.
    2. Repeat the previous exercise, but terminate when a majority of identical responses have been received. Assume N is even.
    3. Random election of a leader amongst n processes. Create n processes. %Let each process flip a coin to decide whether the process wants to %contest the "elections". %Broadcast this to all other processes. Now, each process generates a random number to decide its "vote", and sends the "vote" to the process it is voting for. Each process counts its votes, and broadcasts the results to all other processes. Now everyone knows the leader. (May have to think of starting the process over again in case of a tie, or simply deciding that the process with the larger Id is the leader, or some such thing.) This is a rather silly problem, but it will help your to learn about broadcasts and synchronizing processes, both of which are extremely important for any kind of parallel programming.
- **The eight-queens problem.** The eight-queens problem is concerned with placing eight queens on a chess board in such a way that none can attack another. One queen can attack another if they are in the same row or column or are on the same diagonal.
- **Miscellaneous**
    1. Sum a set of numbers
    2. (Conway) Read 80-character records, write 125 character records. Add an extra blank after each input record. Replace every pair of asterisks (**) with an exclamation point (!).
    3. (Manna and Pnueli) Compute (n k) = n(n-1)...(n-k+1)/k!
    4. (Roussel) Compare the structure of two binary trees
    5. (Dijkstra) Let S and T be two disjoint sets of numbers with s and t the number of elements respectively. Modify S and T so that S contains the s smallest members of S union T and t the t largest members of S union T.

6. (Conway) The game of life
7. (Hoare) Write a disk server that minimizes amount of seek time
8. Show that Lewis' flow-correctness rules are safety or liveness rules.
9. PCN is a *single assignment language* (in a single assignment language, the assignment of a value to a variable may occur just once within a program). In addition, when a program must reference an undefined variable, it waits until the variable becomes defined. Show that PCN programs satisfy Lewis' flow-correctness rules.

Message Passing

The Basics

Programming options include

- Designing a special parallel programming language - **Occam**
- Extending the syntax/reserved words of an existing language - **CC+**, **FortranM**
- Provide a library of message passing procedures for use with an existing language - **PVM**, **MPI**
    1. A method of creating processes on different computers
    2. A method of sending and receiving messages

Process

1. a program in execution
2. a partition (of a program) designated for parallel execution

Process creation

- static - all processes are specified before execution and the system executes a fixed number of processes - Occam, MPI
- dynamic - processes can be created and their execution started during the execution of other processes - PVM

Message passing routines

- send-receive library calls
    - send(*parameterList*)
    - recv(*parameterList*)
- synchronous message passing (*rendezvous, blocking message passing*)
    - procedures return when message transfer is completed
    - no buffer is required
- asynchronous message passing (*nonblocking message passing*)
    - sending procedures returns whether or not the message has been received
    - receiving procedures block until a message is available
    - buffer to hold unread messages is required
- message selection
- broadcast, gather, scatter

NOW - network of workstations

MPI

- process creation and execution
- using the SPMD computation model
- message-passing routines

Evaluating Parallel Programs

performance issues

Maintained by WWC CS Department

Send comments to webmaster@cs.wwc.edu

MPI Tutorial

Overview

MPI is a library of functions and macros that can be used in C programs.

SPMD model: Programs written in the *single-program multiple-data* model may have multiple processess. Each process runs the same executable program however, the processes execute different statements by taking different branches in the program. The branches are determined by the process rank.

- MPI identifiers begin with `MPI_`
- Most MPI constants are all capital letters
- Each MPI function begins with a capital letter and the subsequent characters are lowercase.

The MPI Programming Model

- Communicator: a collection of processes that can send messages to each other.
- Each process has a unique number called its rank.
- The processes share a common program.
- Each process executes those statements that correspond to its rank.
- Each communication includes a send which includes
    - The message, its length and datatype
    - The rank of the receiver process
    - A tag to indicate the class of the message
    - The identity of the communicator.
- Each communication includes a receive which includes
    - The message, its length and datatype
    - The rank of the source
    - The tag which indicates the class of the message
    - The identity of the communicator

      The message status

Syntax

A typical MPI program has the following layout.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

main(int argc, char* argv[]) {
   ...

   int        my_rank;       /* rank of process      */
   int        p;             /* number of processes  */
   int        source;        /* rank of sender       */
   int        dest;          /* rank of receiver     */
   int        tag = 0;       /* tag for messages     */
   char       message[100];  /* storage for message  */
   MPI_Status  status;       /* return status for    */
                             /* receive              */
   ...

   /* Start up MPI -- no MPI functions called before this */
   MPI_Init(&argc, &argv);

   /* Find out my process rank  */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   /* Find out number of processes */
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   if (my_rank != 0) {
      /* Create message */
      sprintf(message, "Greetings from process %d!",
         my_rank);
      dest = 0;
      /* Use strlen+1 so that '\0' gets transmitted */
      MPI_Send(message, strlen(message)+1, MPI_CHAR,
         dest, tag, MPI_COMM_WORLD);
   } else { /* my_rank == 0 */
      for (source = 1; source < p; source++) {
         MPI_Recv(message, 100, MPI_CHAR, source, tag,
            MPI_COMM_WORLD, &status);
         printf("%s\n", message);
      }
   }
   ...
```

```
    /* Shut down MPI -- no MPI functions called after this */
    MPI_Finalize();
    ...

} /* main */
```

Examples

sum.c
rand_data.txt

References

Pacheco, Peter S.
Parallel Programming with MPI Morgan Kaufmann Publishers Inc. 1997

Maintained by WWC CS Department
Last Modified
Send comments to webmaster@cs.wwc.edu

Debugging

We should design mathematically correct programs and, as a consequence, we should never need to do any debugging. -- *Anonymous*

Serial Debugging

Syntax errors: ***Edit-Compile Loop*** (to produce an executable program)

Repeat
1. Edit the program
2. Compile the program

Until there are no more syntactic errors

Semantic errors: ***Test-Debug Loop*** (to produce a working program)

Repeat
1. Run the program
2. If there are errors, do one or more of the following
   - Examine the source code
   - Instrument your program by adding debugging output statements to the program -- print out the values of the variables of interest
   - Add assertions to the program
   - Use a symbolic debugger

Until there are no more bugs

A Debugger

A debugger is a program that acts as an intermediary between the programmer, the system and the program. As stated in the man page for the GNU debugger gdb, debuggers usually provide the following:

- Start your program, specifying anything that might affect its behavior.
- Make you program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Typical bugs

- Infinite loops
- Divide by zero
- Undefined variables

Catalog of debugging techniques
Common programming errors

- Infinite loops
- Divide by zero
- Undefined variables

Tentative diagnoses of common failures

Parallel Debugging

In addition to the problems found in serial programs, parallel programs may be subject to **race** conditions. A race condition occurs when the behavior of a program is dependent on the order or timing of execution or communication. A race condition is an example of **nondeterminism.** A well designed parallel program should be correct regardless of nondeterminism.

The standard techniques for debugging sequential programs are not a productive when used for parallel programs. Intructions inserted to instrument a program can change the timing charactoristics and traditional debuggers for sequential program are not helpful in identifying race conditions.

Error checking code is particularly important in parallel programs to ensure that faulty conditions can be handled and not cause deadlock.

Typical bugs

- Trying to receive data when there has been no send
- Incorrect parameters

Debugging Strategies (Geist)

1. If possible, run the program as a single process and debug as a normal sequential program.
2. Execute the program using two to four multitasked processes on a single computer and check that messages are bing sent to the correct places.
3. Execute the program using two to four processes across several computers. This helps to find problems caused by network delays.

Independent Parallelism

a computation that can be divided into a number of completely indepencent parts, each of which can be executed simultaneously.

Ideal Parallel Computation

- Disconnected computational graph
- Manager-worker with minimal communcation

Examples

- Geometrical transformation of images
  - shifting
  - scaling
  - rotation
  - clipping
- Mandelbrot set
- Monte Carlo Methods
- Parallel random number methods

Author: Anthony A. Aaby
Last Modified - .
Comments and content invited aabyan@wwc.edu

Partitioning and Divide and Conquer Strategies

Functional decomposition:

$$f(x) = g(f_n(x), f_{n-1}(x), \ldots f_1(x), f_0(x))$$

Domain decomposition:

$$\begin{aligned} f(x{:}xs) &= g(x){:}f(xs) \\ f([]) &= [] \end{aligned}$$

Divide and conquer

$$\begin{aligned} f(x) = {} & \text{if terminal}(x) \text{ then } g(x) \\ & \text{else } h(f(x_0), \ldots f(x_n)) \\ & \text{where } \langle x_0, \ldots, x_n \rangle = \text{divide}(x) \end{aligned}$$

Partioning

- JaJa
  - partitioning if main task is dividing the problem (quicksort)
  - divide and conquer if main task is combining the results (mergesort)
- Partitioning strategies
  - functional decomposition
  - data partitioning or domain decomposition
- Divide and conquer
  - tree (binary) structured computation
  - m-ary divide and conquer

Examples

- Bucket sort
  - many sequential sorts are based on the idea of compare and exchange
  - bucket sort is based on the idea of partitioning a range (0, *a)* into *m* regions or buckets

and placing each of the n items into the appropriate bucket, then either sequentially sorting each bucket or recursively applying bucket sort.
    - partioning schemes
        - one bucket/process
        - a/m size region/process; m buckets/process; processes exchange small buckets
- Numerical integration
    - Quadrature - fixed number of intervals
        - rectangles
        - trapezoid rule
        - Simpson's rule
    - Adaptive quadrature - changing number of intervals
- N-body problem - determine the effect of forces between n-bodies
    - Barnes-Hut algorithm
        - octtree
        - orthogonal recursive bisection

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Pipelined Computation

-- the problem is divided into a series of tasks that have to be completed one after the other where

- more than one instance is required for the complete program to be executed, or
- a series of data items must be processed, each requiring multiple operations, or
- information to start the next process can be passed forward before the process has completed all its internal operations.

Fox's wall

- Functional decomposition
  - bricklayer
  - mortar mixer
  - morter/brick carrier
- Data parallel
  - pipeline - one bricklayer/row
  - multiple bricklayers/row -- m-bricks/n-bricklayers

The Pipeline Technique

The pipeline technique is a form of *functional decomposition* where the input data is broken up and each piece is processed independently.

$f([]) = []$
$f(x:xs) = f_n(f_{n-1}(...f_1(f_0(x))...)):f(xs)$ where

$f_i(x:xs) = g_i(x):f_i(xs)$
$f_i([]) = []$

Pipelined solutions are appropriate for

1. Computing more than one instance of a complete problem.
2. A series of data items must be processed, each requiring multiple operations.
3. When information to start the next process can be passed forward before the process has completed all its internal processes.

Computing Platform

```
->P0->P1->...->Pn->
```

Examples

1. Adding numbers - each process receives a partial sum, adds its number to the sum and passes the new partial sum to the next process.
2. Sorting numbers - parallel insertion sort (pass on smaller numbers)
3. Prime number generation (sieve of Erathosthenes) - pass on indivisible numbers
4. System of linear equations (back substitution) - upper triangular form

---

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Synchronous Computations

- processes that must at times wait for each other before preceding, thereby becoming synchronized.

Synchronization

The mechanism that prevents any process from continuing past a specified point until all processes are ready.

- Message passing - explicit barrier

  Scalar * Vector
  ```
  i = myrank;
  a[i] = k*a[i];
  barrier(mygroup);
  ```
- Shared memory (SIMD) - implicit barrier

  Scalar * Vector
  ```
  forall (i = 0; i<n; i++)
     a[i] = k*a[i];
  ```
  Matrix + Matrix
  ```
  forall (i = 0; i<m; i++)
    forall (j = 0; j<n; j++)
      c[i,j] = a[i,j]+ b[i,j];
  ```

  ```
  MPI_Barrier(communicator)
  ```

Implementation

- Counter
- Tree
- Butterfly

Local Synchronization

Deadlock

```
MPI_Sendrecv()MPI_Sendrecv_replace()
```

Synchronized Computation

Data Parallel Computations (SIMD)

Synchronous Iteration - *each iteration is composed of several processes that start together at the beginning of each iteration and the next iteration cannot begin until all processes have finished the previous iteration.*

SIMD

```
for (j=0; j<n; j++)
  forall(i=0; i<N; i++)
    body(i);
```

SPMD

```
for (j=0; j<n; j++){
  i= myrank;
  body(i);
  barrier(mygroup)
}
```

Synchronous Iteration Examples

- Solving a system of linear equations
  - Gauss-Seidel - chapter 10
  - Jacobi iteration
    - solve i-th equation for the i-th variable; x_i = ...
    - initialize $x_i = b_i$
    - iterate in parallel until converged to a solution
    - convergence is guaranteed if $Sum(|a_{i,j}|)_{i!=j} < |x_{i,i}|$
- Simulated annealing - heat distribution problem
  - given the temperatures at points on the edge of a sheet of metal, find the temperatures of the interior points.
    - the temperature of a point is the average of the temperatures of the four neighboring points
    - iterate
      - a fixed number of times or
      - until temperatures stabilize
  - Other applications
    - pressure and voltage
    - solving a system of linear equations
    - finite difference method
    - Laplace's equation
- Cellular automata
  - Game of life
    - Each cell can hold one "organism"
    - n = number of neighboring cells (maximum of 8 - 2D) containing an organism
    - Occupied cell
      - Survival: n in {2,3}
      - Death: n not in {2,3}
    - Empty cell
      - Birth: n = 3
  - Sharks and fishes

        - Each cell can hold "one fish or one shark" (but not both)
        - Fish move by the following rules
            - n = number of unoccupied neighboring cells (maximum of 8 - 2D)
            - n = 0: stay
            - n > 1: choose unoccupied cell at random and move to it.
            - Birth: if of breeding age, leave baby fish behind when moving
            - Death: die after x generations
        - Sharks move by the following rules
            - f = number of fish in neighboring cells (maximum of 8 - 2D)
            - n = number of unoccupied neighboring cells (maximum of 8 - 2D)
            - f = 0: choose unoccupied cell at random and move to it.
            - f > 0: choose a fish occupied cell at random and move to it eating the fish.
            - Birth: if of breeding age, leave baby fish behind when moving
            - Death: die if has not eaten for y generations
    - Foxes and rabbits - 6.20
    - Replacement for differential equations in fluid/gas dynamics
        - movement of fluids and gases around objects (e.g. air flow over a wing)
        - gas diffusion
        - biological growth
        - Erosion of sand dunes at a beach when affected by the waves.
        - Erosion of the banks of a river due to the water.

Mesh partitioning

- one point per process
- square blocks
- rectangular strips (rows or columns)

Communication

Last Modified - . Comments and content invited aabyan@wwc.edu

Load Balancing and Termination

Static load balancing

- round robin
- randomized algorithms
- recursive bisection
- simulated annealing
- genetic algorithm

Dynamic load balancing

- centralized
- decentralized

Centralized dynamic load balancing

Manager-worker or replicated worker or processor farm

- manager
- workpool
- worker

Termination.

When tasks are taken from a task queue, computation terminates when

1. the task queue is empty and
2. every process has made a request to every other process without any new tasks being generated (If worker processes do not generate new tasks, computation terminates when task queue is empty and each worker process has terminated.)

Other termination situations:

- some worker detects termination condition, informs manager who terminates all workers
- all workers reach local termination conditions and inform manager

Decentralized dynamic load balancing

Processes have and generate tasks

1. receiver-initiated
2. sender-initiated

...

Distributed termination detection algorithms

- termination conditions
- using acknowledgement messages
- ring termination algorithms
- tree algorithm
- fixed energy distributed termination algorithm

CPTR 460 Grade Worksheet

Name: **Grade:**

Instructions: Hand in hard copies of fully functioning programs

- 1 problem from 3
- 2 problems from 4: one integration, one n-body
- 1 problem from 5
- 1 problem from 6
- 3 problems from distinct chapters 9-12

with full documentation and full attribution of source and assistance. Be prepared to explain each line of code and alternative designs. Alternatively, place all programs in a public directory and email me with its location.

Program Summary

Chapter	Description	Grade
any problem from 3		
integration from 4		
n-body from 4		
any problem from 5		
any problem from 6		
1st from 9-12		
2nd from 9-12		
3rd from 9-12		
	Total	

Final grade is computed by dividing the total by 8 and using the following table to obtain the letter grade.

Letter Grade	Points/Percent
A	90-100
B	80-90
C	60-80
D	50-60
F	0-50

Resources

- program heading
- program grading criteria

LN: P&DC -- Communication Patterns

- One to one
  - **MPI_Send(message,count,dataype,dest, tag,comm)**
  - **MPI_Recv(message,count,datatype,source,tag,comm,status)**
- Serial distribution & collection -- O(n)
  - one process sends data sequentially to other processes
  - one process receives data sequentially from other processes
- Tree structured (processes are logically structructred as a tree) -- O(log n)
  - data is distributed from the root to the leaves
  - data is passed from the leaves to the root

Collective Communication: All processes in a communicator are involved.

- Broadcast - single process sends the same data to all other processes
  - **MPI_Bcast(message,count,datatype,root,comm)**
    - Sends message from root to each process in the communicator and by the other processes to receive the message.
    - It must be called by *all* processes in the communicator with the same arguments for root.
    - **Example:** trapezoid rule
- Reduction - each process contains an operand which are combined using a binary operator with the result avaliable to the root process.
  - **MPI_Reduce(operand,result,count,datatype,operator,root,comm)**
    - It must be called by *all* processes in the communicator, and count, datatype, operator, and root must be the same on each process.
    - All, except the root, contribute data that is combined using a binary operation.
    - The root process gets the result.
    - **Examples:**
      - trapezoid rule sum
      - dot product
  - **MPI_Allreduce(operand,result,count,datatype,operator,comm)**
    - exactly the same as MPI_Reduce except the result is returned to all processes

  Aside.
  - Note the use of in, out, and in-out on parameter specifications.
  - The syntax is the same for both senders and receivers to simplify the language.
  - *Aliasing* is not permitted on out or in/out arguments on MPI functions.
- Gather - distributed data structure is collected by process rank order onto a root process
  - **MPI_Gather(sentdata,count,datatype,recData,recCount,recType,root,comm)**
  - **MPI_Allgather(sentdata,count,datatype,recData,recCount,recType,comm)**

- Scatter - a data structure is distributed across processes
  - **MPI_Scatter(senddata,count,datatype,recData,recCount,recType,root,comm)**

Last Modified

Send comments to aabyan@wwc.edu

LN: P&DC -- Compound Messages

MPI provides three methods for sending complex data.

1. Arrays with the count parameter
2. build a derived data type
3. use the MPI functions MPI_Pack and MPI_Unpack

The `Count` Parameter

- Suitable for passing contiguous homogeneous elements (arrays)
- Sender: MPI_Send(vector+50, 50, ...);
- Receiver: MPI_Receive(vector+50,50,...)

Derived Types and `MPI_Type_struct`

Only constants and variables may be passed as parameters -- types may not be passed as parameters. So from data types chapter

Product Descriptor	**Values**
	$Descriptor_1$ $value_1$
	... ...
	$Descriptor_n$ $value_n$

MPI_Type_struc(count,blockLengths,displacements,typelist,newTypeName)

- count is the number of domains
- blockLengths is the number of items in a domain
- displacements is the displacements of the fields
- typelist is the list of types (domains)
- newTypeName is the name of the type to be used in refering to message

Pack/Unpack

- Used to send heterogeneous data only once
- Sender

```
position=0;
MPI_Pack(x_1,count,datatype,buffer,size,position,comm);
...
MPI_Pack(x_n,count,datatype,buffer,size,position,comm);
MPI_Bcast(buffer,size,MPI_PACKED,root,comm);
```

- Receiver

```
MPI_Bcast(buffer,size,MPI_PACKED,root,comm);
position=0;
MPI_Unpack(buffer,size,position,x_1,count,datatype,comm);
...
MPI_Unpack(buffer,size,position,x_n,count,datatype,comm);
```

Last Modified
Send comments to aabyan@wwc.edu

LN: P&DC - Communcators and Topology

- Standard matrix multiplication: $c_{ij} = \text{Sum } a_{ik}b_{kj}$ k=0..n
- Fox's Algorithm: $c_{ij} = \text{Sum } a_{i,i+k}b_{i+k,j}$; k=0.. n-1; addition modulo n
- Block variation: $C_{ij} = \text{Sum } A_{i,i+k}B_{i+k,j}$; k=0.. n-1; addition modulo n, A, B, C submatrices
- Communicators
  - intra-communicators
  - inter-communicators
  - **group** - an ordered collection of processes
  - **rank** - a unique number (0,...) assigned to a process in a group
  - **context** - system defined communicator identifier
- Groups, Contexts & Communicators
  - MPI_Comm_group
  - MPI_Group_incl
  - MPI_Comm_create
- Topologies - a mechanism for associating different addressing schemes with processes in a group
  - Cartesian or grid topology
    - The number of dimensions in the grid
    - The size of each dimension
    - The periodicy of each dimension (whether the last entry is adjacent to the first entry
    - Opiimize mapping of virtual topology to the physical processes
    - MPI_Cart_create
      MPI_Comm_rank
      MPI_Cart_coords
    - Partition grid
      MPI_Cart_sub
  - Graph topology

Last Modified
Send comments to aabyan@wwc.edu

LN: P&DC -- I/O

- I/O in parallel programs: problems with nondeterminism
- Process rank 0 - the I/O process
- Collective Operations -- I/O
  - Read/Print by I/O process
  - Broadcast to and received from other processes
- Communicator - collection of processes that can send messages
  - Each process has a unique rank in the communicator
  - Each communicator has a unique context which identifies the communicator
  - Each communicator has a collection of attributes, each identified by a unique key
  - Attributes and attribute keys are process local -- each process may cache different attributes with the same communicator
  - Callback functions - allocate and free memory
- Identifying the I/O process
  - MPI_IO
    - MPI_PROC_NULL
    - MPI_ANY_SOURCE
    - == myRank (I can do I/O)
    - note: no provision has been made to identify processes that can do input
  - MPI_IO does not necessarily identify a unique process
  - How to agree on a unique I/O process

Principles of Parallel Systems and Software Engineering

Motivational examples

- The construction site -- Building a wall: mixing mortar, carring bricks, laying mortar, laying bricks
- Production line --
- Supermarket --
- Matrix multiplication --

Introduction

From Foster 96: Most programming problems have several parallel solutions. The best solution may differ from that suggested by existing sequential algorithms. The design methodology that we describe is intended to foster an exploratory approach to design in which machine independent issues such as concurrency are considered early and machine specific aspects of design are delayed until late in the design process. This methodology structures the design process as four distinct stages: partitioning, communication, agglomeration, and mapping. (The acronym PCAM may serve as a useful reminder of this structure.) In the first two stages, we focus on concurrency and scaleability and seek to discover algorithms with these qualities. In the third and fourth stages, attention shifts to locality and other performance related issues. The four stages which are summarized as follows:

1. ***Partitioning.*** The computation that is to be performed and the data operated on by this computation are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.
2. ***Communication***. The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.
3. ***Agglomeration.*** The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.
4. ***Mapping.*** Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

Algorithm design is presented here as a sequential activity. In practice, however, it is a highly parallel process, with many concerns being considered simultaneously. Also, although we seek to avoid

backtracking, evaluation of a partial or complete design may require changes to design decisions made in previous steps.

Partitioning

The computation that is to be performed and the data operated on by this computation are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.

Function decomposition (control-parallel)

In this approach, the tasks are partitioned among the processes and each process executes a different set of commands. Opportunities for parallel execution arise when the input is structured and may be operated on by multiple functions.
In functional notation $f_n(f_{n-1}(... f_1(f_0(Input))...)) = Output$; or $F(f_0(I_0), ..., f_m(I_n)) = Output$
Its principle advantages are

- some functions may be able to make effective use of specialized resources and
- the allocation of tasks to workers may be done just once with little or no management overhead.

Its principle disadvantages are

- the number of distinct functions is limited
- the functions are different.

Thus it is difficult to utilize additional resources and to have a *load-balanced* solution i.e., function decomposition fails to offer *scaleability of performance with size of either domain or processors.*

Domain decomposition (data-parallel)

In this approach, data is partitioned among the processes and each process executes the same set of commands on the data The scale of the domain is the grain size or granularity.

Its principle advantages are

- it is scalable with respect to number of processes and domain size, and
- load balancing is trivial with little or no management overhead.

Its principle disadvantages are

- the structure of the domain must be regular and
- the work uniform.

Manager-Worker

Manager-worker is a problem solving method where the data is non-uniform and the work required is non-uniform.

It is required whenever any of the following applies:

- Domain structure is irregular and does not match the workforce.
- Work required is distributed non-uniformly across domain.
- Work capability is distributed non-uniformly across workforce.
- No distribution of skills exists to match an efficient decomposition of function with static allocation.

Its principle advantages are

- it is scalable with respect to workers or domain size and
- load balancing is trivial with little or no management overhead.

Its principle disadvantages are

- the requirement for a broadly skilled workforce and
- the management overhead.

Communication

The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.

Synchronization

A protocol a set of rules for communication. interleaved synchronization signal channel of communication message lockstep pipeline latency communication architecture systolic

Alternation

alternate context switch interruption fair

Agglomeration

The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into

larger tasks to improve performance or to reduce development costs.

Mapping

Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

Competition

Time management

scheduling algorithm co-operative multi-tasking priority competitive multi-tasking

Service provision

shared resources client-server queue busy waiting

References

Foster, Ian (1996)
Designing and Building Parallel Programs Addison Wesley
Chandy, K. Mani and Taylor, Stephen (1992)
An Introduction to Parallel Programming Jones and Bartlett, Boston, MA.
Foster, Ian and Tuecke, Steven (1993)
Parallel Programming with PCN Argonne National Laboratory, Chicago, IL.

Program design issues.

Partitioning
a program into concurrent components.
Domain Decomposition.
Divide up the data of a program and operate on the parts concurrently.
Functional Decomposition.
Divide up the function of the problem and operate on the parts concurrently.

1. Jacobi relaxation algorithm for solving Laplace's equation.
2. Red-black relaxation
3. N-body problem -- solved on a ring

Irregular problems.

Mapping

components to computers.

Indexing.

Hashing.

Simulated Annealing.

Load Balancing:

(primarily for irregular problems)

Bin Packing.

Place a partition at the computer with the least data.

Randomization.

Place partitions at a random computer.

Pressure Models.

Move partitions to more lightly loaded neighboring computers.

Manager-Worker.

The manager partitions the problem into components placing them in a central pool of tasks. There is a large number of worker processes that retrieve tasks from the pool, carry out the required computation, and possibly add new tasks to the pool.

Communication and Synchronization

Unbounded Communication:

1. One-to-One (producer/consumer)
2. Broadcasting (producer/consumers)
3. Many-to-One (mergers)
4. One-to-Many (distributors)
5. Two-way Communication

Bounded Communication

Blackboards

Lester

Relaxed Algorithm(2)

Each process computes in a self-sufficient manner with no synchronization or communication between processes.

Synchronous Iteration(6)

Each processor performs the same iterative computation on a different portion of data.

Replicated Workers (10,11)

Pipelined Computation.(4,8)

The processes are arranged in some regular structure such as a ring or two-dimensional mesh. The data flows through the entire process structure with each process performing a certain phase of the computation.

1. Back substitution
2. Numerical Integration:
3. Linear Equations: iterative method %
4. Back substitution
5. Assembler
6. Compiler

Parallel Patterns

Process Creation

- Static process creation
- Dynamic process creation

Message-Passing Computing

1. Special language
2. Language extensions
3. Library routines
    - Basic routines
        - send(...)
        - receive(...)
    - Synchronous message passing
    - Asynchronous message passing
    - Blocking & nonblocking message passing
    - Broadcast, gather, scatter
        -

Functional Style	Imperative Style

Independent parallelism w/static and dynamic process creation

Functional Style	Imperative Style
f(x) = gather($f_0(x_0)$, ... , $f_n(x_n)$) where $<x_0, ... , x_n>$ = scatter(x) f([]) = [] f(x:xs) = g(x):f(xs)	

Functional decomposition

Functional Style	Imperative Style
$f(x) = g(f_n(x), f_{n-1}(x), \ldots f_1(x), f_0(x))$	$\{\| f_0(x), \ldots, f_n(x)\}$

Pipeline

Functional Style	Imperative Style
$f([\,]) = [\,]$ $f(xs) = f_n(f_{n-1}(\ldots f_1(f_0(xs))\ldots))$ where $f_i(x{:}xs) = g_i(x){:}f_i(xs)$ $f_i([]) = []$.	

Domain (data) decomposition

Functional Style	Imperative Style
$f([]) = []$ $f(x{:}xs) = g(x){:}f(xs)$	forall x in xs do g(x);

Divide and conquer

Functional Style	Imperative Style
$f(xs) =$ if stoppingCondition(xs) then baseCase(xs) else gather($f_0(x_0), \ldots, f_m(x_m)$) where $\langle x_0, \ldots, x_m \rangle$ = mWayScatter(xs)	

Synchronous Computations

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Principles of Concurrency

- Synchronization: locks and messages
- separation of policy and implementation

Processes

- Process: locus of control; program counter as it moves through code; path of a token through a Petri network.
- Quanta: time interval
- Concurrent processes: only one process executing in a time quanta
- Parallel processes: more than one process executing in a time quanta
- Preemptive multitasking: concurrent tasks in an operating system
- Critical sections: section of a program that accesses shared data
- Definitions

  Large-grained
  : process execution time > message passing time

  Fine-grained
  : process execution time < message passing time

  Lightweight process
  : process whose setup and runtime overhead is small.

  Heavyweight process
  : process whose setup and runtime overhead is large.

  threads, tasks, processes

Properties of Flow-Correct Processes

- Safety: nothing bad happens.
- Liveness: something good will happen.
- Fairness: no process is starved.

Timing Diagrams

Active: high; inactive: low

Interleave Matrix Analysis

- Axes are labeled with the states of the processes (one process per axes)
- Each square corresponds to a combined state.

- Squares surrounding the combined critical section states are called the safety zone.
- Paths correspond to execution behavior

- Test-and-Set
- Spin-lock
- Queue with wait and Blocking

Bounded Buffers

Path Expressions

- A path is a regular expression that describes a finite state machine for specifying the order of activation of asynchronous processes.
    - AB -- A and B are executed in parallel
    - A+B -- A and B are executed in sequence either A then B or B then A
    - A^n -- n As in parallel
    - [A] -- execution of A is optional
    - (A) -- grouping of processes as in 2(A) meaning A+A
    - * -- 0 or more as in A^*
- Given a path expression and a sequence of requests, the requests are processed according to the path expression.

Axioms of flow-correctness

Flow-correctness in parallel constructs

- data flow temporal predicates
  - UNDEF(x,t) iff x is undefined at time t
  - USE(x,t) iff x is referenced at time t
  - DEF(x,t) iff x is assigned a value at time t

Axioms of data dependencies

- Race conditions
  - variable is referenced before being defined
  - variable is updated by two or more processes in an unpredictable order
  - variable is updated between references when the intention is to update after all references
- Dependencies
  - Output dependency: unpredictable ordering of updates
    OUT(x,t) :- (E dt>0 | DEF(x,t-dt)), DEF(x,t)
  - Flow dependency: update(define) then reference
    FLOW(x,t) :- (E dt>0 | DEF(x,t-dt)), USE(x,t)
  - Antidependency: reference then update
    ANTI(x,t) :- (E dt>0 | USE(x,t-dt)), DEF(x,t)

Iteration space

When a loop body is unrolled, if flow dependencies exist, they are called *loop-carried dependencies*. If a flow dependency exists, then it is called a *forward dependency.* If an antidependency exists, then it is called a *backward dependency.*

Protocols

Communication Protocols

Protocols for the reliable exchange of data between two nodes. Common errors include: lost, duplicate, reordered, and garbled messages.

Balanced Sliding-window Protocol

Designed for the exchange of messages between nodes with a direct connection. Messages may not be

- lost,
- duplicated, or
- reordered (fifo delivery)

but may be garbled.

Timer-based Protocol

Routing Algorithms

Destination-based routing

All-pairs Shortest-path

The Netchange algorithm

Routing with Compact Routing Tables

Hierarchical Routing

Deadlock-free Packet Switching

Last Modified

Send comments to aabyan@wwc.edu

Fundamental Algorithms

Wave and Traversal Algorithms

General Algorithms

- Ring algorithm
- Tree algorithm
- Echo algorithm
- Polling algorithm
- Phase algorithm
- Finn's algorithm

Traversal Algorithms

- Sequential polling
- Torus
- Hypercube
- Tarry

Depth-first Search Algorithms

- Classing
- Awerbuch
- Cidon
- Cidon with neighbor knowledge

Election Algorithms

Definition: An election algorithm is an algorithm that satisfies the following properities

1. Each process has the same local algorithm.
2. The algorithm is decentralized, i.e. compation can be initialized by an arbitrary non-empty subset of the processes.
3. The algorithm reaches a terminal configuration in each computation, and in each reachable terminal configuration there is exactly one process in the state *leader* and all other processes

are in the state *lost*.

Ring algorithms
Arbitrary networks

Termination Detection

Anonymous Networks

Snapshots

Synchrony in Networks

Last Modified
Send comments to aabyan@wwc.edu

Fault Tolerance

Robust Algorithms

Robust algorithms are designed to guarentee the continuous correct behaviour of correctly operating processes in spite of failures occuring in other processes during their execution.

Failure models

- *Initially-dead processes.* Does not execute a single step of its local algorithm.
- *Crash model.* Excecutes its local algorithm correctly up to some moment, and does not execute any step thereafter.
- *Byzantine behavior.* Executes steps that are arbitrary and, not in accordance with its local algorithm. In particular, a Byzantine process sends messages with an arbitrary content.

Stablizing Algorithms

A stabilizing algorithm can be started in any systems configuration, and eventually reaches an allowed state, and behaves according to its specification from then on.

Asynchronous Systems

Synchronous Systems

Stabilization

Last Modified
Send comments to aabyan@wwc.edu

link the resulting file *program1.pam* creating the executable program *myprogram*, and then execute the program *myprogram*

```
pcncomp -c program1.pcn
pcncomp program1.pam -o myprogram -mm program1 -mp main
myprogram
```

The option *-mm* indicates the module which contains the procedure which will be called to start the execution of the program. The option *-mp* is the name of the procedure. The default values are main in both cases. A file with execute privileges containing these commands may be used to compile, link, and execute the program. Such a file is called a *shell script*.

Getting Started with SR

The documentation is in the directory {\tt /home/castor/local/sr/doc}. The directory contains an overview (old but useful) and a report on SR (very useful). Both documents are available in PostScipt form and may be viewed using **pageview** or **gs**. Some examples of SR programs are found in the directory {\tt /home/castor/local/sr/examples}.

Compiling and Running SR programs

An SR program is contained in one or more *file*.**sr** source files. Globals and resources are stored in one or more source files that have an `.sr' suffix. They are compiled by invoking the compiler with the source file names. The compiler is *sr*. See the manual page on *sr* for more information.

Simple Programs

An SR program is contained in one or more *file.**sr*** source files. The compiler, *sr*, translates the SR source code into a C program, then calls the C compiler to create a **.o** file for each resource. Source files must be compiled in the correct order to make exports occur before imports.

Linking is directed by *srl*, which acts as a front end to the Unix linker, ld (1). The main resource in the program is specified at this time. One or more **.o** files of resources are combined with the SR and C runtime libraries to produce an executable program.

Compilation and linking takes place in the context of an Interfaces directory, which is used by the compiler and linker for storing and exchanging interface information.

A linked SR program that does not use virtual machines is completely self-contained, relying on only those files explicitly referenced by the SR program.

Virtual Machines

An SR program using virtual machines runs in cooperation with an execution-time manager, *srx*. *Srx* starts automatically when the SR program first creates an instance of the predefined {\tt vm} resource, or when it calls the predefined {\tt locate} operation.

The *srx* program is not contained within the SR program; it is loaded from a predetermined location (which can be overridden by the environment variable SRXPATH). If a new version of the SR system is installed, existing programs may need to be rebuilt to be compatible with the new *srx* An incompatible version of *srx* aborts with an error message.

Distributed Programs

A program is truly distributed if it runs on more than one physical computer. From SR's standpoint, any program using {\tt create vm() on n} is treated as a distributed program, even if {\tt n} does not

specify a different machine. The actual interpretation of {\tt n} is discussed later.

Srx uses the *rsh (1)* command to run remote portions of a distributed program. Only those networked hosts that you can access via *rsh* are usable by SR. Your login name must be the same on these hosts.

It is possible under some circumstances, and with some trickery, to run a distributed program over machines with dissimilar architectures. This is discussed further in Sec. 6 below. In general, though, SR programs should only be distributed over machines with compatible CPU types, such as Vaxes or Suns but not both. (Note: Sun-3 systems can run Sun-2 programs, but not the reverse.)

Physical Machine Numbers

Integers are used to specify the physical machines upon which new virtual machines are created. Initially, 0 specifies the machine upon which execution commenced, and other integers have default meanings depending on the local network configuration.

Every machine with a network usable by SR has a four-byte Internet-style address, usually given in a form something like {\tt 123.45.67.89}. If a physical machine number {\tt n} is between 1 and 255, it specifies the machine whose network address is the same as the current host but with {\tt n} replacing the last byte. To a host with address {\tt 123.45.67.89}, physical machine {\tt 47} is the machine whose address is {\tt 123.45.67.47}. Physical machine numbers above 255, represented in two or more bytes, replace the corresponding number of bytes of the original host's address.

The host numbers at your site can be obtained from your network administrator. Some machines have multiple addresses because they are on multiple networks; the first address returned by *gethostbyname*(3N) is used.

Default interpretations of physical machine numbers can be altered by calling {\tt locate}. This allows, indirectly, the specification of remote machines by name. The call

{\tt locate(n,hostame)}

associates the specified machine with the integer {\tt n}. The second argument, {\tt hostname}, is the symbolic name of some host machine; this is of course installation dependent. This association between {\tt n} and {\tt hostname} affects the subsequent meaning of machine {\tt n} on *all* virtual machines. In most cases it is advisable to set up explicit associations using {\tt locate} rather than depending on the default mappings.

Remote Execution

When *srx* initiates a new virtual machine using *rsh*, it must execute the SR program on the remote host. Specifying the program's location from the remote host's viewpoint is a difficult problem.

An automatic solution is available on systems that support remote disk access (e.g., NFS) with a systematic naming scheme. The SR installer configures an *srmap* file containing rules for locating and

naming files. The *srmap* file is read from a known location by *srx*; an alternate file can be substituted by defining the environment variable SRMAP.

The automatic scheme can be overridden by using a third parameter on a {\tt locate} call:

{\tt locate(n,hostame,pathname)}

sets {\tt pathname} as the file to be executed by *rsh* when a virtual machine is created on host {\tt n}.

On systems without remote disks, some sort of manual action is usually needed to copy the executable SR program to remote machines. *Rcp*(1) or *rdist*(1) can be used for this. The remote location will depend on the *srmap* file; typically this would be the same location relative to the login directory on both machines, e.g., \verb+~mike/test/a.out+ on both machines. Be sure to recopy the file each time it is rebuilt; mixing old and new versions can lead to disaster. If the automatically generated filename is unsuitable, again an explicit path in a {\tt locate} call can be used to override it.

Heterogeneous Execution

Distributed SR programs are intended to execute in a homogeneous environment. However, under certain circumstances, dissimilar but related systems can be used. It is usually necessary to compile the identical programs separately under all the different environments and to arrange (calling {\tt locate} if necessary) to execute the correct versions. We offer some guidelines here, but experimentation may also be required.

Programs built on Sun-2 systems can distributed to Sun-3 systems without recompilation; however, the reverse is not true. Separately built Sun-2 and Sun-3 programs can also be used.

SR programs have been successfully distributed between Sun-3 and Hewlett-Packard 200 systems, which have similar architectures. The identical program was compiled separately on both systems.

SR programs have also been successfully run on Vax machines running a mixture a 4.3 BSD and Ultrix systems. Again, the identical program was first compiled twice under the two different environments.

Additional SR Tools

Although only *sr* and *srl* are needed to run SR programs, other tools assist with related tasks. *srm* creates a *make*(1) description file for building complex SR programs. *srtex* and *srgrind* format SR programs for typesetting.

CPTR 464 Compiler Design - 4 cr. hr.

Description

Study of the techniques for translating conventional programming language source into executable machine codes. Topics include: lexical analysis, syntactic analysis and parsing, static and runtime storage management, and code generation. Prerequisite: CPTR 143.

This is a project oriented course. You can expect to put in 9-12 hours per week for the class (including lectures) and an additional 3-4 hours per week for the lab/project.

Goals

Upon completion of the course you will

- understand syntax directed programs,
- be able to construct a lexical analyzer(scanner) using regular expressions and a scanner generator tool,
- be able to construct a parser from a context-free grammar and a parser generator tool,
- be able to construct and use a symbol table to support the parsing of context sensitive constructs, and
- be able to generate machine code equivalents of the basic data types and control structures as the output of a parser.
- be able to use **`make`** and makefiles in project development

Evaluation

The course grade is determined by the quantity and quality of work completed on homework assignments, the project, and the tests. The grade expectations document helps to explain the different grades.

WEIGHT %		& GRADES
Project	90%	90 - 100% As
Homework	10%	80 - 89% Bs
Paper/report	0%	70 - 79% Cs
Test	0%	60 - 69% Ds

Resources

Lecture notes and schedule
The project
Compiler Construction Script

Textbook:

Watt, David and Brown, Deryck *Programming Language Processors in Java* Prentice Hall 2000 (ISBN 0-130-25786-9)

Reading List:

- Appel, Andrew W. *Modern Compiler Construction in Java* Cambridge University Press 1998 (ISBN 0-521-58388-8)
- Fraser and Hanson *A Retargetable C Compiler: Design and Implementation* Benjamin Cummings 1995
- Aho, Sethi, Ullman *Compilers: Principles, Techniques and Tools* -- (encyclopedic, the `dragon book')
- Holub *Compiler Construction in C* -- (BU, tools, C)
- Fischer & LeBlanc *Crafting a Compiler* -- (TD, BU, tools, Ada)
- Waite & Goos *Compiler Construction* -- (TD,BU,Pascal)
- Proebsting, T. A., 1995. BURS Automata Generation. *ACM TOPLAS 17*, 3 461-186.
- Bacon et.al. 1994. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys 26*, 4
- Aaby, A. *Compiler Design with Flex and Bison*
- Holmes, Jim *Object-Oriented Compiler Construction* Prentice-Hall 1995
- Holmes, Jim *Building Your Own Compiler with C++* Prentice-Hall 1995

Tools:

- Cool: portable project for compiler construction
- Lex(Flex), YACC(Bison)
- Eli Compiler Construction System
- PCCTS(Purdue Compiler-Construction Tool Set)
- Prolog Parser Tools (Aaby)
- JavaCC - Java Compiler Compiler

WWW:

http://www.idiom.com/free-compilers

Usenet News Groups:

comp.compilers, comp.compilers.tools.pccts

Technical Journals:

JACM, TOPLAS, SigPlan

Outdated stuff

labs

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Compiler Design Project

The compiler design and implementation project is expected to require 80-90 hours of out of class activity and is likely to be the largest project you have attempted as an undergraduate. The project should be done in teams of at least two. You may want to experiment with the *Extreme Programming* method and do pairs programming. A more ambitious project than detailed here will require a larger team effort and strong software engineering skills and may involve the use of software tools to assist in the construction of the compiler. Such tools are necessary for table driven methods and are optional for the method of recursive descent.

Students who would like to work on a different problem may propose an alternative project at any point in the quarter. The project must be well-defined, approved by the instructor, and involve roughly the same amount of work as the remaining assignments.

Your project should be one of the following:

1. Given a partial implementation, complete the implementation. Project grade will be based on the fully impletemented feature set beyond those fully implemented features in the partial implementation.
    - ***Recommended for Spring 2001:*** Extend the mini-Triangle compiler implementation from the textbook Watt & Brown, *Programming Language Processors in Java* Prentice-Hall 2000.
    - Implement the Tiger programming language from Appel *Modern Compiler Construction in Java* Cambridge University Press1998
2. Design and implement a programming language or a subset of an existing language.
3. Special projects. These should not be attempted unless there is a reasonable chance of success.
    - Compiler construction tools.
    - Heap manager.
    - Register allocation module.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

CPTR 464 Compiler Design 4

Schedule (subject to change without notice)

Lecture Notes	Text	Assignments	Due

Introduction
- Script
 	
Old - Introduction to Compilers
	1
	App B
		1. Read Chapter 1
		2. Insure that you have available a Java environment
		
		• From java.sun.com, obtain and install
		  ○ the Java 2 SDK
		  ○ Forte for Java (IDE)
		  ○ Java 2 Runtime Environment
		
		• Insure that you have available the Triangle compiler and interpreter
		
		• From *www.dcs.gla.ac.uk/~daw/books/PLPJ* obtain and install
		  ○ the Triangle compiler and TAM interpreter
		
		• Insure that the Java environment, the Triangle compiler, and TAM interpreter work and you can use them by
		
		• create a small Triangle program, compile and execute it.
		• Write up a HOWTO installation and user's guide for the Triangle environment.
		
		• Optional: obtain and install
		
		• JLex
		• CUP
		• SPIM (optional)
		
		from www.cs.princeton.edu/~appel/modern/java	

Language Processors	2	Read chapter 2; no lectures	
Compilation	3 App D		
Syntactic Analysis Grammars Syntactical specification Top Down Parsing LL(1) Grammar RD Parser Script Bottom Up Parsing notes on:	4	Construct a compiler front end (parser, scanner, & generate an AST) for a subset of Simple Context-free grammar for Simple *program* ::= LET *definitions* IN *command_sequence* END *definitions* ::= **e** | INTEGER *id_seq* IDENTIFIER . *id_seq* ::= **e** | *id_seq* IDENTIFIER , *command_sequence* ::= **e** | *command_sequence command* ; *command* := SKIP | READ IDENTIFIER | WRITE *exp* | IDENTIFIER := *exp* | IF *exp* THEN *command_sequence* ELSE *command_sequence* FI | WHILE *bool_exp* DO *command_sequence* END	End of 3rd week

CUP yacc (bison); Yacc/Bison Lexical Analysis Scanner script JLex lex - (flex) Lex/Flex with yacc/bison Abstract Syntax		`exp ::= exp + term	exp - term	term` `term :: term * factor	term / factor	factor` `factor ::= factor^primary	primary` `primary ::= NUMBER	IDENT	(exp)` `bool_exp ::= exp = exp	exp < exp	exp > exp`	
Contextual Analysis Contextual Analysis Symbol Tables	5	By this point, you should have finished adding the syntactical extensions to your project (modifications to both parser and scanner).	End of 5th week
Run-time Organization Run-Time Organization	6 App C	By this point, you should have finished making the necessary changes to the contextual analyzer.	End of 6th week
Code Generation Instruction Selection Stack Machine Code Generation (1.5)	7 App C		
Interpretation	8	By this point you should have finished making the necessary changes to the code generator.	End of 8th week
Optimization	9		
Final Exam	11	Project presentation and/or written exam postscript html	

Example

A complete compiler using flex and bison

Not covered:

1. RE to NFA
2. NFA to DFA conversion
3. Bottom up parser generation

Previous Projects
Object-Oriented Methods
Analysis & Design

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Compiler Design Paper & Report

Write a report (term paper ~10 pages) and make a presentation in class on one of the following topics:

- None for Spring 2001

Last Modified - . Comments and content invited aabyan@wwc.edu

CPTR 464 Compiler Design

Comprehensive Final Exam

Name:

Problem	Max. Score	Score
1	20	
2	20	
3	20	
4	20	
5	20	
6	20	
7	20	
8	20	
9	20	
Total	180	

Self evaluation:

Letter grade you feel you deserve for this test:

Letter grade you feel you deserve for the course:

Instructions: Comprehensive and complete answers are not expected however, your grade depends on whether your answer makes it clear that you understand the concept and are capable of producing a high quality implementation. Each answer is expected to be confined to a single sheet of paper.

1. (20 points) Describe the phases of a compiler.
2. (20 points) Compare and contrast the structure of single pass and multipass compilers.
3. (20 points) Explain what regular expessions are and how to use them to describe the tokens of a programming language.
4. (20 points) Explain and illustrate how to implement a hand written scanner from regular expression descriptions of tokens.
5. (20 points) Explain what a context-free grammar is and how to use one to describe the syntax of a programming language.

6. (20 points) Explain and illustrate how to construct a recursive descent parser from a context-free grammar.
7. (20 points) Explain contextual analysis and illustrate how it is done. Include a disscussion on symbol tables.
8. (20 points) Describe the required run time support for nested blocks and recursion.
9. (20 points) Describe and illustrate the key code generation issues for a stack machine.

Language design and implementation

The Programming Language Life Cycle

Purpose		**To guide a team though developing a programming language and compiler.**
Entry Criteria		• A programming language needs statement • Materials, facilities, and resources for team support. • A development team
General		The following phases are not sequential but proceed in parallel and are interative with feed back from one phase to another.
Phases	**Acitivities**	**Description**
1	Design	Produce a language design to meet the need/requirement.
2	Specification	The overriding criterion for a language's syntax is that programs should be *readable* and should facilitate semantic understanding of the program. Therefore, the syntactic forms and the semantic concepts should be (more or less) in one-to-one correspondence. • Formalize the design with a formal or informal specification (syntax and semantics) to facilitate communication of the design to other people. • Use the BNF, EBNF or syntax diagrams • To encourage sematic simplicity and regularity produce a formal semantic specification.
3	Implementation	1. Produce a prototype implementation (interpreter, an interpretive compiler, ...) to assist in refining the design and specification. 2. Produce an industrial-strength compiler when the language design has stabilized with compile and run time error reporting and recovery and optimization features.
4	Manuals and training guides	• Language specification • Programmer's guide • Tutorial
Exit criteria		• Industrial-strength compiler • Manuals and Training guides

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Compiler Construction using Flex and Bison

Anthony Aaby

Contents

Book in .dvi format
Book in .ps.gz format

Parser Tools in Prolog

This document describes some tools for processing context-free grammars into LL(1) form. The following tools are available

Context-Free Grammar

The specification of the contex-free grammar for a language consists of four items, the specification of the terminal symbols of the language, the specification of the nonterminals, the productions or derivation rules, and the start symbol of the grammar. As an example of a context-free grammar, here is a specification of the context-free grammar for arithmetic expression.

```
terminal('+').
terminal('*').
terminal('(').
terminal(')').
terminal(id).

nonterminal(e).   % expression
nonterminal(t).   % term
nonterminal(f).   % factor

start(e).

p(e,[t]).
p(e,[e,'+',t]).
p(t,[f]).
p(t,[t,'*',f]).
p(f,[id]).
p(f,['(',e,')']).
```

Removal of Left-Recursion

Note that the second production for expression in the grammar of the previous section is left recursive. The use of such a production in the design of a recursive descent parser would result in an infinite loop. Sometimes left recursion may be eliminated.

```
terminal('+').
terminal('*').
terminal(id).
terminal('(').
terminal(')').

nonterminal(e).
nonterminal(e0).
nonterminal(t).
```

```
nonterminal(t0).
nonterminal(f).

start(e).

p(e,[t,e0]).
p(e0,['+',t,e0]).
p(e0,[epsilon]).
p(t,[f,t0]).
p(t0,['*',f,t0]).
p(t0,[epsilon]) .
p(f,['(',e,')']).
p(f,[id]).
```

Here is a Prolog program which removes left recursion if possible.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                        Removal of Left Recursion
%                          Waite and Goos: p. 126
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- dynamic p/2 .
:- dynamic nonterminal/1 .

remove_left_recursion :- setof(N,nonterminal(N),Ns),
                         rm_left_rec([],Ns).

rm_left_rec(Xjs,[]).
rm_left_rec(Xjs,[Xi|R]) :- step2(Xjs,Xi),
                           step3(Xi,Xip,Xjs),
                           append(R,Xip,Rs),
                           append(Xjs,[Xi],NXjs),
                           rm_left_rec(NXjs,Rs).

step2([],Xi).
step2([Xj|Xs],Xi) :- e_bagof(W,p(Xi,[Xj|W]),Ws), Ws \= [], Ws \= [[]],
                     e_bagof(XXj,p(Xj,XXj),XXs),
                     retractall(p(Xi,[Xj|Wx])),
                     replace_each0(Xi,Xj,Ws,XXs),
                     step2(Xs,Xi).
step2([Xj|Xs],Xi) :- step2(Xs,Xi).

replace_each0(Xi,Xj,[],_).
replace_each0(Xi,Xj,[W|Ws],XXs) :- replace_each00(Xi,Xj,W,XXs),
                                   replace_each0(Xi,Xj,Ws,XXs).

replace_each00(Xi,Xj,[],XXs).
replace_each00(Xi,Xj,W,[]).
replace_each00(Xi,Xj,W,[XXj|XXs]) :- append(XXj,W,XXjW),
                                     assert(p(Xi,XXjW)),
                                     replace_each00(Xi,Xj,W,XXs).

step3(Xi,[Xip],Xjs) :- e_bagof(W,p(Xi,[Xi|W]),Ws), Ws \= [], Ws \= [[]],
                       e_bagof([K|X],(p(Xi,[K|X]),K\=Xi),Xs),
                       retractall(p(Xi,Rhs)),
                       newN(Xi,Bi),
                       assert(nonterminal(Bi)),
                       assert(p(Bi,[epsilon])),
```

```
                        replace_each1(Xi,Bi,Ws,Xjs),
                        replace_each2(Xi,Bi,Xs).
step3(Xi,[],Done).

newN(Xi,Xip) :- atomtolist(Xi,L), int(N),
                atomtolist(N,Ln), append(L,Ln,Lp), atomtolist(Xip,Lp),
                \+ nonterminal(Xip).

int(0).
int(N) :- int(M), N is M+1.

replace_each1(Xi,Bi,[],Xjs).
replace_each1(Xi,Bi,[W|Ws],Xjs) :- append(W,[Bi],WBi),
                                   assert(p(Bi,WBi)),
                                   replace_each1(Xi,Bi,Ws,Xjs).
replace_each2(Xi,Bi,[]).
replace_each2(Xi,Bi,[XX|XXs]) :- append(XX,[Bi],XXBi),
                                 assert(p(Xi,XXBi)),
                                 replace_each2(Xi,Bi,XXs).
```

Left-Factoring

Another common problem is when two productions for the same nonterminal share a common prefix on the right-hand side of the productions. The common prefix makes it impossible to choose (with a fixed amount of look-ahead) the proper production in top-down parsing. This elimination of the common prefix is called left-factoring. Here is a grammar for the **if-then-else** construct.

```
terminal(a).
terminal(b).
terminal(e).
terminal(i).
terminal(t).

nonterminal(ss).
nonterminal(cc).

start(ss).

p(ss,[i,cc,t,ss,e,ss]).
p(ss,[i,cc,t,ss]).
p(ss,[a]).
p(cc,[b]).
```

Left factoring the grammar produces the following result.

```
terminal(a).
terminal(b).
terminal(e).
terminal(i).
terminal(t).

nonterminal(ss).
nonterminal(cc).
nonterminal(ss0).

start(ss).
```

```
p(ss,[i,cc,t,ss,ss0]).
p(ss,[a]).
p(ss0,[e,ss]).
p(ss0,[epsilon]).
p(cc,[b]).
```

The following code left-factors the given grammar.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                         Left Factoring
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

do_left_factoring :- setof(N,nonterminal(N),Ns), do_lf(Ns).

do_lf([]).
do_lf([A|Ns]) :- p(A,R1), p(A,R2), R1 \= R2,
                 maxcommon(R1,R2,C,R1p,R2p), C \= [],
                 newN(A,Ap),
                 assert(nonterminal(Ap)),
                 restofcommon(A,C,Ap),
                 append(C,[Ap],Rhs),
                 assert(p(A,Rhs)),
                 do_lf([A|Ns]).
do_lf([A|Ns]) :- do_lf(Ns).

restofcommon(A,C,Ap) :- p(A,Rhs),
                        append(C,R,Rhs),
                        retract(p(A,Rhs)),
                        \+ p(Ap,R),
                        ((R =[], \+ p(Ap,[epsilon]), assert(p(Ap,[epsilon])));
                         (R\=[], \+ p(Ap,R),         assert(p(Ap,R)))),
                        restofcommon(A,C,Ap).
restofcommon(A,C,Ap).

maxcommon([X|R1],[X|R2],[X|C],R1p,R2p) :- maxcommon(R1,R2,C,R1p,R2p).
maxcommon(R1,R2,[],R1,R2).
```

First and Follow Sets

Even if left-recursion can be eliminated and the productions have been left-factored, we may not a have a grammar which is suitable for top-down parsing with one symbol of look-ahead. For example, in the following grammar it is not possible to choose between the productions for the nonterminal `aa` because `x` is also derivable from the nonterminal `bb`.

```
terminal(x).
terminal(z).

nonterminal(aa).
nonterminal(bb).
nonterminal(cc).

start(aa).

p(aa,[bb,z]).
p(aa,[x,z]).
p(bb,[cc]).
p(bb,[cc,bb]).
p(cc,[x]).
```

The set of initial terminals derivable from the right-hand side **rhs** of a production is called **first(rhs)**. For this grammar, the first sets are:

```
first([bb,z],[x])
first([x,z],[x])
first([cc],[x])
first([cc,bb],[x])
first([x],[x])
```

The set of first sets are computed by the following program.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                First Sets
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  First(A) = Fs where
%                        x       in Fs iff A =>* xB
%                        epsilon in Fs iff A =>* epsilon
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


:- dynamic first/2 .

firstsets :- setof(Rhs,N^p(N,Rhs),Rhss), mk_firsts(Rhss).

mk_firsts([]).
mk_firsts([Rhs|Rhss]) :- first_set(Rhs,Fs), assert(first(Rhs,Fs)),
                         mk_firsts(Rhss).

first_set(L,Fs) :- e_setof(X,a_first(L,[],X),Fs).

a_first([],V,epsilon).
a_first([epsilon],V,epsilon) :- !.
a_first([epsilon|L],V,X) :- a_first(L,V,X).
a_first([X|L],V,X) :- terminal(X).
a_first([N|L],V,X) :- nonterminal(N),
                      \+ in(N,V),
                      p(N,LN),
                      append(LN,L,NL),
                      a_first(NL,[N|V],X).
```

When the production `p(x,[epsilon])` is included in the set of productions, selection must also be based on the set of terminals which can follow a given non-terminal. We can summarize the previous discussion as follows. A grammar is LL(1) (suitable for top-down parsing with one symbol of look ahead) iff it satisfies the following two rules.

1. The sets of initial symbols of all sentences that can be generated from the right-hand sides of a given non-terminal must be disjoint.
2. The set of initial symbols of each non-terminal which generates the empty string must be disjoint from the set of symbols which can follow it.

The code which constructs the set of initial terminals which can follow a non-terminal follows.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                               Follow Sets
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Follow(A) = Fs where
%                        x       in Fs iff S =>* aAxb
```

```
%                        epsilon in Fs iff S =>* aA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- dynamic follow/2 .

followsets :- setof(N,(nonterminal(N),start(S),N \= S),Ns),
              mk_mt_follows(Ns,MT),
              mk_follows([(S,[eof])|MT],FS),
              assertFS(FS).

mk_mt_follows([],[]).
mk_mt_follows([N|Ns],[(N,[])|MT]) :- mk_mt_follows(Ns,MT).


mk_follows(CFS,AFS) :- p(A,L), append(Alpha,[B|Beta],L), nonterminal(B),
                       Beta \= [],
                       first_set(Beta,FirstBeta),
                       delete(epsilon,FirstBeta,FB),
                       append(Fp,[(B,FSB)|RFs],CFS),
                       union(FB,FSB,NFSB), \+ samesets(FSB,NFSB),
                       append(Fp,[(B,NFSB)|RFs],NCFS),
                       mk_follows(NCFS,AFS).

mk_follows(CFS,AFS) :- p(A,L), append(Alpha,[B],L), nonterminal(B),
                       append(Fp,[(B,FSB)|RFs],CFS),
                       in((A,FSA),CFS),
                       union(FSA,FSB,NFSB), \+ samesets(FSB,NFSB),
                       append(Fp,[(B,NFSB)|RFs],NCFS),
                       mk_follows(NCFS,AFS).

mk_follows(CFS,AFS) :- p(A,L), append(Alpha,[B|Beta],L), nonterminal(B),
                       Beta \= [],
                       first_set(Beta,FirstBeta), in(epsilon,FirstBeta),
                       append(Fp,[(B,FSB)|RFs],CFS),
                       in((A,FSA),CFS),
                       union(FSA,FSB,NFSB), \+ samesets(FSB,NFSB),
                       append(Fp,[(B,NFSB)|RFs],NCFS),
                       mk_follows(NCFS,AFS).
mk_follows(FS,FS).

assertFS([]).
assertFS([(N,Fs)|FS]) :- assert(follow(N,Fs)), assertFS(FS).
```

Parsing Table and an LL(1) Parser

The set of first and follow sets may be used to construct a table which can be used to guide the parser of an language which has an LL(1) grammar. The table is indexed by a non-terminal and the current input symbol. The following code constructs such a table.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   Construction of Parsing Tables
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

do :- firstsets, followsets, make_LL1_table.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   Construction of Predictive Parsing Table
%                      Aho & Ullman: Algorithm 5.4 p.190
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


:- dynamic table/3 .

make_LL1_table :- setof((N,Rhs),p(N,Rhs),Productions),
                  make_entries(Productions).

make_entries([(N,Rhs)|Productions]) :- make_an_entry((N,Rhs)),
                                       make_entries(Productions).

make_an_entry((N,Rhs)) :- first(Rhs,First),
                          each_entry((N,Rhs),First).

each_entry((N,Rhs),[]).
each_entry((N,Rhs),[eof]).

% 2.
each_entry((N,Rhs),First) :- in(T,First), terminal(T),
                             delete(T,First,Rfirst),
                             table(N,T,(N,Rhs)),!,
                             each_entry((N,Rhs),Rfirst).

each_entry((N,Rhs),First) :- in(T,First), terminal(T),
                             delete(T,First,Rfirst),
                             assert(table(N,T,(N,Rhs))),
                             each_entry((N,Rhs),Rfirst).
% 3.
each_entry((N,Rhs),First) :- in(epsilon,First),
                             follow(A,Follow),
                             each_terminal((N,Rhs),Follow),
                             eof_part((N,Rhs),Follow),
                             table(N,T,(N,Rhs)),
                             delete(epsilon,First,Rfirst),
                             each_entry((N,Rhs),Rfirst).


eof_part((N,Rhs),Follow) :- in(eof,Follow),        table(N,eof,(N,Rhs)),!.
eof_part((N,Rhs),Follow) :- in(eof,Follow), assert(table(N,eof,(N,Rhs))).
eof_part((N,Rhs),Follow).

each_terminal((N,Rhs),[]).
each_terminal((N,Rhs),[T|Follow]) :- table(N,T,(N,Rhs)),!,
                                     each_terminal((N,Rhs),Follow).
each_terminal((N,Rhs),[T|Follow]) :- assert(table(N,T,(N,Rhs))),
                                     each_terminal((N,Rhs),Follow).
each_terminal((N,Rhs),[epsilon|Follow]) :- each_terminal((N,Rhs),Follow).
each_terminal((N,Rhs),[    eof|Follow]) :- each_terminal((N,Rhs),Follow).
```

A parser uses the table generated by the previous program as follows. Initially the start symbol of the grammar is placed on a stack. Then the next two rules are followed until the input is empty or a situation arises for which there is no entry in the table.

1. If the current input symbol and the stack top symbol are the same, then pop the stack and consume the input symbol.
2. If the stack top symbol is a non-terminal then consult the table, pop the stack and push the table entry onto the stack.

Here is the code for an LL(1) parser.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%                               LL(1) Parser Program
%                             Aho & Ullman: p.184,186
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ll_1_parser :- start(S), input(Input), ll_1_parser([S,eof],Input).

ll_1_parser([],[]).
ll_1_parser([T|Stack],[T|Input]) :- ll_1_parser(Stack,Input).
ll_1_parser([N|Stack],[X|Input]) :- table(N,X,(N,Rhs)),
                                    append(Rhs,Stack,NStack),
                                    ll_1_parser(NStack,[X|Input]).

ll_1_parser(Stack,Input) :- error_recovery_routine(Stack,Input).
```

Here are some miscellaneous predicates required by the previous code.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          Miscellaneous Predicates
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

append([],L,L).
append([X|A],B,[X|L]) :- append(A,B,L).

in(X,[X|L]).
in(X,[Y|L]) :- in(X,L).

union([],B,B).
union(A,[],A).
union([X|A],B,C) :- in(X,B), union(A,B,C).
union([X|A],B,[X|C]) :- \+ in(X,B), union(A,B,C).

subset([],B).
subset([X|A],B) :- in(X,B), subset(A,B).

samesets(A,B) :- subset(A,B), subset(B,A).

delete(X,[],[]).
delete(X,[X|R],R).
delete(X,[Y|R],[Y|L]) :- X \= Y, delete(X,R,L).


e_bagof(A,B,C) :- bagof(A,B,C),!.
e_bagof(A,B,[]).

e_setof(A,B,C) :- setof(A,B,C),!.
e_setof(A,B,[]).

sort(List,Sorted) :- sort(List,[],Sorted).
sort([],Sorted,Sorted).
sort([X|List],PSort,Sorted) :- insert(X,PSort,Psort), sort(List,Psort,Sorted).

insert(X,[],[X]).
insert(X,[Y|List],[X,Y|List]) :- X @=< Y.
insert(X,[Y|List],[Y|ListP]) :- X @> Y, insert(X,List,ListP).
```

Compiler Design: Labs

Preliminaries

Each student must maintain a Lab Journal. The journal is a written record of the activities and changes made to the original Minipas compiler which led to the completed project. The journal entries must be dated and summarize

- all ideas,
- all design decisions,
- all code modifications, and
- all problems encountered and the solutions found.

The journal should be complete enough to allow someone else to reproduce the sequence of activities which led to the completed project.

The Labs

1. Introduction to the Minipas Compiler
    - ftp /pub/compiler.tar from gboro.rowan.edu
    - tar xvf compiler.tar
    - change cc to gcc in mk and makefile files
    - change mk to execute rights
    - run mk
    - test resulting system
2. Lexical Analysis: Lex & Flex
    - ~aabyan/Pub/Lex
    - Construct a lex program to count characters, words and lines
    - Construct a lex program to convert lower case to upper case
    - Construct a lex program to convert English to Morse Code
    - Construct a lex program to print out the following token class numbers for Pascal source input.
        1. Identifier
        2. Numeric constant
        3. :=
        4. Comparison Operator
        5. Arithmetic Operator
        6. String Constant
        7. Keyword

        8. Comments
    - Extend the Minipas compiler to recognize ...
3. Syntax Analysis: YACC & Bison
    - ~aabyan/Pub/Yacc
    - Construct a Yacc program to implement a post-fix notation calculator
    - Construct a Yacc program to implement an infix notation calculator
    - Construct a Yacc program to implement a multifunction algebraic notation calculator
    - Construct a Yacc program to generate stack machine code for arithmetic expressions.
    - Extend the Minipas compiler to recognize ...
4. Contextual Analysis: The Symbol Table
    - Attributes: Stack information
    - Context requirements
    - Symbol Table ADT: Functions, contents (name and attibutes)
    - Symbol Table Implementation: list, tree, hash table; Blocks & Scope
    - Translation grammars
5. Run-time Support
    - Monolithic Program: Data Segment, Code Segment
    - Expressions: expression stack
    - Subroutines: non-recursive, recursive
    - Nested environments
    - Symbol table extensions
6. Intermediate Code
    - Abstract Syntax Trees
    - Quads (three address code -- op, arg1, arg2, result)
7. The Interpreter
    - Accumulator machine
    - Stack machine
    - Register machine
8. Code Generation

CPTR496, 497, & 498 Seminar

Description

Presentation and discussion of current topics of interest with computer science. Each student is required to conduct an approved design project from conception to final oral and written reports. Prerequisite: Senior standing in computer science.

Each class session each you will be expected to report on your progress and plans for the next week.

Over the course of the year, the project will consume at least 120 hours.

Evaluation

The course grade is determined by the quantity and quality of work completed on the project.

The oral report on the project must be presented to the computer science faculty and students in the spring quarter. The written report must follow the style of articles in professional journals but must also include a title page and a table of contents. The use of LaTeX is encouraged.

Course Goal

Upon completion of this course you will have completed a project that is the capstone of your academic work. It showcases your strengths, skills and interests in Computer Science.

Resources

Ian Parberry

How to present a paper in theoretical computer science *SIGACT News***19**, 2 (1988), pp. 43-47 Available online.

McGeoch & Moret

How to present a paper on experimental work with algorithms *SIGACT News* **30**, 4 (1999) pp. 85-90

Basse, Sara

A Gift of Fire: Social, Legal, and Ethical Issues in Computing Prentice Hall 1997

Oz, Effy, **Ethics for the Information Age** B&E Tech 1994.
Huff & Finholt **Social Issues in Computing** McGraw-Hill 1994.
Perrolle, Judith A., **Computers and Social Change** Wadsworth 1987.

Usenet

depends on project domain but the following are recommeded for software engineering issues: comp.software-eng, comp.software.licensing, comp.software.testing, comp.specification, comp.specification.z .

Previous Projects

Senior Projects

The senior project is to be the capstone of your academic work here at WWC. It is intended to showcase your strengths, skills and interests in computer science to the Computer Science faculty, prospective employers and/or graduate schools.

Projects may range from concerns central to computer science, computer engineering, and computer information science to computer applications in other domains. Suitable projects may involve the design and implementation of software or hardware or may be theoretical or experimental in nature. The following pages describe some possible projects.

Software projects are expected to conform to good software engineering practices and must be complete with a specification document, a design document, well documented code and a user's manual.

Theoretical projects are expected to be summarized in a technical report. It should conform in style to typical published papers in Computer Science.

Assignments and grade sheet

Topics

Algorithms and data structures

- Develop an algorithm to ...
- Find a better algorithm to ...
- Develop a parallel algorithm for...
- ...

Architecture

- Construct a simulator for an alternative architectures (Aaby)
  - P-machine
  - SECD-machine
  - Lambda machine
  - Logic Machine
- VLSI -- implement a lambda calculus machine (Aaby, Aamodt)
- Construct a universal assembly language and assembler. (Aamodt, Aaby)
- ...

Artificial Intelligence

- An expert system for ... (Aaby)
- A neural net to ... (Aamodt)
- A natural language ... (Aaby,Klein)
- An automated reasoning ... (Aaby)
- A game ...

Database

- Do a performance analysis for Oracle on NT and Linux

Human Computer Interaction

- GUI
- Use XML to describe a windowing environment.
- Speech Synthesis
- Speech Recognition
- Handicapped Access
- Develop an editor ...
- Develop a user interface ...
- Utilize computer graphics to ...
- Perform image processing to ...
- Produce a computer animation of
- ...WEB...
- ...

Numerical and Symbolic Computation

Operating Systems

- Design and implement an OS or portions an OS.
- Add/Modify features of an OS or Network
- Do something with Unix(NetBSD, Linux, etc.), Mach, Ameoba, OS2, Windows NT
- Collect, modify, develop tools for monitoring, analyzing and/or simulating a network.
- Develop sys admin materials for the NT environment
- Develop sys admin materials for networking
- ...

Programming Languages

- Design a multiparadigm programming language

- Design and implement a programming language
- Compare language based memory managers (including garbage collectors) to OS memory managers.
- Compilers etc.
    - Construct a universal assembly language and assembler. (Aamodt, Aaby)
    - Use ELI to construct a compiler
    - Construct a compiler for ???
    - Develop a hardware (VLSI) lambda calculus interpreter
    - Complete the development of Prolog based compiler writing tools.
    - Port Aaby's Prolog based compiler example to PCN.
    - Construct/assemble supporting routines for a compiler.
    - Translate Lucent Technologies Limbo to Java
    - Construct a compiler to translator SPECS to C++ (Wether & Conway (1996) "A Modest Proposal: C++ Resyntaxed" *ACM SIGPLAN 31:11 Nov 1995 p 74.)*
- Runtime Environment
    - SECD-machine (Lispkit)
    - Lambda machine: a lambda calculus interpreter (parallel)
    - Prolog machine (Prologkit)

Software Engineering

- Specify, design, implement and formally verify the correctness of ...
- ...

Social and Professional Issues

Miscellaneous

Projects may span more than one area of Computer Science. For example, a project could involve the development of a distributed algorithm thus combining algorithms, operating systems and software engineering.

- Construct a program to model ...
- Configure a Linux workstation for scientific work (Chemistry, Engineering)
- Construct a project management systems
    - web based
    - enter project assignments
    - review progress
    - adjust assignments
- Construct a simulation of a network
- Construct a program to simulate ...
- Develop a web based grade book which provides restricted access for students and full access for the instructor & grader. Use Java. Provide interface to administrative database to facilitate access to class lists and grade submission. Hint modify Ken Wiggins' grade program.

- Develop an electronic form to submit contract teacher information should include authentication.
- Develop an automated environment for managing the routine work of the WWC Curriculum Committee. The environment should provide for
    - Electronic form submission -- must provide source authentication
    - Electronic signature submission -- originating department and other required signatures.
    - Public browsing of submissions.
    - Record of committee action
    - Generate the committee Minutes
    - Generate a report to Faculty Senate
    - Allow for additional agenda items

Last update:

Send comments to: webmaster@cs.wwc.edu

Senior Projects

2000-2001

Graham, Todd
Halvorsen, Chad
Mueller, Brett
Rodriguez, Sam
Shrock, Court
Van Dolson, Ray
Woehler, Aaron

1999-2000

Bowman, Cliff
Wesslen, Todd

1998-1999

Beeson, Eric
Buchheim, Hans

1997-1998

Fortiner, Samuel
Hanson, Eric
Reinhardt, Martin
Vliet, John

1996-1997

Driesen, Erwin
    Parallel Programming with MPI
Francis, Karl

Power PC 604 Simulator and Assembler

1995-1996

Downs, Warren.

1. Survey of Minimal OSs

Engelman, John.

1. Survey of Network Analysis Tools
2. Network Analysis Tool Design Proposal
3. Network Sniffer Implementation

Foster, Mark.

1. Multimedia Applications on the WWW
    - Example HTLM document: Maintaining a Todo and Done lists
    - A syllabus for WEB based Multimedia for secondary teachers.
    - Design and implementation of a web server.

McNeil, James.

1. Computer Assisted Learning
    - Graphical user interface for rote memorization of arithmetic tables
2. Project Design Proposal

Russell, Timothy.

1. Computer Based Language Instruction Tools

1994-1995

Shannon Dobbins, Paul Ford, & Roger Santo

Electronic Implementation of Curriculum Committee Forms

Last update:

INFO 150 Software Application 1

Bulletin Description

Study of application software from a user perspective. Topics vary and may be repeated for credit when topics vary. Prerequisites vary depending on the software package and level. *Does not apply toward a major or minor in Computer Science.*

Additional Information

These are practical (non-theoretical) courses dealing with application software packages requiring about 30 clock hours of work. Some of the course materials are interactive workbooks meaning that the student is expected to have an appropriate computing environment available or in some cases, a web based interface may be available. These courses are appropriate for highly motivated students, who are resourceful, capable of independent work, and are seeking to learn a specific skill. Students are expected to obtain their own textbooks.

Evaluation

Course grades may be determined by an exam, term paper, or term project and may include computer demonstrations and/or presentation by the student. The final grade is subjective. Distance students may be required to make arrangements through a proctor who must submit a notarized statement of identity. Many of the courses use the work report form to keep track of your work. For a topic in programming languages use the programming language work report form.

Topics

Excel for scientists and engineers MathCad Matlab Maple

Suggested textbooks **- Textbook publishers**

- www.coriolis.com
- www.course.com
- www.ddcpub.com

- www.fortuitous.com
- www.idgbooks.com
- www.manning.com
- www.mcp.com
- www.oreilly.com
- www.osborne.com
- www.phptr.com
- www.wrox.com

Last Modified - . Send comments to aabyan@wwc.edu

INFO 250 System Software 1

Bulletin Description

Study of system software from a user and/or administrative perspective. Topics vary and may be repeated for credit when topics vary. Prerequisite: Permission of the instructor. *Does not count toward either the BA or BS degree in Computer Science.*

Additional Information

These are practical (non-theoretical) courses dealing with systems software and programming requiring about 30 clock hours of work. Some of the course materials are interactive workbooks meaning that the student is expected to have an appropriate computing environment available or in some cases, a web based interface may be available. These courses are appropriate for highly motivated students, who are resourceful, capable of independent work, and are seeking to learn a specific skill. Students are expected to obtain their own textbooks.

Evaluation

Course grades may be determined by an exam, term paper, or term project and may include computer demonstrations and/or presentation by the student. The final grade is subjective. Distance students may be required to make arrangements through a proctor who must submit a notarized statement of identity. Many of the courses use the work report form to keep track of your work. For a topic in programming languages use the programming language work report form.

Topics

OS & System & Network Administration

INFO250A Syst Software: Introduction to Unix 1.0
INFO250B Syst Software: Advanced Unix 1.0
INFO250C Syst Software: Unix Shell Prog 1.0
INFO250D Syst Software: Unix System Admin 1.0
INFO250E Syst Software: Unix Network Admin 1.0
INFO250F Syst Software: Web Server Admin 1.0

Database

Software engineering & CASE tools

Development with CVS, Bugzilla & Make
Lex & Yacc (Flex & Bison)
Standard Template Library (STL)
OO-Design and UML
Design Patterns

Webmaster Curriculum (WOW)

INFO250G Syst Software: SQL Programming 1.0
INFO250H Syst Software: Oracle Appl Prog 1.0
Oracel DBA
Oracle SQL
Oracel Forms
Oracle PL/SQL

Introduction to HTML
Introductory Webserver Administration
Webserver Security & Maintenance
Web Marketing
Project Management
Web Interface Design
Advanced Web Server Administration
e-Commerce & Internet Law

GUI Programming

INFO250I Syst Software: Visual C++ MFC 1.0
GUI Design
GUI Programming

Documentation

DocBook, TeX, LaTeX

Programming Languages

INFO250J Syst Software: Perl Programming 1.0
INFO250K Syst Software: Python 1.0
INFO250L Syst Software: PHP 1.0
Awk and Sed Programming
Java2 Programming
Java Script

Suggested textbooks **- Textbook publishers**

- www.coriolis.com
- www.course.com
- www.ddcpub.com
- www.fortuitous.com
- www.idgbooks.com
- www.manning.com
- www.mcp.com
- www.newriders.com
- www.oreilly.com
- www.osborne.com
- www.phptr.com
- www.wrox.com
- www.westnetinc.com

Other Topics

Software Engineering

- Development with CVS & Bugzilla

    - Karl Fogel *Open Source Development with CVS* Coriolis 1999
- Standard Template Library (STL)
    - *Text:* Murray & Pappas *Visual C++ Templates* Prentice Hall PTR 1999

System Administration

- MS Windows2000 server

  Unix System Administration

XML

- IDG Books

Web Programming

- Jones & Batchelor *Open Source Linux Web Programming* IDGBooks 1999

Last Modified - . Send comments to aabyan@wwc.edu

INFO 250 Work Report Form

Generic

Name: **Date:** **Grade:**

Instructions: Since the topics and content of INFO 250 are highly variable, evaluation is also variable and course expectations are dependent on the topic. Complete this form and hand it in with your work as described in the next section. Place a completed copy of this form and all work in a public directory and email its location. Or. place a completed copy of this form and all programs in a directory, tar and compress the work, and email the tar file.

Textbook:

Description of required work (completed in consultation with the instructor:

Reading	
Workbook	
Programs	
Other	
Exam	Oral: Written: None:

Workbooks should be completed. Programs should be fully functioning with full documentation, unit and functional tests, and full attribution of source and assistance. Student should be prepared to explain each line of code and alternative designs.

Instructor's Signature:

Weekly Activity Summary

Date	Activity	Time spent

Total		

Depending on the material one of the following columns will be used to determine your grade.

Hours	Percent	**Letter Grade**
27	90	A
24	80	B
21	70	C
15	50	D
		F

INFO 250 Work Report Form

For topic in a programming language

Name: **Date: Grade:**

Instructions: Complete the following to receive credit for your work.

Textbook:

Language features checklist:

Language Feature	Reading (pages & time)	Exercises and or program
Declarations • Literals • Identifiers • Constants • Variables • Types • Procedures, exceptions • Blocks, scope, & vsibility		
Expressions • Standard • Functions & parameters		
Commands • Statements • Procedure & parameters		

Data types • Basic • Structured		
Objects		
Exceptions		
Threads		
Interfaces & modules		
Generics		
Idioms		
Case Study		
Other:		

Depending on the material one of the following columns will be used to determine your grade.

Hours	Percent	**Letter Grade**
27	90	A
24	80	B
21	70	C
15	50	D
		F

Course Evaluation/Comments:

INFO 250A-F,J Sys. Software

Instructions

- Obtain a copy of the textbook The book store does not stock the textbook. You need to order a copy for yourself.
- Each chapter consists of reading, labs and projects. Do what is necessary to be comfortable with the material.
- Take the online quizzes and have the results e-mailed directly to me.
- Keep track of your progress using the progress check sheet below. The time includes reading the text, doing the exercises, projects, and quizzes.
- Upon completion of the course, hand in your workbook and a copy of your progress check sheet.
- The final exam is at the discretion of the instructor and may be a written exam, an oral exam, or a practical exam.

Progress Check sheet

Week	Comments/Description/Accomplishments/Progress	Time
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
	Total	

Suggested Textbooks

INFO 250 System Software: Lex & Yacc

Use the notes on Lex and Yacc, modify the sample compiler and keep track of your progress using the work report form.

INFO 250 System Software: OO-Design & UML

Use one or more of the texts listed below and keep track of your progress using the work report form.

Texts

Fowler & Scott UML Distilled, Second Edition: A Brief Guide to the Standard Object Modeling Language Addison-Wesley 1999

Kulak & Guiney Use Cases: Requirements in Context Addison-Wesley 2000

Page-Jones, Meilir Fundamentals of object-oriented design in UML Addison-Wesley Longman Pub Co 2000

INFO 250 System Software: Patterns

Use one or more of the texts listed below and keep track of your progress using the work report form.

Texts

Cooper, James Java design patterns: a tutorial Addison-Wesley 2000

Gamma et. al Design patterns: elements of reusable object-oriented software Addison-Wesley 1995

Buschmann et. al Pattern-Oriented Software Architecture: A System of Patterns Wiley1996

Martin Fowler Refactoring: Improving the Design of Existing Code Addison-Wesley 1999

Thomas Kühne A Functional Pattern System for Object-Oriented Design Verlag Dr. Kovac 1999

INFO 250 Technical Documents

Assignments

Currently underdevelopment. Use the following texts:

- Math Majors - TeX & LaTeX
- CS Majors - Walsh & Muellner *DocBook: The Definitive Guide* O'Reilly & Associates, Inc. See also www.oasis-open.org/docbook
- LyX - advanced document processor which produces LaTeX.

These exercises should take approximately 30 hourse to complete.

Topics and activities

	Topic	Text	Assignments	Time
1	Ccoding style & standards			
2	Man page			
3	Info file			
4	Command line help			
5	DocBook			
6	Tex/LaTeX			

Last Modified - . Comments and content invited aabyan@wwc.edu

INFO 250 System Software

Textbooks

- *INFO250A* MacMullan, John *Unix Users Interactive Workbook* Prentice-Hall PTR 1999
- *INFO250B* MacMullan, John *Advanced Unix Users Interactive Workbook* Prentice-Hall PTR 2000
- *INFO250C* Vickery, Chris *Unix Shell Programmer's Interactive Workbook* Prentice-Hall PTR 1999
- *INFO250D* Kaplenk, Joe *Unix System Administrator's Interactive Workbook* Prentice-Hall PTR 1999
- *INFO250E* Kaplenk, Joe *Linux Network Administrator's Interactive Workbook* Prentice-Hall PTR 2000
- *INFO250F* Mohr, Jim *Unix Web Server Administrator's Interactive Workbook* Prentice-Hall PTR 1998
- *INFO250J* Lowe, Vincent D. *Perl Programmer's Interactive Workbook* Prentice-Hall PTR 2000
- For courses available on request
    - Analyzing E-Commerce and Internet Law Interactive Workbook by Brinson *et al.*

      - Learn how to align Web sites with your organizational and e-commerce strategy and the current state of Internet law issues.
    - Oracle DBA Interactive Workbook by Melanie Caffrey and Douglas Scherer

      - Intended for beginners in the world of Oracle database administration, this hands-on guide takes you from creating a database to fine tuning performance. Also available as part of Oracle Database Administration: The Complete Video Course.
    - Oracle SQL Interactive Workbook by Alex Morrison and Alice Rischert

      - Uses the proven-successful format of the interactive workbook to teach SQL programming on an Oracle database.
    - Oracle Forms Interactive Workbook by Baman Motivala

      - The fastest way to master Oracle Forms with coverage of every key Oracle Forms technique. Also available as part of Oracle Forms Developer: The Complete Video Course.
    - Oracle PL/SQL Interactive Workbook by Benjamin Rosenzweig and Elena Silvestrova

      - Master Oracle PL/SQL fast with this complete book-and-Web hands-on course. Also available as part of Oracle PL/SQL: The Complete Video Course.
    - HTML User's Interactive Workbook by Alayna Cohn and John Potter

- Master HTML and start creating Web pages now!

- Understanding Web Development Interactive Workbook by Arlyn Hubbell

  - Start your Web career off right.

- Administrating Web Servers, Security, and Maintenance Interactive Workbook by Eric Larson and Brain Stephens

  The nuts and bolts of building, configuring, and maintaining Web sites, including how to maintain security.

- Exploring Web Marketing and Project Management Interactive Workbook by Donald Emerick and Kim Round, with Susan Joyce

  - Develop a sound Internet strategy, build an effective Web team, and understand the legal and marketing issues of your growing e-business.

- Java 2 Programmer's Interactive Workbook by Kevin Chu and Eric Brower

  - Master the Java programming language now, with this easy, hands-on introduction-the perfect course for absolute beginners.

- Linux Network Administrator's Interactive Workbook by Joe Kaplenk

  - Learn all the Linux networking skills you need with this integrated book-and-Web learning solution.

- A+ Certification Interactive Workbook by Emmett Dulaney and Robert Bogue

  - Master every skill covered in both A+ certification exams through a series of real-life labs. Includes coverage of PC components, peripherals, networking, and more!

- Advanced UNIX User's Interactive Workbook by John McMullen

  - Become a UNIX Power User *now*! Control your environment, including scripts, start-up files, X configuration, and email, networking, and file management skills.

- Perl Programmer's Interactive Workbook by Vincent D. Lowe

  - Master Perl programming *now*!

- UNIX System Administration Interactive Workbook by Joe Kaplenk

  - Master the technical and "thinking" skills you need to administer any UNIX system.

- UNIX Web Server Administrator's Interactive Workbook by Jim Mohr

  - Master the world's #1 Web server, Apache!

- UNIX User's Interactive Workbook by John McMullen

  - This hands-on workbook starts with basics of login and logout and brings you up to power-user status quickly.

- UNIX Awk and Sed Programmer's Interactive Workbook by Peter Patsis

  - A quick, friendly, hands-on tutorial on UNIX programming with awk, sed, and grep.

- UNIX Shell Programmer's Interactive Workbook by Chris Vickery

  - Whatever your experience, UNIX Shell Programmer's Interactive Workbook will transform you into a power shell programmer, fast!
- *Designing Web Interfaces, Hypertext, and Multimedia* by Reese, White, and White
- Supporting Web Servers, Networking, Programming, and Emerging Technologies by White, Dara-Abrams, and Aleem

INFO250 UNIX System Administration - 1

Description:

Introduction to UNIX system administration basics including startup, shutdown, user accounts, the file system, system backup and restore procedures, device installation, simple network management, print service, process management and system security. Prerequisite: Previous user level experience with Unix.

Textbook: Kaplenk, Joe **Unix System Administrator's interactive workbook** Prentice-Hall PTR 1999

Instructions:

- For each chapter do the following:
    1. Complete all assigned **Labs** in the workbook. As an alternative, construct an electronic document containing your answers.
    2. Complete all lab **Self Reviews** placing your answers in your book or in an electronic document
    3. Answer each assigned chapter's **Test Your Thinking** questions placing your answers in an electronic document. Be sure to check your answers at the **publisher's web site**.
    4. Take the **Practice Questions exam** at the **publisher's web site** and have your score emailed to aabyan@wwc.edu. You should take one exam per week. Failure to do so may result in a lower grade for the course.
- You may work with other students on the Labs but not on the Test Your Thinking questions or the Practice Questions exam. You are honor bound to follow this requirement.
- The final exam is an oral and/or practical exam. During during *dead week*, schedule the final oral exam to take place during *test week*.

Grading:

Grades are subjective but are based on

1. your completed workbook or electronic documents containing your work,
2. your electronic documents containing your answers to the Test Your Thinking questions,
3. the emailed results of your Practice Questions exam, and

4. the final oral/practical exam.

Progress: Use the following form to record your progress. Note both the date completed and the score received.

Chapter	Topic	Labs	Date/Score	Self Review	Test your thinking Date & Score	Practice Questions Date & Score
1	System Security	1.1 1.2 1.3				
2	The Bourne Shell User	2.1 2.2				
3	The Korn Shell User	3.1 3.2				
4	The C Shell User	4.1 4.2 4.3				
5						
6						
7						
8						
9						
10						

Goal

Upon completion of this course you will be able to perform the following system administration tasks for a UNIX environment and have the skills of a Junior Systems Administrator.

- Perform System Startup and Shutdown
- Manage User Accounts
- Manage the File System
- Backup and Restore Files
- Install serial communication devices: Terminals and Modems
- Manage a UNIX Network: workstations
- Manage the UNIX Print Service
- Perform Job scheduling (task automation) with cron
- be familiar with security, system accounting, system monitoring and performance issues.

Assignments

Resources

Textbook:

Kaplenk, Joe **Unix System Administrator's interactive workbook** Prentice-Hall PTR 1999
Komarinski & C. **Linux System Adminstration Handbook** Prentice-Hall PTR 1998.
Nemeth, et al. *UNIX System Administration Handbook 2/e*, Prentice-Hall 1995.
Wang, Paul. *An Introduction to Unix with X and the Internet*, PWS Publishing Company.
Pearce, Eric. *An Overview of Windows NT & UNIX Integration from a Unix Perspective*, O'Reilly 1998
Henriksen, Gene. *Windows NT and UNIX Integration*, Macmillan Technical Publishing 1998

Video:

An Introduction to UNIX System Administration by Ray Swartz

References:

- For an excellent source of systems adminstration references see: http://www.oreilly.com

USENET News Groups:

comp.unix.admin, comp.unix.shell, comp.os.linux.*, comp.os.bsd.*

WWW:

- Online courses
  OSU Basic Unix Guide
  OSU SysAdmin course
  UW SysAdmin course
  Unix System Administration Independent Learning (USAIL)
- Berkeley
- Unix Guru or Unix Guru
- Jumbo SysAdm Site
- Stokely Consulting

- Usenix and SAGE
- Uniforum
- SunWorld Online
- Unix System Administration Magazine

Students interested in pursuing a career in systems administration should become members of USENIX and SAGE.

Lab Notebook

Each student is required to keep a laboratory notebook containing an activity log. It may be kept in either paper form (bound or unbound) or electronically. Typical entries will include

- Job/Problem Description
- Date/Time (completed,time spent)
- Who (performed the activity)

- Solution, Method & System Modifications (required to solve the problem)
- Difficulties (encountered in performing the activity)
- Parameters & System Dependencies (required to solve the problem)

You will be required to submit either your notebook or copies of your entries.

Project

- Learn a scripting language: eg. shell programming, perl. Provide an introductory guide to the language and sample programs (minimum 5 pages).
- Software installation: eg. programming language, server (www). Provide documentation and summary of experience (minimum 2 pages).
- System service: eg. webpages. Provide documentation and summary of experience (minimum 2 pages).
- Readings: Summarized in a short paper (minimum 4 pages)
- Future Topics: develop lecture notes and exercises in html format for one of the future topics listed.

Evaluation

The lab grade is based upon completion of the lab manual exercises and the completeness of the lab notebook. The lab exercises and appropriate pages from the lab notebook are due one week after the lab. Oral EXAM - Final grade is subjective.

Grade form

Grading Information

Integrity, Computing, & Disability Policies

Grades

- What does your grade mean? Description of "A" and "C" students
- Your preparation for or likely hood of success in industry.
  - Skills: foundational, business, & technical
  - *Top 10 signs you are a stellar software developer*
- Letter grade/percent conversion

Letter Grade	Points/ Percent
A	90-100
B	80-90
C	60-80
D	50-60
F	0-50

- Grading worksheets
  - Programming assignments: programs must be well documented and use the standard program heading.
    - program heading
    - program grading criteria
  - CPTR 141 Intro to Programming
  - CPTR 352 OS worksheet
  - CPTR 415 DB worksheet
  - CPTR 425 Networking worksheet
  - CPTR 435 SE worksheet
  - CPTR 460 Parallel worksheet
  - CPTR 464 Compiler worksheet

Article/Book Review -- you have several options

- write a summary
- write a reaction
- write a "What I have learned"
- write a "What I will be able to do from what I have read"

Grades - grading is an inherently subjective process. I reserve the right to exercise my best judgment. The material in this section is for illustration purposes only.

- Letter grades and percentages

GRADING WEIGHTS depends on the class but often		LETTER GRADES	
Labs & homework	50%	As	90 - 100%
Tests	50%	Bs	80 - 89%
		Cs	70 - 79%
		Ds	60 - 69%

-

Projects and project courses - group activities require additional documentation beyond the deliverables. A project time card must be maintained. A self-evaluation must be completed and at least two peer reviews are required for each evaluation period.

- Time card - Due the first class of each week.
- Performance review - performed by the instructor each review period.
- Peer review - two required per review period, peers submit their evaluation directly to the instructor.
- Self evaluation - one required per review period.
- Performance level guidelines and categories

Course Evaluation

- Course evalutation

Last Modified
Send comments to aabyan@wwc.edu

Grade Expectations

The "A" Students -- Outstanding Students

Attendance
: Virtually perfect attendance. Their commitment to the class resembles that of the teacher.

Preparation
: Always prepared for class. They always read the assignment. Their attention to detail is such that they occasionally catch the teacher in a mistake.

Curiosity
: Show an interest in the class and in the subject. They look up or dig out what they don't understand. They often ask interesting questions or make thoughtful comments.

Retention
: Have retentive minds. They are able to connect past learning with the present. They bring a background with them to the class.

Attitude
: Have a winning attitude. They have both the determination and the self-discipline necessary for success. They show initiative. They do things they have not been told to do.

Talent
: Have something special. It may be exceptional intelligence and insight. It may be unusual creativity, organizational skills, commitment -- or a combination thereof. These gifts are evident to the teacher and usually to the other students as well.

Results
: Make high grades on tests -- usually the highest in the class. Their work is a pleasure to grade.

The "C" Students -- Average or Typical Students

Attendance
: Miss class frequently. They put other priorities ahead of academic work. In some cases, their health or constant fatigue renders them physically unable to keep up with the demands of high-level performance.

Preparation
: Prepare their assignments consistently but in a perfunctory manner. Their work may be sloppy or careless. At times, it is incomplete or late.

Attitude
: Not visibly committed to the class. They participate without enthusiasm. Their body language often expresses boredom.

Talent
: They vary enormously in talent. Some have exceptional ability but show undeniable signs of poor self-management or bad attitudes. Others are diligent but simply average in academic ability.

Results

Obtain mediocre or inconsistent results on tests. They have some concept of what is going on but clearly have not mastered the material.

From *Clarifying Grade Expectations* by John H. Williams in **The Teaching Professor**

Building blocks of a successful IT career

Foundation Skills	Business Skills
Ability to learn new skills Analytic capabilities and problem-solving skills Communication skills (verbal and written) Flexible Self-motivated Collaboration/teamwork Broad education and global perspective	General management Project management Leadership Conflict resolution Understanding of business operation
Technical Skills	**References are asked about the following traits**
Current technologies Programming languages	Strengths and weaknesses Work ethic Personality Relationship with supervisors Customer service skills Ability to work under stress Communication & organizational skills

Reference misplaced.

Factors that contribute to innovation, broader application of technology, and valued by employers.

- Intellectual accomplishment in other disciplines.
- Leadership
- Motivation
- Communication skills
- Breadth of ability and experience
- Social commitment

From

American Society for Engineering Education. (1994) *Engineering Education for a Changing World.* Joint project report of the Engineering Deans Council and the Corporate Roundtable of the ASEE, http://www.asee.org/publications/reports/green.cfm.

Dahir, M. (1993) "Educating engineers for the real world," *Technology Review*,

August/September, pp. 14-16.

Questions asked of references by prospective employers.

- System Administrator position (2000.01.31)
    1. What is the length and nature of your relationship to ....?
    2. Please describe and rate (excellent, good, fair, poor) his Customer Service skills.
    3. What would you say are his strengths?
    4. What would you say are his weaknesses?

Maintained by WWC CS Department
Last Modified
Send comments to webmaster@cs.wwc.edu

Top 10 signs you are stellar software developer

Maggie Biggs in INFOWORLD Electric

No. 10 Superlative technical skills: A great developer is highly skilled in analysis, software design, good coding practices, and debugging.

No. 9 Strength in business skills: A great developer understands the business impact of the application providing a powerful competitive advantage for the organization.

No. 8 Speak English as well as "tech-ese": Software developers communicate across larger sections of the organization.

No. 7 Paradigm openness: A great developer uses the paradigm which makes the most sense for the project and the environment at hand.

No. 6 No tool or platform marriages: A great developer views his or her skills outside the scope of tools and platforms, and applies the best solutions.

No. 5 Leaning as an ongoing process: A great developer does not stand still in knowledge attainment. Great developers continually challenge them to learn new skills.

No. 4 Code reuse: A great developer always maintains common code for later reuse and will share code with other developers as an expanded means of reuse.

No. 3 Collaboration skills: A great developer utilizes other developer skills to ask questions, provide help on a particular issue, or to discuss best development practices.

No. 2 Teach others: A great developer gives back to the software community by means of presentations and helping other developers.

No. 1 A sense of humor: A great developer with heavy project schedules and the demands of solving ever-complex business and technical issues finds balance in having a good sense of humor .

Last Modified Tue Feb 23 09:37:46 1999

Send comments to webmaster@cs.wwc.edu

```
/*****************************************************************************
 Program file name:
 Class:  Assignment:
 Language:  Operating System:
 Compiler:

 Programmer:
 Date Written:
 Revisions:

 Description:

 Inputs:

 Outputs:

 Special requirements:

 Criteria Grades (0 to 5 points each):
        Program Design:                 _____ x 5% = _____
        Program Execution:              _____ x 4% = _____
        Specification Satisfaction:     _____ x 4% = _____
        Coding Style:                   _____ x 3% = _____
        Comments:                       _____ x 2% = _____
        Creativity:                     _____ x 2% = _____
        Late Submission Penalty:                   _____
                                           Total % = _____
 Overall Program Grade:
                            Program's Point Value = _____
                                  Program's Score = _____
 Comments:

*****************************************************************************/
```

Program Grading Criteria

- Program execution (20%) Programs should compile and run cleanly, and produce correct results: proper values, formatting, and completeness.
    - 5 -- Program runs correctly.
    - 3 -- Program produces correct output half of the time.
    - 1 -- Program runs, but mostly incorrect.
    - 0 -- Program does not compile or run at all.
- Specification satisfaction (20%) Programs should satisfy their specifications: required language constructs, modules, proper i/o formats.
    - 5 -- Program satisfies specifications completely and correctly.
    - 3 -- Many parts of the specification not implemented.
    - 1 -- Program does not satisfy specifications.
- Program design (25%) Programs should be well designed: program structure, modularity, algorithm and data structure selection.
    - 5 -- Solution well thought out.
    - 3 -- Solution partly planned.
    - 1 -- *ad hoc* solution; program "designed at the keyboard".
- Coding style (15%) The program should be designed for readability: variable and subprogram names, language constructs and capabilities, and white space.
    - 5 -- Well-formatted, understandable code; appropriate use of language capabilities; good variable and subprogram names.
    - 3 -- Code hard to follow in one reading; poor use of language capabilities.
    - 1 -- Incomprehensible code, appropriate language capabilities unused.
- Comments (10%) The program should be well commented for administrative purposes and to enhance understanding of the code: administrative information, pre- and post-conditions for each subprogram, brief description of each logical paragraph of code.
    - 5 -- Concise, meaningful, well-formatted comments.
    - 3 -- Partial, poorly written or poorly formatted comments.
    - 1 -- Wordy, unnecessary, incorrect, or badly formatted comments.
    - 0 -- No comments at all.
- Creativity (10%) A program is deserving of an `A' when it provides more than expected: an interesting solution, more robust than required, particularly user friendly interface, code that is a pleasure to read.
    - 0 to 5 points to programs that usefully extend the requirements, that use the capabilities of the language particularly well, that use a particularly good algorithm, or that are particularly well-written.

---

adapted from *On Criteria for Grading Student Programs* by James W. Howatt in SIGSCSE BULLETIN vol 26 No 3 Sept 1994 p 3

Last update:

Send comments to: webmaster@cs.wwc.edu

CPTR 141 Grade Worksheet

Name: **Grade:**

Instructions: Hand in hard copies of fully functioning programs

Program Summary

Homework	Score (%)	Factor	Score
Lab 1		x 0.083	
Lab 2		x 0.083	
Lab 3		x 0.083	
Lab 5		x 0.083	
Lab 6		x 0.083	
Lab 8		x 0.083	
Tests			
Midterm		x 0.25	
Final		x 0.25	
		Total	

The total should be a number between 0 and 100. The following table may be used to obtain the letter grade.

Letter Grade	Points/Percent
A	90-100
B	80-90
C	60-80
D	50-60
F	0-50

Resources

- program heading

- program grading criteria

SE Grade Worksheet

NAME: **GRADE:**

Grade Calculation

Use the following table to determine your points for the course.

	Percent		**Estimate/score**		**Points**
Documents	40%	Estimated percent		*2/5=	
Evaluation	30%	Average Score		* 5 =	
Time Cards	10%	Total hours		/12 =	
Participation	20%	Score		*2/3=	
				Total	

Enter your course letter grade at the top of the worksheet based on the following table.

Letter Grade	**Points/Percent**
A	90-100
B	80-90
C	60-80
D	50-60
F	0-50

Documents 40%

Documents are the course deliverables and include software, documentation, and for this course, the course environment.

Perform a rough estimate of the percent of your direct contribution to the course deliverables. Points are calculated by multiplying your estimate by two and dividing by five.

Evaluation 30%

Evaluation refers to the self evaluation and peer review documents. Compute the average of your evaluation levels on your self evaluation and peer evaluations. Points are calculated by multiplying the average by five.

Time Cards 10%

Total your documented hours for the course. Your points are calculated by dividing your total hours by twelve.

Participation 20%

Participation refers to verbal participation in class discussion. Key elements include leadership, suggestion of alternatives, evaluation of alternatives, frequency, quality, and influence on the final outcome. Points may be determined by reference to the following table.

Category	**Evaluation** (circle one)	**Comments**
Leadership	1 2 3 4 5 6	
Frequency	1 2 3 4 5 6	
Suggestions	1 2 3 4 5 6	
Quality	1 2 3 4 5 6	
Influence	1 2 3 4 5 6	
Total		

Points are calculated by summing the circled values and multiplying by 2/3.

Level 6 - Exceptional participation with consistently high enthusiasm, quality, quantity, and influence on the final outcome.

Level 5 - Excellent participation with consistent enthusiasm, quality, quantity, and influence on the final outcome.

Level 4 - Above average participation with frequent contributions of a high quality.

Level 3 - Average participation, satisfactory and acceptable.

Level 2 - Minimally/marginally acceptable participation.

Level 1 - Rarely participates. Exhibits little interest in the proceedings.

Time Card

Name:
Week ending:
Class:
Instructor:

Date	Activity	Start	Stop	Time
			Total	

Instructions: Record activity, date, and time and total the time spent.

Performance Review

Name:

Course:

Instructor:

Date:

Evaluation period

From:
To:

Definitions for the performance categories and performance level guidelines.

Performance Category	Evaluation (circle one)	Comments
Quality of Work	1 2 3 4 5 6	
Quantity of Work	1 2 3 4 5 6	
Initiative	1 2 3 4 5 6	
Planning	1 2 3 4 5 6	
Ability to Work With Others	1 2 3 4 5 6	
Adaptability	1 2 3 4 5 6	
Communications	1 2 3 4 5 6	
Cooperation & Teamwork	1 2 3 4 5 6	
Job Knowledge	1 2 3 4 5 6	
Leadership	1 2 3 4 5 6	
Average:		

Overall Evaluation of Employee Performance Level: 1 2 3 4 5 6

Comments:

Attendance ___ Problem ___ No Problem Comments:

What are this student's strengths?

Please provide specific examples of this students's major achievements during the review period.

How can this student improve his/her performance?

What training or learning experience would help this student improve his/her performance?

What goals should this student reach between now and the end of the next review period?

II. Employee Comments (Optional)

__ No Comments

General comment about the evaluation of your performance:

Read and acknowledged by:

Student_____Date_____
(Employee signature only indicates receipt of appraisal and is not necessarily in agreement.)

Peer Review

This is a confidential review.

Peer review for Name:

Reviewer Name:

Course:

Instructor:

Date:

Evaluation period

From:
To:

Definitions for the performance categories and performance level guidelines.

Performance Category	Evaluation (circle one)	Comments
Quality of Work	1 2 3 4 5 6	
Quantity of Work	1 2 3 4 5 6	
Initiative	1 2 3 4 5 6	
Planning	1 2 3 4 5 6	
Ability to Work With Others	1 2 3 4 5 6	
Adaptability	1 2 3 4 5 6	
Communications	1 2 3 4 5 6	
Cooperation & Teamwork	1 2 3 4 5 6	
Job Knowledge	1 2 3 4 5 6	
Leadership	1 2 3 4 5 6	
Average:		

Overall Evaluation of Employee Performance Level: 1 2 3 4 5 6

Comments:

Attendance ___ Problem ___ No Problem Comments:

What are this student's strengths?

Please provide specific examples of this student's major achievements during the review period.

How can this student improve his/her performance?

Signatures

Reviewer:	Date
Instructor:	Date

Self Evaluation

Name:

Course:

Instructor:

Date:

Evaluation period

From:
To:

Definitions for the performance categories and performance level guidelines.

Performance Category	Evaluation (circle one)	Comments
Quality of Work	1 2 3 4 5 6	
Quantity of Work	1 2 3 4 5 6	
Initiative	1 2 3 4 5 6	
Planning	1 2 3 4 5 6	
Ability to Work With Others	1 2 3 4 5 6	
Adaptability	1 2 3 4 5 6	
Communications	1 2 3 4 5 6	
Cooperation & Teamwork	1 2 3 4 5 6	
Job Knowledge	1 2 3 4 5 6	
Leadership	1 2 3 4 5 6	
Average:		

Overall Evaluation of Performance Level: 1 2 3 4 5 6

Comments:

Attendance ___ Problem ___ No Problem Comments:

Please provide specific examples of your major achievements during the review period.

How can you improve your performance?

What goals should you reach between now and the end of the next review period?

Signatures

Student:	Date
Instructor:	Date

Performance Level Guidelines

Level 6 - Far Exceeds Job Requirements Performance at this level far exceeds the maximum requirements of this position. Duties and responsibilities are exceptionally met and consistently exceeded.

Level 5 - Consistently Exceeds Job Requirements Performance at this level is at the maximum and always beyond acceptable requirements for the position. Duties and responsibilities are not only excellently met, but consistently exceeded.

Level 4 - Meets and Usually Exceeds Job Requirements Performance at this level is above average in acceptable requirements for the position. Duties and responsibilities are well met and usually exceeded.

Level 3 - Consistently Meets Job Requirements Performance at this level is at the average of acceptable requirements for the position. Duties and responsibilities are met consistently and in a satisfactory and acceptable manner.

Level 2 - Inconsistent in Meeting Job Requirements Performance at this level is at the minimum of acceptable requirements for the position. Duties and responsibilities are marginally met.

Level 1 - Does Note Meet Job Requirements Performance at this level is below the minimum of acceptable requirements for the position. Duties and responsibilities are not met in an acceptable manner.

Performance Categories

Quantity of Work - Volume of work regularly produced. Speed and consistently of output.
Quality of Work - Extent to which employee can be counted upon to carry out assignments to completion.
Initiative - Extent to which employee is a self starter in attaining objectives of the job.
Planning - Extent to which employee is able to sequence activites to maximize production and/or anticipate change. ?
Job Cooperation - Amount of interest and enthusiasm shown in work.
Ability to Work With Others - Extent to which employee effectively interacts with others in the performance of his/her job.
Adaptability - Extent to which employee is able to perform a variety of assignments within the scope of his/her job duties.
Communications - Extent to which employee ...
Cooperation & Teamwork - Extent to which emloyee ...
Job Knowledge - Extent of job information and understanding possessed by employee.
Leadership - Extent to which employee exhibits ability to direct others in the their work. ?

CPTR 435 SE - Course Evaluation

Winter 2000
Instructor: Anthony Aaby

Instructions: For each, circle the number that best reflects your preference - 1 is least agree, 6 is most agree.

Course content/Textbook

The textbook helps to define the subject area of a course.

Category	Evaluation Comments
	(circle one)
The textbook corresponds with my view of S.E.	1 2 3 4 5 6
Will keep the textbook for future reference.	1 2 3 4 5 6
The textbook is too hard.	1 2 3 4 5 6
The textbook is poorly organized.	1 2 3 4 5 6
There is too great a leap from small programs in previous courses to the large project concept in this course.	1 2 3 4 5 6
Previous courses have prepared me for this material.	1 2 3 4 5 6
CASE tools should be introduced prior to this course.	1 2 3 4 5 6
CASE tools should have been selected prior to this course and used from the beginning.	1 2 3 4 5 6
	1 2 3 4 5 6

Course Organization

Category	Evaluation Comments
	(circle one)
The course should have a traditional lecture organization.	1 2 3 4 5 6
The course should be two quarters in length. First quarter a traditional lecture course. Second quarter a project.	1 2 3 4 5 6
There was too much emphasis on the project.	1 2 3 4 5 6
The project was too big.	1 2 3 4 5 6

The project should have been better defined with key documents provided so that the work could have focused on design and implementation.	1 2 3 4 5 6
I disliked this course.	1 2 3 4 5 6
	1 2 3 4 5 6

The Project

A project provides opportunity to practice the concepts.

Category	**Evaluation Comments**
	(circle one)
The project was too big.	1 2 3 4 5 6
The project was not in my area of interest.	1 2 3 4 5 6
A quarter is too short to have a project.	1 2 3 4 5 6
The project should be small and well defined.	1 2 3 4 5 6
Students should be able to pick their own project.	1 2 3 4 5 6
The project should be complex and mult-year in length.	1 2 3 4 5 6
I really wanted to spend most of my time coding.	1 2 3 4 5 6
I disliked this project.	1 2 3 4 5 6
	1 2 3 4 5 6

In the space available, suggest additional projects that you feel would give you marketable skills.

Instructor

Category	**Evaluation Comments**
	(circle one)
The instructor should be a software engineer.	1 2 3 4 5 6
	1 2 3 4 5 6

A Natural Language Processor

Author: Anthony Aaby
Document status: unfinished
History: Current document initiated April 1999, Initial coding c. 1990.

Abstract: A program for demonstrating natural language processing. The program accepts grammatical sentences in a subset of English and constructs a database. Questions posed in English result in queries against the database. The replies are in logical form. The program does not formulate natural language replies.

Introduction

Among the welter of varied linguistic features, two important grammatical relations seem to be held in common by all known languages:

1. some kind of *actor-action-goal* relation;
2. some kind of relation between names of objects and modifying qualities.

There are a variety of nonobligatory grammatical relations.

Number: singular, dual, or plural - actor-action agreement

Defniniteness: determiners that preceed a noun.

> *definite*: the, this (these), that (those) - may be preceeded by *all*
> *indefinite*: a, an, any, each, either, neither, every, no one, somewhat, whatever, which, whichever, many a , such a, what a - may not be preceeded by all.
>
> Representation in logic - *determiner noun*
> Existential determiners - exists (x , noun(x)): singular - a, an, this, the; some
> Universal determiners - all (x, noun(x)): singular - any, each, every; plural - all, the

Tense: when the action took place (past, present, future).
Mood (or *mode*): how the speaker reguards the action; expressed with *verbal auxiliaries*
Aspect: finshed or proceeding in the past; expressed with *verbal auxiliaries*
Comparison:

> First degree: ... is as valuable as ...;
> Comparative degree: ... is more valuable than ...;

Superlative degree: ... is the most valuable of all ...;

Gender: masculine, feminine, and neuter.
Voice: active, passive

Active	acts
Passive	acted upon
Dynamic	performs the action for itself
Reflexive	turns the action upon itself

Case: expressed by prepositions

the relationship between a noun and a pronoun and some other noun or pronoun in the same clause or phrase.

Person:

	singular	**plural**
first person	I, me	we, us
second person	you	you, you
third person	he, she, it him, her, it	they, them

In English, these relations are expressed through three devices

Fixed word order	*actor* (subject), *action* (predicate), *action-goal* (object); *modifiers noun.*
Relation words	prepositions express case; verbal auxiliaries express mood and aspect; conjunctions express relationships between phrases.
Changing the word form	express number, gender, tense, or comparison; pronouns express relationships between continguous sentences or parts of sentences.

Many other languages express the relations through *inflections* (suffixes attached to words).

The essential grammar of modern English applies to word groups rather than to the word as such. Thus, the grammar is described as *analytical* or a primarily *syntactical*, language.

Logical form

The *actor-action-goal* may be expressed in first-order logic as *action* (*actor*, *goal*) eg John loves Mary is represented as loves(John, Mary). Sometimes the goal is not present as in John runs. The logical representation is runs(John). A man loves a woman is represented as exists(x,man(x)) and

exists(y woman) and loves(x,y).

Definite Clause Grammar

A definite clause grammars are an syntactical variant of Prolog. They are attribute grammars. In their simplest form they resemble context-free grammars.

nonterminal `-->` *comma separated list, possibly empty, of terminals and non-terminals*`.`

Terminals are enclosed with brackets.

`[` *comma separated list, possibly empty, of terminals* `]`

The non-terminals may be parameterized with both inherited and synthesized attributes. Prolog predicates may be included among the list of terminals and non-terminals on the righthand side of a rule and must be separated by commas and inclosed between braces.

nonterminal `--> ..., {write("hello mom"), nl},...`

The non-terminals are implicitly parameterized with the input. The input is matched against the terminals. The terminal is removed from the input and the remaining input is passed to the next term. Any remaining input is passed back through a second parameter. The queries must include the two parameters which follow any other parameters. For example, the start symbol of the grammar must include parameters for attributes and the input parameters.

`start(`*attributes*`, Input, RemainingInput)`

`Example:`

```
program     --> declaration, body.
declaration.
declaration --> type, variable, declaration.
statement   --> variable [:,=] expression.
...
```

Talk (user interface, grammar, dictionary)

A Subset of English Grammar

The Dictionary

The Scanner - `read_sentence(Sentence, [])`

Clausifier (FOL to Clause Logic) - `tranlate(FOL, CL)`

Conclusions

Further work

- Extend the dictionary to include a basic English vocabulary.
- The program should be able to expand its vocabulary and retain the new vocabulary between sessions..

  In particular, the `db`predicate should be easy to extend to allow for new predicates.
- Generate natural language sentences from logical statements permitting natural language replys.
- The program should be able to expand its grammar.
- Save and restore the database.
- Add tense handling, past, present, and future eventually, full blown temporal logic.
- Add a truth maintenance subsystem.

References

Pereira, Fernando C. N

Prolog and natural-language analysis 1987.

Talk (user interface, grammar, dictionary)

Author: Anthony Aaby
Document status: unfinished
History: Current document initiated April 1999, Initial coding c. 1990.

Once the program is loaded, it is started by entering the query `?- dm.`

```
/*************************************************************************

        TALK Program:  A Natural Language Interface for a Data Base

COMMENTS:
  1.  The program should accept grammatically correct sentences.
  2.  The program should translate grammatically correct sentences to their
      correct logical form.
  3.  The program should reject grammatically incorrect sentences.
  4.  The program should ask for clarification of grammatically incorect
      sentences.
  5.  The program should be able to expand its vocabulary.
  6.  The program should be able to expand its grammar.
  7.  The expanding program should maintain the correctness of the previous
      levels.

*************************************************************************/

/*=========================================================================
                Operator Declarations
=========================================================================*/

:- op(  30, fx, ~).
:- op( 100, xfy, #).
:- op( 100, xfy, &).
:- op( 150, xfy, =>).
:- op( 150, xfy, <=>).
```

Dialog Manager

The dialog manager

```
dm :- dialog_manager.

dialog_manager :- write('>> '),                    % Dialog prompt
                  read_sentence(Sentence, []),!,   % Read user input
                  talk(Sentence, Reply),           % Process the input
```

```
                print_reply(Reply),             % Generate the reply
                continue(Reply).

continue(quit).
continue(_) :- dialog_manager.                  % continue with more
input

%%%========================
%%%  talk( Sentence, Reply )
%%%
%%%  ==> Sentence
%%%  <== Reply
%%%========================

talk(Sentence,Reply) :- parse(Sentence, LogicalForm, Type),
                        translate(LogicalForm, Clauses), !,
                        reply(Type, Clauses, Reply).

talk(Sentence,Reply) :- Reply =
           error('I am unable to understand your sentence. Please
restate.').

%%%=======================================
%%%  reply( Type, FreeVars, Clause, Reply )
%%%=======================================

reply( quit,  _, quit ) :- !.

reply( query, ([cl([answer(Answer)],C)]), Reply ) :-
      free_vars(C,FreeVars), makebody(C,Condition),
      (setof( Answer, FreeVars^Condition, Answers )
        -> Reply = answer(Answers)
        ;  (Answer = yes
            -> Reply = answer([no])
            ;  Reply = answer([none]))),!.

reply( assertion, Assertions, asserted(Assertions) ) :-
                                    assertclauses(Assertions),!.

reply( Type,      Clause,    error('Unknown type') ).

%%% Free Variables

free_vars(C,FreeVars) :- free_vars(C,[],FreeVars).

free_vars( [], FVs, FVs ).
free_vars( [C0|Cs], Fvs, FVs ) :- c_free_vars( C0, Fvs, Ifvs ),
```

```
                                    free_vars( Cs, Ifvs, FVs ).


c_free_vars( C, Fvs, FVs ) :-
     functor(C,F,N), c_free_vars( C,Fvs,N,FVs).

c_free_vars( C, FVs, 0, FVs ).
c_free_vars( C, Fvs, N, FVs ) :-
     N > 0, arg(N,C,A), var(A), putin(A,Fvs,Fvs0),
     N1 is N - 1, c_free_vars(C,Fvs0,N1,FVs).
c_free_vars( C, Fvs, N, FVs ) :- N > 0, arg(N,C,A), nonvar(A),
                                 free_vars([A],Fvs,Fvs0),
                                 N1 is N - 1,
c_free_vars(C,Fvs0,N1,FVs).
```

Assert Clauses

```
assertclauses([]).
assertclauses([cl([Head],[])|Clauses]) :- assert(Head),!,
                                          assertclauses(Clauses).

assertclauses([cl([Head],B)|Clauses]) :- makebody(BC,Body),
                                         assert((Head :- Body)),
                                         assertclauses(Clauses).
% empty or multiple heads
assertclauses([cl(H,B)|Clauses]) :- assert(cl(H,B)),
assertclauses(Clauses).

makebody([C],C).
makebody([C|Cs],(C,Csp)) :- makebody(Cs,Csp).

%%%======================
%%%  print_reply( Reply )
%%%
%%%  ==> Reply
%%%======================

print_reply(quit) :- write('Ok.  '),
                     write('Its been a pleasure serving you.  Good-
bye!'),nl.

print_reply(error(ErrorType)) :- write('Error: "'),
                                 write(ErrorType), write('"'), nl.

print_reply(asserted(Assertion)) :- write('Asserted "'),
                                    write(Assertion), write('"'), nl.
```

```
print_reply(answer(Answers)) :- print_answers(Answers).

%%%=========================
%%%  print_answers( Answers )
%%%
%%%  ==> Answers
%%%=========================

print_answers([Answer]) :-  write(Answer), write('.'), nl.
print_answers([Answer|Answers]) :-  write(Answer), write(','),
                                   print_reply(answer(Answers)).
```

A Grammar for a Subset of English

Notation

Regular Expressions	
Alternatives	A | B
Optional	[A]
Zero or more	{ A }
Grouping	(A)
Terminals	**bold** | *italic* - reserved words & symbols only appear on the right-hand side - of grammar rules
Non-terminal	*italic font* - appear on either side of grammar rules
Grammar Rule	
Not implemented	*non-terminal* ::= *regular expression*
Implemented	*non-terminal* ::= *regular expression*

English Grammar

The essential grammar of modern English applies to word groups rather than to the word as such. Thus, the grammar is described as *analytical* or a primarily *syntactical*, language.

Sentence

Sentence	::=	*declaritiveSentence* **.**
	|	*interrogatorySentence* **?**
	|	*imperativeSentence* **!**
declarativeSentence	::=	*simpleSentence*
	|	*compundSentence*
	|	*complexSentence*
	|	*compound-complex*

imperativeSentence ::= [**You**] *verbPhrase*

interrogatorySentence ::= *interrogatoryPronoun verbPhrase*
| [*interrogatoryPronoun*] *aux nounPhrase verbPhrase*
| **Is / Are** *nounPhrase nounPhrase*

simpleSentence ::= *independentClause*

compoundSentence ::= *independentClause* { *coordinateConjunction independentClause* }

complexSentence ::= { *dependentClause* } *independentClause* { *dependentClause* }
- independent clause and one or more dependent clauses

compound-complex ::= *independentClause* ***but, though*** *dependentClause* **,** *independentClause*

Clauses and Phrases

Clause -- contains a subject-verb combination

-- by type

independentClause ::= *nounPhrase verbPhrase*

dependentClause ::= *subordinateConjunction independentClause*

-- by function

clause ::= *nounClause*
-No one could read **what he wrote**.
| *adjectivalClause*
-The man **who lives next door** is ill.
| *adverbialClause*
-**Before he started eating**, he washed his hands.

Phrase -- does not contain a subject-verb combination and functions as a single part of speech

phrase ::= *prepositionalPhrase*
-- **by introductory word** | *participialPhrase*
| *gerundPhrase*
| *infinitivePhrase*

prepositionalPhrase ::= *preposition nounPhrase*
participialPhrase ::= *participle ...*
gerundPhrase ::= *gerund ...*
absolutePhrase ::= *...*

phrase ::= *verbPhrase*
-- **by function** | *nounPhrase*
| *adjectivePhrase*
| *adverbPhrase*
absolutePhrase

nounPhrase ::= *properNoun* — Bill
| *determiner* { *adjective* } *noun* [*relativeClause*] — The big dog

verbPhrase ::= *intransitiveVerb*
| *transitiveVerb nounPhrase*
| *aux verbPhrase*
| *rov nounPhrase verbPhrase*
| **is/are** *nounPhrase*

Sentences: `parse( Sentence, LogicalForm, Type )`

Three sentence forms are recognized,

- *quit* (single word requests to terminate the session),
- *queries* (questions which are querys to the database), and
- *assertions* (statements which contain information to be asserted to the database).

The parse predicate takes a sentence and returns its logical form, and type.

```
parse( Sentence, LogicalForm, Type ) :-
   sentence(Type, LogicalForm, Sentence, []).

sentence( Type, LogicalForm ) --> imperative( Type, LogicalForm ).

sentence( query, LogicalForm ) --> interrogatory( LogicalForm ).

sentence( assertion, LogicalForm ) --> declarative( LogicalForm ).
```

Type: quit (session termination)

```
imperative(Type, LF) --> quit(Type), [!].
```

The session termination requests are determined by the following:

```
quit(quit) --> [bye].
quit(quit) --> [done].
quit(quit) --> [exit].
quit(quit) --> [finished].
quit(quit) --> [halt].
quit(quit) --> [quit].
quit(quit) --> [terminate].
quit(quit) --> [stop].
```

Type: query (the database)

```
interrogatory( LogicalForm )--> query( LogicalForm ), [?].
interrogatory( LogicalForm )--> inv_sentence( LogicalForm, _ ), [?].
```

While queries may be recognized as sentences ending with a question mark, questions come in several forms. The second rule determines if the input is a question and formulates query for the database. Questions.

- who paints?
- who does the dog like?
- does the dog like the cat who paints?
- is ---?

Wh_pronoun ***is*** *determiner* [*binaryPredicate* ***of*** *Y* | *unaryPredicate*] ***?***

```
query( S => answer(X)   ) --> wh_pronoun, db( S, X ).
```

Wh_pronoun verbPharase ***?*** -- e.g. Who bought the picture?

```
query( S => answer(X)   ) --> wh_pronoun,
                              verb_phrase( Number, finite, X^S,
nogap ).
```

Wh_pronoun aux nounPhrase verbPhrase ***?*** -- Who did the dog bite?

```
query( S => answer(X)   ) --> wh_pronoun,
                              inv_sentence( S, gap(np, X) ).
```

Aux nounPhrase verbPhrase ***?*** -- Did the dog bite John?

```
query( S => answer(yes) ) --> inv_sentence( S, nogap ).
```

Is/Are *nounPhrase nounPhrase****?*** -- e.g., Is the dog a brown dog?

```
query( S => answer(yes) ) -->
   (([is], {Number=singular});([are],{Number=plural  })),!,
     noun_phrase( Number, (X^S0)^S, nogap ),
     noun_phrase( Number, (X^true)^exists(X,S0&true), nogap ).
```

Inverted Sentences: eg. does john like mary? or Did the dog bite the mailman ?

```
inv_sentence( S, GapInfo ) --> aux( finite/Form, VP1^VP2 ),
                               noun_phrase( Number, VP2^S, nogap ),
                               verb_phrase( Number, Form, VP1, GapInfo
).
```

Type: assertions (data to be added to the database)

The third rule determines if the input is a declarative sentence which then is added to the database.

Sentence types:

1. Declarative: the dog likes the cat who paints

Grammar:

1. Third person/singular and plural/present tense
2. Number agreement between noun phrase and verb phrase
3. Transitive and intransitive verbs
4. The determiners 'a' and 'every'.
5. Relative clauses

Declarative Sentences

Two sentence forms recognized by the db predicate:

X is the _ of Y
X is a _

Where the blanks are binary and unary predicates respectively. The standard declarative sentence has the form, noun phrase followed by a verb phrase.

```
declarative(LF, nogap) --> proper_noun(PN), db(LF, PN), [.].
declarative(LF, GapInfo ) -->
   noun_phrase(Number, VP^LF, nogap),
   verb_phrase(Number, finite, VP, GapInfo), [.].
```

Unimplemented

```
declarative(LF,_) --> [if], indepClause, [then], indepClause.
declarative(LF,_) --> indepClause, [if], indepClause.
declarative(LF,_) --> indepClause, moreIndepClauses.
moreIndepClauses --> corConj, indepClause, moreIndepClauses.
moreIndepClauses.
```

Phrases

Noun Phrases

```
-- Mary
noun_phrase( singular, NP, nogap ) -->
 proper_noun(NP).
-- the dog that bit the mailman
noun_phrase( Number,   NP, nogap ) -->
 determiner( Number, N2^NP ),
```

```
%adjective( X, Adj ),
 noun( Number, N1 ),
 rel_clause( Number1, N1^N2 ).

noun_phrase( Number, (X^S)^S, gap(np, X) ) --> [].
```

Verb Phrases

```
verb_phrase( Number, Form, Subject^LF, GapInfo ) -->
    tv( Number, Form, Subject^Object^VSO ),
    noun_phrase( Number1, Object^VSO^LF, GapInfo ).

verb_phrase( Number, Form, VP, nogap ) -->
    iv( Number, Form, VP ).

verb_phrase( Number, Form1, VP2, GapInfo ) -->
    aux( Form1/Form2, VP1^VP2 ),
    verb_phrase( Number, Form2, VP1, GapInfo ).

verb_phrase( Number, Form1, VP2, GapInfo ) -->
    rov( Form1/Form2, NP^VP1^VP2 ),
    noun_phrase( Number, NP, GapInfo ),
    verb_phrase( Number, Form2, VP1, nogap ).

verb_phrase( Number, Form1, VP2, GapInfo ) -->
    rov( Form1/Form2, NP^VP1^VP2 ),
    noun_phrase( Number, NP, nogap ),
    verb_phrase( Number, Form1, VP1, GapInfo ).

verb_phrase( Number, finite, X^S, GapInfo ) -->
    (([is],{Number=singular});([are],{Number=plural})),
    noun_phrase( Number, (X^P)^exists(X,S&P), GapInfo ).
```

X ***is*** *determiner binaryPredicate* ***of*** *Y*

```
db( LF, X ) --> [is], determiner( _,_ ), [BinaryPredicate, of, Y],
 { binary_predicate(BinaryPredicate), LF =..[BinaryPredicate,X,Y]}.
```

X ***is*** *determiner unaryPredicate*

```
db( LF, X ) --> [is], determiner( _,_ ), [UnaryPredicate],
 { unary_predicate(UnaryPredicate), LF =..[UnaryPredicate,X]}.
```

Clauses

Relative Clauses

```
rel_clause( Number, (X^S1)^(X^(S1&S2)) ) -->
   rel_pronoun, verb_phrase( Number, finite, X^S2, nogap ).

rel_clause( Number, (X^S1)^(X^(S1&S2)) ) -->
   rel_pronoun, sentence( S2, gap(np, X) ).

rel_clause( Number, N^N ) --> [].
```

Terminals

conjunction ::= *coordinateConjunction*
| *subordinateConjunction*

coordinateConjunction ::= and but or for nor so yet
-- connects grammatically same |
correlativeConjuction

correlativeConjuction ::=

both *A* and *B*
either *A* or *B*
neither *A* nor *B*
whether *A* or *B*
not only A B
A but also *B*

subordinateConjunction ::=
-- connects a dependent clause to a main clause.

after	even though	though
although	how	unless
as	if	until
as if	in order that	when
as long as	since	whenever
as though	so than	where
because	than	wherever
before	that	while

preposition ::=
-- connects noun or nounphrase to the part...

about	by	on
above	down	on account of
according to	due to	out
across	during	out of
after	for	since
against	from	through
among	in	to
around	in front of	toward
at	in regard to	under
because of	into	until
before	like	up
behind	near	with
below	of	without
between	off	with respect to

noun ::= *properNoun* --
-- name of someone or something
nounPhrase -- e.g.. baseball stadium
possessiveNoun
gerundPhrase

pronoun ::= *personalPronoun*
-- substitute for a noun or noun phrase
| *possessiveNoun*
| *demonstrativePronoun*
| *indefinitePronoun*
| *relativePronoun*
| *interrogativePronoun*

indefinitePronoun ::=

all	everybody	none
another	everything	nothing
any	many	one, two etc.
anyone	more	other
both	most	several
either	neither	some
enough	nobody	something

personalPronoun ::=

Singular	**Subject**	**Object**	**Possessive**
First person	I	me	my, mine
Second person	you	you	your, yours
Third person	he, she, it	him, her, it	his, her, hers, its
Plural			
First person	we	us	our, ours
Second person	you	you	your, yours
Third person	they	them	their, theirs

demonstrativePronoun - refer to particular people or things	::=	this that these those
relativePronoun - introduce relative clauses	::=	who whose whom that which what
interrogativePronoun - relative pronoun appearing at the beginning of a sentence.	::=	who whose whom which what
verb -- links subject to noun, pronoun or adjective	::= | |	*transitiveVerb* *intransitiveVerb* *participle*
adjective -- modifies noun or pronoun	::=	
adverb -- modifies verb, adjective, or adverb	::= |	*relativeAdverb* *conjunctiveAdverb*
relativeAdverb	::=	how when where why whenever wherever
conjunctiveAdverb	::=	therefore accordingly besides furthermore instead meanwhile nevertheless

NOUNS pronouns nouns proper nouns person: first---singular, plural I, we second--singular, plural you, you third---singular, plural he, they VERBS transitive intransitive inflextional forms: nonfinite: infinitive, present pariticiple, past participle to take, taking taken finite: person--first or third, number--singular or plural tense--present or past finite(pers3,singular,pres) -s or -es finite(_, _, pres) infinitive form finite(_, _, past) tense: past, present, future, mood: voice: active, passive RELATIVE CLAUSES

Determiners

The determiners correspond to quantifiers. Existential quantification requires a conjunction - exists(X, P(X) & Q(X)). Universal quantification requires implication - all(X, P(X) => Q(X)).

```
determiner( Number, LF ) --> [D],
          { det( D, Number, Type ), detLF( Type, LF ) }.

detLF(exists, (X^S1)^(X^S2)^exists(X, S1 &  S2 )).
detLF(all,    (X^S1)^(X^S2)^all(   X, S1 => S2 )).
```

Nouns

```
noun( singular, X^LF ) --> [Noun], { noun( Noun, _ ),
```

```
    LF =..[Noun,X] }.
noun( plural,   X^LF ) --> [Plural], { noun( Noun, Plural ),
    LF =..[Noun,X] }.
```

Proper Nouns

```
proper_noun( (PN^S)^S ) --> [PN], { proper_noun( PN ) }.
```

Pronouns: Relative & Interrogatory

```
rel_pronoun --> [RelPronoun],  { rel_pronoun( RelPronoun )}.
wh_pronoun  --> [WhPronoun],   { wh_pronoun( WhPronoun ) }.
```

Conjunctions

Coordinating conjunctions connect phrases or clauses that are grammatically the same.
WARNING: Not implemented

```
coorConj --> [CC], { coordinateConjunction(CC, LOp) }.
```

Transitive Verbs

```
tv( Number, Form, S^O^v(S,O)).
tv( plural,   Form, LF ) -->
  [TV], { tv(TV,_,_,_,_,Form), tvLF(Form,LF) }.
tv( singular, Form, LF ) -->
  [TV], { tv(_,TV,_,_,_,Form), tvLF(Form,LF) }.
tv( N, nonfinite, LF ) -->
 [TV], { tv(TV,_,_,_,_,Form), tvLF(Form,LF) }.
tv( N,    finite, LF ) -->
 [TV], { tv(_,TV,_,_,_,Form), tvLF(Form,LF) }.
tv( N,    finite, LF ) -->
 [TV], { tv(_,_,TV,_,_,Form), tvLF(Form,LF) }.
tv( N, past_part, LF ) -->
 [TV], { tv(_,_,_,TV,_,Form), tvLF(Form,LF) }.
tv( N, pres_part, LF ) -->
 [TV], { tv(_,_,_,_,TV,Form), tvLF(Form,LF) }.

tvLF( Verb, Subject^Object^VSO ) :-
  VSO =..[Verb,Subject,Object].
```

Adjectives

```
adjective( X, LF ) --> [A],
          { adjective( A ), adjLF( A, LF ) }.

adjLF(A,X,LF) :- LF =..[A,X].
```

Auxillaries

```
aux( Form, LF ) --> [Aux], { aux(Aux, Form, LF ) }.
```

Intransitive Verbs

```
iv( Number, Form, VP ).
iv( plural,   Form, LF ) --> [IV], { iv(IV,_,_,_,_,Form),
ivLF(Form,LF) }.
iv( singular, Form, LF ) --> [IV], { iv(_,IV,_,_,_,Form),
ivLF(Form,LF) }.

iv( N, nonfinite, LF ) --> [IV], { iv(IV,_,_,_,_,Form),
ivLF(Form,LF) }.
iv( N,    finite, LF ) --> [IV], { iv(_,IV,_,_,_,Form),
ivLF(Form,LF) }.
iv( N,    finite, LF ) --> [IV], { iv(_,_,IV,_,_,Form),
ivLF(Form,LF) }.
iv( N, past_part, LF ) --> [IV], { iv(_,_,_,IV,_,Form),
ivLF(Form,LF) }.
iv( N, pres_part, LF ) --> [IV], { iv(_,_,_,_,IV,Form),
ivLF(Form,LF) }.

ivLF( F, X^LF ) :- LF =..[F,X].

rov( nonfinite/Requires, LF ) --> [ROV],
 { rov(ROV,_,_,_,_,Form,Requires),rovLF( Form, LF ) }.
rov(    finite/Requires, LF ) --> [ROV],
 { rov(_,ROV,_,_,_,Form,Requires), rovLF( Form, LF ) }.
rov(    finite/Requires, LF ) --> [ROV],
 { rov(_,_,ROV,_,_,Form,Requires), rovLF( Form, LF ) }.
rov( past_part/Requires, LF ) --> [ROV],
 { rov(_,_,_,ROV,_,Form,Requires), rovLF( Form, LF ) }.
rov( pres_part/Requires, LF ) --> [ROV],
 { rov(_,_,_,_,ROV,Form,Requires), rovLF( Form, LF ) }.

rovLF( Form, ((X^LF)^S)^(X^Comp)^Y^S ) :- LF =..[Form,Y,X,Comp].
```

Jottings

John loves Mary.	loves(John, Mary)
The clock runs.	exists(X, clock(X) /\ run(X))
The dog chased the cat.	exists(d, dog(d) /\ exists(c, cat(c) /\ chase(dog,cat)))
The big red dog chased the small cat.	exists(d, big(d) /\ red(d) /\ dog(d) /\ exists(c, small(c) /\ cat(c) /\ chase(dog,cat))
if John loves Mary, then	

Need to be able to handle tense.

Dictionary

Author: Anthony Aaby
Document status: unfinished
History: Current document initiated April 1999, Initial coding c. 1990.

Dictionary Interface

Determiners

```
determiner( Number, LF ) --> [D],
          { det( D, Number, Type ), detLF( Type, LF ) }.

detLF(exists, (X^S1)^(X^S2)^exists(X, S1  & S2 )).
detLF(all   , (X^S1)^(X^S2)^   all(X, S1 => S2 )).
```

Adjectives

```
adjective( X, LF ) --> [A],
          { adjective( A ), adjLF( A, LF ) }.

adjLF(A,X,LF) :- LF =..[A,X].
```

Nouns

```
noun( singular, X^LF ) --> [Noun], { noun( Noun, _      ),
    LF =..[Noun,X] }.
noun( plural,   X^LF ) --> [Noun], { noun( Noun, Plural ),
    LF =..[Noun,X] }.
```

Proper Nouns

```
proper_noun( (PN^S)^S ) --> [PN],  { proper_noun( PN ) }.
```

Pronouns: Relative & Interrogatory

```
rel_pronoun --> [RelPronoun],  { rel_pronoun( RelPronoun )}.
wh_pronoun  --> [WhPronoun],   { wh_pronoun(  WhPronoun ) }.
```

Auxillaries

```
aux( Form, LF ) --> [Aux], { aux(Aux, Form, LF ) }.
```

Transitive Verbs

```
tv( Number, Form, VP ).
tv( plural,   Form, LF ) --> [TV], { tv(TV,_,_,_,_,Form),
tvLF(Form,LF) }.
tv( singular, Form, LF ) --> [TV], { tv(_,TV,_,_,_,Form),
tvLF(Form,LF) }.

tv( N, nonfinite, LF ) --> [TV], { tv(TV,_,_,_,_,Form),
tvLF(Form,LF) }.
tv( N,    finite, LF ) --> [TV], { tv(_,TV,_,_,_,Form),
tvLF(Form,LF) }.
tv( N,    finite, LF ) --> [TV], { tv(_,_,TV,_,_,Form),
tvLF(Form,LF) }.
tv( N, past_part, LF ) --> [TV], { tv(_,_,_,TV,_,Form),
tvLF(Form,LF) }.
tv( N, pres_part, LF ) --> [TV], { tv(_,_,_,_,TV,Form),
tvLF(Form,LF) }.

tvLF( Form, X^Y^LF ) :- LF =..[Form,X,Y].
```

Intransitive Verbs

```
iv( Number, Form, VP ).
iv( plural,   Form, LF ) --> [IV], { iv(IV,_,_,_,_,Form),
ivLF(Form,LF) }.
iv( singular, Form, LF ) --> [IV], { iv(_,IV,_,_,_,Form),
ivLF(Form,LF) }.

iv( N, nonfinite, LF ) --> [IV], { iv(IV,_,_,_,_,Form),
ivLF(Form,LF) }.
iv( N,    finite, LF ) --> [IV], { iv(_,IV,_,_,_,Form),
ivLF(Form,LF) }.
iv( N,    finite, LF ) --> [IV], { iv(_,_,IV,_,_,Form),
ivLF(Form,LF) }.
iv( N, past_part, LF ) --> [IV], { iv(_,_,_,IV,_,Form),
ivLF(Form,LF) }.
iv( N, pres_part, LF ) --> [IV], { iv(_,_,_,_,IV,Form),
ivLF(Form,LF) }.
```

```
ivLF( F, X^LF ) :- LF =..[F,X].

rov( nonfinite/Requires, LF ) --> [ROV], {
rov(ROV,_,_,_,_,Form,Requires),
                                        rov( Form, LF ) }.
rov(    finite/Requires, LF ) --> [ROV], {
rov(_,ROV,_,_,_,Form,Requires),
                                        rov( Form, LF ) }.
rov(    finite/Requires, LF ) --> [ROV], {
rov(_,_,ROV,_,_,Form,Requires),
                                        rov( Form, LF ) }.
rov( past_part/Requires, LF ) --> [ROV], {
rov(_,_,_,ROV,_,Form,Requires),
                                        rov( Form, LF ) }.
rov( pres_part/Requires, LF ) --> [ROV], {
rov(_,_,_,_,ROV,Form,Requires),
                                        rov( Form, LF ) }.

rov( Form, ((X^LF)^S)^(X^Comp)^Y^S ) :- LF =..[Form,Y,X,Comp].
```

Conjuctions

Coordinating conjunctions connect grammatically same
WARNING: Not implemented

coorConj --> [CC], { coordinateConjunction(CC, LOp) }.

Dictionary

A *noun* is a name for someone or something, can be particular or general and is often proceeded by a *determiner*. *Pronouns* are substitutes for nouns or noun phrases.

Determiners

Determiners proceed nouns and appear in both singular and plural forms and correspond to the logical quantifiers. The following predicates classify the determiners.

det (*word, number, quantifier*).

```
det( a,     singular, exists).
det( an,    singular, exists).
det( this,  singular, exists).
det( the,   singular, exists).
```

```
det( some,  Number,   exists).
det( all,   plural,   all   ).
det( any,   singular, all   ).
det( the,   plural,   all   ).
det( each,  singular, all   ).
det( every, singular, all   ).
```

The preceeding does not distinguish between definite (*the*) and indefinite (*a/an*) articles as in the following sentence.

> One day, *a* child met *a* dog and *the* child immediately trusted *the* dog and went up to it.

Definite articles: the, this (these), that (those) - may be preceeded by *all*
Indefinite articles: a, an, any, each, either, neither, every, no one, somewhat, whatever, which, whichever, many a , such a, what a - may not be preceeded by all.

Pronouns

The *relative pronoun*s introduce relative phrases

```
rel_pronoun( that  ).
rel_pronoun( what  ).
rel_pronoun( who   ).
rel_pronoun( whom  ).
rel_pronoun( whose ).
rel_pronoun( which ).
```

An important subclass of relative pronouns, the `wh_pronoun`s, are used at the beginning of an interrogatory sentence.

```
wh_pronoun( what ).
wh_pronoun( who  ).
wh_pronoun( whom ).
wh_pronoun( whose ).
wh_pronoun( which ).
```

Proper Nouns

The proper nouns are

```
proper_noun( john  ).
proper_noun( annie ).
proper_noun( monet ).
```

Nouns

noun (singular, plural)

```
noun( baby,   babies  ).
noun( boy,    boys    ).
noun( cat,    cats    ).
noun( father, fathers ).
noun( family, families).
noun( female, females ).
noun( girl,   girls   ).
noun( man,    men     ).
noun( male,   males   ).
noun( mouse,  mice    ).
noun( woman,  women   ).
```

Predicates/Adjectives

```
unary_predicate(male).
unary_predicate(female).
binary_predicate(father).
binary_predicate(mother).
```

adjective (*word*)

```
adjective( big   ).
adjective( small ).
adjective( red   ).
```

Verbs

Verb are action words in a sentence. They may link the subject to a noun, pronoun or adjective. They are classified as transitive or intransitive. Transitive verbs take an object while intransitive verbs do not.

% person/number/tense
% Inflectional forms:
% V Vs Ved Ven Ving
% infinitive finite finite, past part, pres part, LF
% 3rd pers 3rd pers
% plural singular
% present present

inflectional forms

Nonfinite verbs	**infinitive**	**present participle**	**past participle**
example	to take	taking	taken

Finite verbs	voice	Person	Number	Tense
	active	first	singular	past
	passive	third	plural	present
				future

In the following, the verb entries correspond to:

1. The nonfinite form of the verb (the infinitive -- to *verb*)
2. The third person singular form: He/She *verb*
3. The past tense: He/She *verb-past*
4. The past participle: He/She has *verb*
5. The present participle: He/She is *verb*
6. The logical form used in the database: usually third person singular

Intransitive verbs

iv (*infinitive, thirdPersonSingular, PastTense, PastParticiple, PresentParticiple, LogicalForm*).

```
iv( come,     comes,    came,     come,     comming,    comes
).
iv( dance,    dances,   danced,   danced,   dancing,    dances
).
iv( go,       goes,     went,     gone,     going,      goes
).
iv( halt,     halts,    halted,   halted,   halting,    halts
).
iv( paint,    paints,   painted,  painted,  painting,   paints
).
iv( sleep,    sleeps,   slept,    slept,    sleeping,   sleeps
).
iv( walk,     walks,    walked,   walked,   walking,    walks
).
iv( rest,     rests,    rested,   rested,   resting,    rests
).
```

Transitive verbs

tv (*infinitive, thirdPersonSingular, PastTense, PastParticiple, PresentParticiple, LogicalForm*).

```
tv( adimire,  admires,  admired,   admired,   admiring,   admires
).
tv( are,      is,       was,       was,       are,        isa
).
tv( buy,      buys,     bought,    bought,    buying,     buys
).
tv( concern,  concerns, concerned, concerned, concerning, concerns
).
```

```
tv( eat,      eats,     ate,       ate,       eating,     eats
).
tv( hate,     hates,    hated,     hated,     hating,     hates
).
tv( have,     has,      had,       had,       having,     has
).
tv( give,     gives,    gave,      gave,      giving,     gives
).
tv( like,     likes,    liked,     liked,     liking,     likes
).
tv( meet,     meets,    met,       met,       meeting,    meets
).
tv( run,      runs,     ran,       run,       running,    runs
).
tv( scare,    scares,   scared,    scared,    scaring,    scares
).
tv( see,      sees,     seen,      seen,      seeing,     sees
).
%   show
%   take
%   tell
tv( write,    writes,   wrote,     written,   writing,    writes
).

rov( want,    wants,    wanted,    wanted,    wanting,    wants,
infinitive ).
```

Auxiliary verbs

Those commented out have not been verified.

aux(form, a/b, VP^VP)

```
aux( be,    nonfinite  / pres_part, VP^VP ).
aux( been,  past_part  / pres_part, VP^VP ).
%aux( can,   finite     / nonfinite, VP^VP ).
aux( could, finite     / nonfinite, VP^VP ).
%aux( is,   finite     / nonfinite, VP^VP ).
aux( did,   finite     / nonfinite, VP^VP ).
aux( does,  finite     / nonfinite, VP^VP ).
aux( has,   finite     / past_part, VP^VP ).
aux( have,  finite     / past_part, VP^VP ).
%aux( may,  finite     / past_part, VP^VP ).
%aux( might,  finite     / past_part, VP^VP ).
%aux( shall,  finite     / past_part, VP^VP ).
%aux( should,  finite    / past_part, VP^VP ).
```

```
aux( to,    infinitive / nonfinite, VP^VP ).
%aux( would,  finite     / past_part, VP^VP ).
```

Conjunctions

coordinateConjunction (conjunction, logicalOperator)

```
coordinateConjunction(and, and).
coordinateConjunction(but, and).
coordinateConjunction(or, or)
%coordinateConjunction(for, X).
%coordinateConjunction(nor, X).
coordinateConjunction(so, and).
coordinateConjunction(yet, and).
```

A Scanner for English Sentences

Author: Anthony Aaby
Document status: unfinished
History: Current document initiated April 1999, Initial coding c. 1990.

`read_sentence(Sentence, [])` - (scanner)

Sentences are sequences of words terminated with either a period, exclamation point, or a question mark. Punctuation symbols are considered words. The following code is based on standard scanner design.

```
read_sentence --> {get0(C)}, word(C,W,C1), rest_sent(C1,W).
```

Given the next character and the previous word, read the rest of the sentence

```
rest_sent(C,W) --> {lastword(W)}.  % empty
rest_sent(C,_) --> word(C,W,C1), rest_sent(C1,W).
```

word(LookAheadChar, Word, NextLookAheadChar)

```
word(C,W,C1) --> {single_character(C),!,name(W,[C]), get0(C1)},
[W].  % !,.:;?
word(C,W,C2) --> {in_word(C,Cp), get0(C1), rest_word(C1,Cs,C2),
                  name(W,[Cp|Cs])},[W].
word(C,W,C2) --> {get0(C1)}, word(C1,W,C2). % consumes blanks
```

These words terminate a sentence.

lastword(Char)

```
lastword('.').
lastword('!').
lastword('?').
```

This reads the rest of the word plus the next character.

rest_word(LookAheadChar, Chars, Following Char)

```
rest_word(C,[Cp|Cs],C2) :- in_word(C,Cp), get0(C1),
rest_word(C1,Cs,C2).
rest_word(C,[],C).
```

single_character(Char)

```
single_character(33). % !
single_character(44). % ,
single_character(46). % .
single_character(58). % :
single_character(59). % ;
single_character(63). % ?
```

case(UpperCase, LowerCase)

```
case(C,C) :- C > 96, C < 123.               % a,b,...,z
case(C,L) :- C > 64, C < 91, L is C + 32.   % A,B,...,Z
case(C,C) :- C > 47, C < 58.                % 0,1,...,9
case(39,39).                                % '
case(45,45).                                % -
```

The Clausifier (FOL to Clause Logic)

translate(FOL, CL)

Operator Declarations

The logical operators listed in binding strength (highest to lowest).

```
:- op(  30, fx, ~).     % Not
:- op( 100, xfy, #).    % Or
:- op( 100, xfy, &).    % And
:- op( 150, xfy, =>).   % If
:- op( 150, xfy, <=>).  % Iff
%  all(X, P)              Universal Quantification
%  exixts(X, P)           Existential Quantification
```

Translate(FOL-formula, Clausal-formula)

1. Replace conditional and biconditionals with equivalent formulas
2. Negation normal form: negations are moved inwards.
3. Remove existential quantifiers and replace existential variables with Skolem functions.
4. Remove universal quantifers as the universal variables are unique.
5. Rearrange formula into conjunctive normal form.
6. Translate CNF to clauses.

```
translate( F, Clauses ) :- impl_out( F,  F1 ),
                           neg_in(   F1, F2 ),
                           skolem(   F2, F3, [] ),
                           univ_out( F3, F4 ),
                           conjnf(  F4, F5 ), write( F5), nl,
                           clausify( F5, Clauses, []).
```

Rules for Removing Conditional and Biconditional operators.

Replace implications (A => B) with (~A # B) and biconditionals (A<=>B) with (A&B)#(~A&~B).

impl_out(Formula, ImplicationFreeFormula)

```
impl_out( (P => Q), (~ P1 # Q1) ) :- !, impl_out( P, P1 ), impl_out(
```

```
Q, Q1 ).
impl_out( (P <=> Q), ((P1 & Q1) # (~P1 & ~Q1)) ):- !, impl_out( P,
P1 ),
                                                    impl_out( Q,
Q1 ).
impl_out( ~P, ~P1 ) :- !, impl_out( P, P1 ).
impl_out( all(X,P),    all(X,P1) ) :- !, impl_out( P, P1 ).
impl_out( exists(X,P), exists(X,P1) ) :- !, impl_out( P, P1 ).
impl_out( (P & Q), (P1 & Q1) ) :- !, impl_out( P, P1 ), impl_out( Q,
Q1 ).
impl_out( (P # Q), (P1 # Q1) ) :- !, impl_out( P, P1 ), impl_out( Q,
Q1 ).
impl_out( P, P ).
```

Rules For Negation Normal Form: `neg_in( Formula, NegationNormalForm )`

In negation normal form, negations only appear just before atomic formulas.

```
neg_in(          ~~P,           P1 ):- !, neg_in(  P, P1 ).
neg_in(    ~all(X,P), exists(X,P1) ):- !, neg_in( ~P, P1 ).
neg_in( ~exists(X,P),    all(X,P1) ):- !, neg_in( ~P, P1 ).
neg_in(     ~(P & Q),    (P1 # Q1) ):- !, neg_in( ~P, P1 ),
                                          neg_in( ~Q, Q1 ).
neg_in(     ~(P # Q),    (P1 & Q1) ):- !, neg_in( ~P, P1 ),
                                          neg_in( ~Q, Q1 ).
neg_in(     all(X,P),    all(X,P1) ):- !, neg_in(  P, P1 ).
neg_in(  exists(X,P), exists(X,P1) ):- !, neg_in(  P, P1 ).
neg_in(      (P & Q),    (P1 & Q1) ):- !, neg_in(  P, P1 ),
                                          neg_in(  Q, Q1 ).
neg_in(      (P # Q),    (P1 # Q1) ):- !, neg_in(  P, P1 ),
                                          neg_in(  Q, Q1 ).
neg_in(            P,            P ).% Literal formula
```

Replace Existential Variables with Skolem Functions

Skolem functions are unique functions of the free variables in a formula.

```
skolem( all(X,P),    all(X,P1), Vars ) :- !, skolem( P, P1, [X|Vars]
).
skolem( exists(X,P), P2,        Vars ) :- !, gensym( f, F ), Sk
=..[F|Vars],
                                          subst( X, Sk, P, P1 ),
                                          skolem( P1, P2, Vars ).
skolem( (P & Q),     (P1 & Q1), Vars ) :- !, skolem( P, P1, Vars ),
                                             skolem( Q, Q1, Vars ).
skolem( (P # Q),     (P1 # Q1), Vars ) :- !, skolem( P, P1, Vars ),
```

```
                                                skolem( Q, Q1, Vars ).
skolem( P,           P,         Vars ).

subst( X, Sk, all(Y,P),    all(Y,P1)    ) :- !, subst( X, Sk, P, P1
).
subst( X, Sk, exists(Y,P), exists(Y,P1) ) :- !, subst( X, Sk, P, P1
).
subst( X, Sk, (P & Q),     (P1 & Q1)    ) :- !, subst( X, Sk, P, P1
),
                                                subst( X, Sk, Q, Q1
).
subst( X, Sk, (P # Q),     (P1 # Q1)    ) :- !, subst( X, Sk, P, P1
),
                                                subst( X, Sk, Q, Q1
).
subst( X, Sk, P,           P1           ) :- functor(P,F,N),
                                                subst1( X, Sk, P, N, P1
).

subst1( X, Sk, P, 0, P  ).
subst1( X, Sk, P, N, P1 ) :- N > 0, P =..[F|Args], subst2( X, Sk,
Args, ArgS ),
                             P1 =..[F|ArgS].

subst2( X, Sk, [], [] ).
subst2( X, Sk, [A|As], [Sk|AS] ) :- X == A, !, subst2( X, Sk, As,
AS).
subst2( X, Sk, [A|As], [A|AS]  ) :- var(A), !, subst2( X, Sk, As,
AS).
subst2( X, Sk, [A|As], [Ap|AS] ) :- subst( X, Sk, A, Ap ),
                                    subst2( X, Sk, As, AS).
```

Remove Universal Quantifiers

Universal quantifiers may be removed as there are no existential quantifiers and universally quantified variables are unique.

```
univ_out( all(X,P),       P1  ) :- !, univ_out( P, P1 ).
univ_out(  (P & Q), (P1 & Q1) ) :- !, univ_out( P, P1 ), univ_out(
Q, Q1 ).
univ_out(  (P # Q), (P1 # Q1) ) :- !, univ_out( P, P1 ), univ_out(
Q, Q1 ).
univ_out(        P,         P ).
```

Conjunctive Normal Form (CNF)

In conjunctive normal form (CNF), conjunctions are the outermost connective.

```
conjnf( (P # Q), R         ) :- !, conjnf( P, P1 ), conjnf( Q, Q1 ),
                                   conjnf1( (P1 # Q1), R ).
conjnf( (P & Q), (P1 & Q1) ) :- !, conjnf( P, P1 ), conjnf( Q, Q1 ).
conjnf( P,       P         ).

conjnf1( ((P & Q) # R),  (P1 & Q1) ) :- !, conjnf1( (P # R), P1),
                                           conjnf1( (Q # R), Q1 ).
conjnf1( (P # (Q & R)),  (P1 & Q1) ) :- !, conjnf1( (P # Q), P1),
                                           conjnf1( (P # R), Q1 ).
conjnf1( P,              P         ).
```

Clausify - converts CNF to clauses

```
clausify( (P & Q), C1, C2 ) :- !, clausify( P, C1, C3 ),
                                  clausify( Q, C3, C2 ).
clausify( P, [cl(A,B)|Cs], Cs ) :- inclause( P, A, [], B, [] ), !.
clausify( _, C, C ).

inclause( (P # Q), A, A1, B, B1 ) :- !, inclause( P, A2, A1, B2, B1
),
                                       inclause( Q, A,  A2, B,  B2
).

inclause( ~P, A,  A, B1, B ) :- !, notin( P, A ), putin( P, B, B1 ).
inclause( P,  A1, A, B,  B ) :- !, notin( P, B ), putin( P, A, A1 ).

notin(X,[Y|_]) :- X==Y, !, fail.
notin(X,[_|Y]) :- !,notin(X,Y).
notin(X,[]).

putin(X,[],   [X]   ) :- !.
putin(X,[Y|L],[Y|L] ) :- X == Y,!.
putin(X,[Y|L],[Y|L1]) :- putin(X,L,L1).
```

Automated Reasoning

- Available Systems (anl)
- Results
- Analytic Tableaux
- Research opportunities
- Logic
    - First order logic
- Resources
    - LeanTaP
    - ileanTAP: an intuitionistic theorem prover
    - ModLeanTAP: Propositional Modal Logics
- Implementations
    - Logics
    - Code
- Peano

The Logical Foundations of Computer Science and Mathematics

Document status: under development.

1. Overview	
2. Classical logic	Appropriate for static systems
3. Modal logics	Belief, knowledge, temporal progression
4. Multivalued logics	Uncertainty, fuzzy membership systems
5. Constructive logic	Logic of constructive systems
6. Nonmonotonic logic	
7. Algorithmic logic	The logic of functional programming
8. The Lambda Calculus	

Tar file of this material: logic.tgz (may not be upto date)

Supplementary material

- General setting for incompleteness
- Analytic proof style
- Analytic tableaux
- Axiomatic method
- Free variables
- Hilbert style proofs
- Horn clause logic
- Modal logic - tableau rules
- Natural Deduction
- Normal forms and Skolem functions
- Prolog technology for theorem proving
- Resolution
- Semantics
- Sequent (Gentzen) systems
- Substitution
- Syntax
- Temporal logic
- Truth tables

- Unification
- References
  - Peter Suber's A Bibliography of Non-Standard Logics

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Automated Reasoning

Paper

Code

Systems Research Group

The Systems Research Group (SRG) ...

Announcement

Computer Systems and System Software

System simulation environment

- Assembly language, assemblers, linkers, and loaders: Universal assembler
- Architecture: Universal interpreter
- Architecture & networking
- Compilers
- Operating Systems and Networking
- Database

Architecture

- Construct a simulator for an alternative architectures
    - P-machine
    - SECD-machine
    - Lambda machine
    - Logic Machine
- VLSI -- implement a lambda calculus machine (Aaby, Aamodt)

Data Base

Human Computer Interation

- Construct a Web based interface to a computer, scanner, printer, modem to produce a copier and fax machine.

Operating Systems

- SimOS (Stanford University)
- OS tool kit (University of Utah)
- CC++
- Design and implement an OS or portions an OS.
- Add/Modify features of an OS or Network
- Do something with Unix(NetBSD, Linux, etc), Mach, Ameoba, OS2, Windows NT
- Collect, modify, develop tools for monitoring, analyzing and/or simulating a network.
- Develop sys admin materials for the NT environment
- Develop sys admin materials for networking

Programming Languages

- Design a multiparadigm programming language
- Design and implement a programming language
- Compare language based memory managers (including garbage collectors) to OS memory managers.
- Compilers etc
    - Construct a universal assembly language and assembler. (Aamodt, Aaby)
    - Use ELI to construct a compiler
    - Construct a compiler for ???
    - Develop a hardware (VLSI) lambda calculus interpreter
    - Complete the development of Prolog based compiler writing tools.
    - Port Aaby's Prolog based compiler example to PCN.
    - Construct/assemble supporting routines for a compiler.
    - Translate Lucent Technologies Limbo to Java
    - Construct a compiler to translator SPECS to C++ (Wether & Conway (1996) "A Modest Proposal: C++ Resyntaxed" *ACM SIGPLAN 31:11 Nov 1995 p 74.)*
- Runtime Environment
    - SECD-machine (Lispkit)
    - Lambda machine: a lambda calculus interpreter (parallel)
    - Prolog machine (Prologkit)
- Develop a logic programming language using infinite valued logic.

Systems Research Group

announce the formation of the systems

Universal Assembler

- Report
- Release notes
- Programmer's Guide

Universal Interpreter

Abstract: The univeral is an interpreter for object/binary code which uses user supplied machine definition and table lookup to simulate machine execution.

Object Code

Object code file format: line of space separated integers. Each integer represents a field in an instruction format. See Universal Assembler

Files

simulators

Machine Descriptions

Architecture

1. CPU Project
2. 0 register (Stack Machine)
3. 1 register (Accumulator Machine)
4. n registers (Register Machine)
5. The IAS Computer
6. SPARC

Unified Paradigm Grammar

The goal of this work is

- to design an abstract grammar for those elements that programming languages have in common in particular, for abstraction, generalization, and modules and
- to integrate the grammar with abstract grammars for a variety of programming paradigms.

This work is supports ideas developing in *Introduction to Programming Languages* where abstraction, generalization and computational models are used as unifying concepts for understanding programming languages.

Notation

Figure M.N: **Notation**

Symbols	Meaning	Font	Meaning
N ::= RHS	grammar rule	Standard	grammar symbols
A | B	alternatives	**Bold**	literal terminal
(A)	grouping	*Italic*	nonterminal
[...]	optional		
[...]*	zero or more		
[...]+	one or more		
$item_{subscript}$	subscripts are used to distiguish instances		

The Grammar

Figure N.M: **Unified Paradigm Grammar**

Module:

module	::=	*name* [**library** | **adt** | **class**] [**implementation** | **definition**] **module** [**extends** *name*] *environment*
environment	::=	$declaration^{+}$

declaration ::= [**export** | **private** | **protected** | **initial** | **final**] *abstraction*
| **import** *name*

Block: (the abstract is limited to commands and expressions)

block ::= **let** *environment* **in** *abstract* **tel**

Abstraction & invocation

abstraction ::= *name* **is** *abstract* **.**

abstract ::= *name*
| *generic*
| *module*
| *expression*
| *command*
| [**a**] *type*

invocation ::= *name* | *abstract* | *application* | *query*
| *specialization*
| *name*[**(** *arguments* **)**]

signature ::= [*name* **:**]*type* [--> *type*]

Generalization & specialization

generic ::= \ *param*[**,** *param*]$^+$ **.** *abstract*

specializtion ::= **(** *generic arguments* **)**

arguments ::= *value* [**,** *value*]*

param ::= *name* **is a** *abstract*
| [*type*] *identifier* [**,** *identifier*]*
| **var** *identifier* [**,** *identifier*]* of *type*

STUFF

application ::= (*generic* | *name*) [*expression*]*

reference ::=

assignable ::=

Functional Programming: reduction of an exression to a normal form.

expression ::= *constant*
| *variable*
| *name*
| **(** *expression expression* **)**
| \ *param* [**,** *param*]$^+$ **.** *expression*

Logic Programming: deduction that either fails or returns a list of bindings

logic_program ::= *theory query*

theory ::= *clause*$^+$

clause ::= *predicate* **.** | *predicate* **:-** *predicate* [**,** *predicate*]* **.**

predicate ::= *atom* | *atom***(** *term* [**,** *term*]* **)**

term ::= *numeral* | *atom*[**(** *term* [**,** *term*]* **)**] | *variable*

query ::= **?-** *predicate* [**,** *predicate*]* .

Imperative Programming: a sequence of bindings.

command ::= **skip**
| *event*
| $identifer_0$ **,...,** $identifier_n$ **:=** $expression_0$ **,...,** $expression_n$
| **{;** *command* [**,** *command*]* **}**
| **{?** *guard* --> *command* [**,** *guard* --> *command*]*
[**, elsif** *boolean_expression* --> *command*]*[**, else** --> *command*]**}**
| **{*** *guard* --> *command* [**,** *guard* --> *command*] **}**
| **{||** *command* [**,** *command*]* **}**
| *invocation*

guard ::= (*event* | *boolean_expression*)[**,** *boolean_expression*)]*

boolean_expression ::= *value* **?=** *pattern*

pattern ::= *list* | *tuple*

Communication and Event Primitives

event ::= *send*
| *receive*

send ::= **send** *message* **to** *process_identifier* | p**!**e
| **output** *expression*

receive ::= **receive** *message* **from** *process_identifier* | p**?**x
| **input** *variable*

message ::= **<info,** *a***,** *b***>**

Values

constant ::= *atomic* | *structured*

atomic ::= **null** | ***** | *boolean* | *character* | *string* | *number*

structured ::= *range*
| *tuple*[**.***name*]
| *function*
| *name***[***arguments***]**

Exceptions

Threads

Types

type ::= *primitive* | *type_def*

prmitive ::= **Boolean** | **Character** | **String** | **Number**

type_def ::= *enumeration* | *range* | *sum* | *product* | *function*

enumeration ::= **[** *item* [**,** $item_n$]* **]**

range ::= **[** *i* **..** *j* **]**
| **[** i**,** j **..** k **]**

product ::= **(*** [*field_name***:**]*type* [**,** [*field_name***:**]*type*]* **)**

sum ::= **(+** [*tag_name***:**]*type* [**,** [*tag_name***:**]*type*]* **)**

function ::= [**mutable**] *type* **-->** *type*

Class

class ::= **class** [**implementation** | **definition**] **module** *environment*
[**initial** *abstraction*][**final** *abstraction*]

Names i.e. Identifiers

Reserved words, keywords, and user defined names are identifiers which are sequences of alphabetic charactors and digits. The first charactor must be an alphabetic character. Case does (does not) matter.

Functional Programming

A functional program is an expression. The expressions include

- constants
- variables
- function application and
- function abstraction

Constants

Constants include numbers, the boolean values, nil, the arithmetic and relational operators, and other predefined function symbols.

Variables

Variable are identifiers. If the variable is the name of an abstract, then its value is the abstract otherwise its value is undefined.

Function Application

Function application takes the form

$$(\ expression_1 \ expression_2 \ )$$

The result is the reduction of the application to normal form. Reduction to normal form is function

evaluation which if $expression_1$ is a generic then the quantifier is removed from the expression and $expression_2$ is substituted, in the body of $expression_1$, for the quantified variable. If the resulting expression is reduceable, then it is reduced.

Function Abstraction

A function abstraction is in normal form and stands for its self.

Logic Programming

Imperative Programming

An imperative program is a command. The commands include

- Skip Command
- Application Command
- Assignment Command
- Parallel Command
- Sequential Command
- Choice Command
- Iterative Command
- Abstraction
- Invocation

Skip Command

The skip command has the form

```
skip
```

It does nothing.

Application Command

The application command has the form

```
name( actual parameters )
```

The action performed by an application command is determined by its definition.

Assignment Command

The assignment command has the form:

$$identifer_0, \ldots, identifier_n := expression_0, \ldots, expression_n$$

For n>=0. The effect is as if the expressions are evaluated and assigned in parallel with the i^{th} $identifier_i$ assigned the value of the i^{th} $expression_i$. The identifier and expression must be type compatible (matching types).

Parallel Command

The parallel command is of the form:

$$\{|| \ command_0, \ldots, command_n \ \}$$

The programmer may make no assumptions about the degree of parallelism with which the commands execute.

Sequential Command

The sequential command is of the form:

$$\{; \ command_0, \ldots, command_n \ \}$$

The programmer may assume that the commands execute in sequence from left to right with each command terminating before the next begins.

Choice Command

The choice command is nondeterministic and is of the form:

$$\{? \ guard_0 \ \text{-->} \ command_0,$$
$$\ldots,$$
$$guard_n \ \text{-->} \ command_n$$
$$\}$$

The programmer may assume that if no guard evaluates to true, that the command terminates and that if some guard is true, that exactly one of the commands corresponding to a guard that evaluates to true is executed.

Iterative Command

The iterative command is nondeterministic and is of the form:

```
{* guard_0 --> command_0,
   ...,
   guard_n --> command_n
}
```

The programmer may assume that while some guard is true, exactly one of the commands corresponding to a guard that evaluates to true is executed and that if no guard evaluates to true, that the command terminates. The guards are reevaluated after the execution of a command.

Abstraction

Inline abstractions are restricted to

Invocation

Invocations are restricted to direct recursion within an abstraction,

Implementation

The implementation iwill be in Java.

References

Chandy, M. K. & Taylor Stephen
An Introduction to Parallel Programming Jones and Bartlett 1992.
Modula-3
Java

Compiler Construction Tools in Prolog

Project Leader: A. Aaby

Scanning

Context-free grammars

- Introduction
- Context-free grammar representation
- EBNF to Prolog representation
- Left-factoring
- Left-recursion
- First sets
- Follow sets

Recursive descent parsing

- Left-factoring
- Left-recursion
- First sets
- Follow sets
- Parser construction

Table driven parsing

- Top-down
- Bottom-up

Miscellaneous

- The user interface
- A scanner written in Prolog

Last update:

Send comments to: webmaster@cs.wwc.edu

The C Family

This document is under development. Completion anticipated before

Introduction

The C family of languages are expression oriented imperative programming languages. C was designed for systems programming. C++ was designed for simulation and to support object-oriented programming(OOP) thus it includes support for the definition and encapsulation of objects and for

inheritance. Java was designed for embedded systems programming and has evolved into a general purpose programming language.

Notation

In what follows, `fixed width font` is used for the symbols and reserved wors of the languages. Plurals Occuring in the description of syntax refer to *a comma separated list.*

C

A general pupose programmming language originally developed for systems programming.

```
/* Hello C Users */
#include <stdio.h>
main() {
printf("Hello C Users!\n");
}
```

C++

A superset of C that provides *object-oriented programming* (OOP) capabilities.

```
// Sample Hello C++ Users
#nclude <iostream.h>
main() {
cout << "Hello C++ Userss!\n";
}
```

Java

- Object-Oriented
- Multi-threading
- Automatic garbage collection

NO

- structs
- typedefs
- #defines
- pointers
- explicit memory deallocation

Sample Java program

```
// Hello Java Users
import java.io.*
class HelloJavaUsers {
      public static void main(String[] args)
{
    }
}
```

Applet Example

```
// Hello Java Users
import java.applet.Applet;
import java.awt.Graphics;

public class HellloJavaUsers
extends java.applet.Applet {
public void paint (Graphics g)
{
g.drawString ("Hello Java Users!", 25 25);
}
}
```

Keyfeatures

C	**C++**	**Java**
• Procedural language	• O-O *superset* of C classes and object inheritance polymorphism	• fully O-O/C++ *like*
• weakly typed	• strongly typed	• strongly typed compiler rejects references to undefined variables and absence of exception handling
• Multiline comments	• Single line comments	• `HTML documentation via javadoc` extracts /**...*/ comments

• separate compilation	• class libraries and interfaces	
• declarations before code	• declatation anywhere in code	
• macros		
• `typedef`	• function prototypes	

• Preprocessor
`#include`
`#define`
`#if #else`
`#endif`
	• Preprocessor
	`#include`
	`#define`
	`#if #else`
	`#endif`
		• No preprocessor

• Makefiles and library linking	• Makefiles and library linking	• No makefiles or library llinking
• Direct memory access	• Direct memory access	• No direct memory access
• Programmer managed storage	• Improved programmer managed storage `new` *TypeName* and `delete` *Name*	• Automatic storage management
• Pointer arithmetic	• Pointer arithmetic	• No pointer arithmetic
	• Operator overloading	• No operator overloading No multiple inheritance
	• Multiple inheritance	
• Platform independent language	• Platform independent language	• Platform independent code
		• Network ready
		• Dynamic loading and linking
		• Multi-threaded
	• Exception handling	• Exception handling
	• `inline` functions	
	• `enum, struct, union` create new types	
	• reference patameters	
	• stream i/o	• Support for native methods
	• constant variables with const *Type Name = Value;*	• declarations anywhere in code

		• Graphics & multimedia

Lexical Structure

Character set, Comments and Identifiers

Key features

	C	C++	Java
Character Set	ASCII	ASCII	Unicode \uddd - d is a hexadecimal digit
Comments	/* . . . */ may not be nested	// . . . terminates at the end of line	/** . . . */ for HTML documentation
Identifiers	*letters, digits,* and *underscores;* does not begin with a digit	*letters, digits,* and *underscores;* does not begin with a digit	*letters* and *digits;* the first must be a letter
Separators			White space is required to separate tokens that would otherwise constitute a single token

Reserved Words

C			C++			Java		
auto	break	case	adds the following			adds the following		
char	const	continue	asm	catch	class	abstract	boolean	byte
default	do	double	delete	friend	inline	cast	catch	extends
else	enum	extern	new	operator	private	final	finally	future
float	for	goto	protected	public	template	generic	implements	import
if	int	long	this	throw	try	inner	instanceof	interface
register	return	short	virtual			native	null	outer
signed	sizeof	static				package	rest	super
struct	switch	typedef				synchronized	throws	transient
						volatile		

union unsigned void volatile while		but drops the following from C auto enum extern register signed sizeof struct typedef union unsigned and drops the fuollowing from C++ asm delete friend inline template virtual

Separators

Scope Rules

Operators

	C/C++/Java	Semantics
Arithmetic	Unary + - Binary + - * / %	positive, negative add, subtract, multiply, divide, remainder
Assignment	*Name* = Expr *Name* BinaryOp=Expr *Name* ++ *Name* --	*Name* = *Name BinaryOp Expr* *Name* = *Name*+ 1 *Name* = *Name*- 1
Bitwise	unary ~ binary & | ^ *Name*<< *n*, *Name*>> *n* *Name*>>> *n*	negation and, or, xor shift left, shift right *n* bits right shift fill with zeros **(Java)**
Boolean	&& || !	and, or, not
Conditional	*BoolExp* ? Exp_1 : Exp_2	conditional expression
Decrement	--*Name, Name*--	pre and post decrement
Increment	++*Name, Name*++	pre and post increment
Relational	== != > < >= <=	equality, not equals, greater than, less than, greater or equal, less than or equal

String	+	string concatenation

Types and Literals

	C/C++	Java
Void	`void`	`void`
Boolean	NA	NA
Boolean literals	NA	`false true`
Character	`char` `unsigned`	`char`
Character literals	*'Character'*	*'Character'*
String Literals	*"Characters"*	*"Characters"*

Integer	`short int,`
	`short`
	`unsigned`
	`short int,`
	`unsigned`
	`short`
	`int`
	`unsigned`
	`int,`
	`unsigned`
	`long int,`
	`long`
	`unsigned`
	`long int,`
	`unsigned`
	`long`
		`byte` 8-bit
		`short` 16-bit
		`int` 32-bi**t**
		`long` 64-bit
		
		(signed two's complement)

Integer literals	D^+	$0D^+$ - octal 0xD+ - hexadecimal 0xD+ - hexadecimal D+[l] - decimal (long) D+[L] - decimal (long)

Floating point	`float`
	`double`
	`long double`
		`float` 32-bit
		`double` 64-bit

Floatiing point literals	$D^+[.D^+[e[+\|-]D^+]]$ $D^+[.D^+[E[+\|-]D^+]]$ $D^+[.D^+[d[+\|-]D^+]]$ $D^+[.D^+[D[+\|-]D^+]]$	$D^+[.D^+[e[+\|-]D^+]]$ $D^+[.D^+[E[+\|-]D^+]]$ $D^+[.D^+[d[+\|-]D^+]]$ $D^+[.D^+[D[+\|-]D^+]]$
Reference		
Reference literals		`null`

Conversions

Names and Variables

Names

A name is an identifier that has been given meaning in a program declaration and denotes either a package, a type, a field, a group of methods, a formal patameter, alocal variable, or a label.

Avariable is a typed storage location

Program Structure

C	**C++**	**Java**
A program consists of declarations in possibly different files. The files may be separately comiled. Function declarations may not be nested. The fundamental unit of programming is the *function.* The function main() is used as the starting point for execution of the program. External libraries provide input/output. The information the program needs to use these libraries resides in the files `iostrem.h`, `stream.h`, and `stdio.h`.	Adds *Classes*	A program is organized into `packages` that have hierarchical names. Each package consists of a number of compilation units. Drops functions *class*. Class libraries are `imported` No preprocessor

A preprocessor to handles a set of directives, such as the include directive, to convert the program from its preprocessing form to the pure syntax. These directives are introduced by the symbol #.		
Preprocessor directives *Definitions and Declarations*		*Import statements* *Class definition*

Blocks and Statements

Except as described, statements are exeuted in sequence. Statements are executed for their effect, and do not have values.

A block is a sequence of statements inclosed between braces ({}).

Jave requires that variables be clearly initialized befor use.

Declaration Statements

A declaration has the form:

Modifiers Type ListOfIdentifiers;

and may appear at any point in the code. In the following, the *Modifiers* is implicit.

Variables and Pointers

	C	**C++**	**Java**
Modifiers	`const static`	`public private protected static`	`public private protected static synchronized final`
Constants	`#define` *Name Value*	const *Type Name* = *Value*;	static final *Type Name* = *Value*;
Variables	*Type * Names;*	*Type * Names;*	*Type * Names;*
Pointers	*Type * Name*; **Ptr* - value at *Ptr* &*Name* - address of *Name*	*Type * Name*; **Ptr* - value at *Ptr* &*Name* - address of *Name*	none

Arrays, Records, and Unions

	C/C++	**Java**
Array Declaration	*Type Name* $[n_1][n_2]\ldots[n_j]$; size must be specified	*Type* [ ][ ] . . . [ ] *Name*;
Reference	*Name* $[i_1]\ldots[i_n]$	*Name* $[i_1]\ldots[i_n]$
Record Type Definition	`struct` *StructType* { *fields* }; fields are variable definitions	NA
Record Declaration	*StructType StructVariable* ;	NA
Field access	*StructVariable* `.FieldName` *PtrToStruct* -> `FieldName`	NA
Union Type Definition	*union UnionType* { *members*} ; alternatives are variable declarations	NA
Union Declaration	*UnionType UnionVariable* ;	NA
Member Access	*UnionVariable* . *Member*	NA

Subscripts lie in the range of 0 to n_j - 1. An array name by itself is an address, or pointer value, and pointers and arrays are almost identical in terms of how they are used to access memory. A pointer is a variable that takes an address as its value. An array name is a particular fixed address that can be thought of as a constant pointer. Thus pointer arithmetic provides an alternative to array indexing.

```
// a is an array of 100 integers values and p is

an address of an integer

int a[100], *p

...
p = a; p = &a[0];
// these are equivalent assignments as are
p= a + 1; p = &a[1];
```

Functions

	C	**C++**	**Java**
Function protoype	*Type Name* (*Formals*)	*Type Name* (*Formals*)	NA
Formals	The formals are a list of types	The formals are a list of types	NA
Functions type is void for procedures	*Type Name* (*Formals*) *Block*	*Type Name* (*Formals*) *Block*	NA
Formals	list of declarations	list of declarations	NA
Parameters are Call-by-	value	value	NA
Formals reference parameter	List of *Type Name* *Type *Name*	List of *Type Name* *Type &Name*	NA
Actuals reference parameters	List of *Expr* *&Name*	List of *Expr* *Name*	NA

Classes and Objects

	C	**C++**	**Java**
Class Definition	NA	class *ClassName* { *fields and methods* } ;	class *ClassName* { *fields and methods* }
Object declaration	NA		*ClassName Name* = new *ClassName*() ;
Fields - State	NA		*Modifiers Type ListOfIdentifiers* ;
Methods -	NA		*Modifiers Ype Name* (*Formals*) *Block*
Parameters are Call-by-	NA	value	value
Formals reference parameter	NA	list of *Type Name* *Type &Name*	list of *Type Name*
Actuals reference parameter	NA	list of *Expr* *Name*	list of *Expr*

Names are a comma separated list.

If a `main` method is present, it is executed when the class is run as an application. It can create objects, evaluate expressions, invoke other method, and do anything else needed to define an object's behavior.

Expression Statements

	C/C++/Java
Assignment	*Name = Expression;* *Name Op = Expression;* *++Name; --Name;* *Name++; Name--;*
Calls	*Name (Formals)*

Control Flow Statements

	C/C++/Java
Empty statements	;
Labeled Statements (case and default only appear in switch statements)	*Label: Statement* `case` *ConstatExpression : Statement* `default : Statement`
Selection Statements	`if` *(BooleanExpr) Statement* `if` *(BooleanExpr) Statement*$_1$ `else` *Statement*$_2$ `switch` *(IntegerExpr) Block*
Iteration Statements	`while` *(BooleanExpr) Statement* `do` *Statement while (BooleanExpr)* `for` *(InitExpr; BooleanExpr; IncrExpr) Statement*
Jump Statements	`break` *Label*$_{Opt}$; `continue` *Label*$_{Opt}$; `return` *Expression*$_{Opt}$;
Block/Compound	{ *sequence of statements* }

Input/Output

C	C++	Java
`#include <stdio.h>`	`#include <iostream.h>`	`import java.io.*`
`scanf();`	`cin >>` *Name;*	`System.in`
`printf();`	`cout <<` *Expression;*	`System.out`

Exception handling

	C	C++	Java
Exception handling is	NA	optional	required
define an exception	NA		`public class` *Exception* `extends` *ExeptionClass* *{* `public` *Exception (Formals) Block }*
Declare an exception	NA	NA	`throws` *Exception ;*
Generate an exception	NA	`throw` *Exception () ;*	`throw` *Exception (Actuals) ;*
Detect and handle exceptions	NA	`try` *Block* `catch` *(Exception* `Exception`*)* $Block_0$ ... `catch` *(Exception* `exception`*)* $Block_{n-1}$	`try` *Block* `catch` *(Exception* `Exception`*)* $Block_0$ ... `catch` *(Exception* `exception`*)* $Block_{n-1}$
Done whether an exception occurs on not	NA	NA	`finally` $Block_n$
Catch any exception	NA	NA	`catch (Exception e)` *Block*

Multithreading

	Java

Standard Libraries and Packages

C	**C++**	**Java**

`assert.h`
`ctype.h`
`errno.h`
`float.h`
`limits.h`
`local.h`
`math.h`
`setjump.h`
`signal.h`
`stdarg.h`
`stddef.h`
`stdio.h`
`stdlib.h`
`string.h`
`time.h`
	the C libraries plus
	`iostream.h`
	and commercial class libraries
		`java.applet`
		`java.awt`
		`java.awt.image`
		`java.awt.peer`
		`java.io`
		`java.lang` -automatically imported
		`java.net`
		`java.util`
		Enterprise API - JDBC, IDL, RMI
		Server API
		Security API
		Commerce API
		Management API
		Media API - 2D, Framework, Share, Animation, Telephony, 3D
		Beans API
		Embedded API

Tools

Program editor	Unix	Wintel	**C**	**C++**	**Java**
Compiler Interpreter Linker & Loader Preprocessor Cross references Source-level debugger Debugging aids System builder Version manager Design editor Code generator Testing aids Documentation management	vi, emacs make rcs		gcc gdb	g++ gdb	javac java jdb javap (dissassembler) appletviewer javah - C header files javadoc

---

Last update:

Send comments to: webmaster@cs.wwc.edu

Notes on Ethics

Anthony Aaby
Walla Walla College
aabyan@wwc.edu
Last Modified: .
Comments and content invited: aabyan@wwc.edu

Theoretical Ethics

Computational Ethics

A rough sketch

Anthony Aaby
Walla Walla College
aabyan@wwc.edu

Status: work in progress - rough draft
Started: November 2000
Last Modified: .
Comments and content invited: aabyan@wwc.edu

ABSTRACT: Competition for and consumption of resources is at the core of ethical issues. Solutions to these problems have been at the core of both operating system design and internet algorithms. The solutions are traditionally integrated into the software. However, the emergence of intelligent autonomous agents such a bots and spiders which compete with human users for resources on the internet have introduced unpredictable and uncontrollable elements into the environment.

Theoretical studies in evolutionary ethics and experiments with artificial life suggest ways in which ethical behavior may emerge in autonomous agents. In this paper, a rough sketch of how ethics may be given a computational formulation that will assist in the emergence of ethical autonomous agents.

This is a rough sketch of an ethical theory based on metaphysics where ethics can be viewed as a formal system, an abstraction of reality, much as the various geometries are abstractions of reality. The result is a teleological computational ethical theory which provides justification for altruism and reciprocity.

1 Introduction

Human and non-human users share the Internet - the WWW. Non-human agents are both autonomous and social just as are human users. They send and receive messages, communicating with both humans and machines. Some autonomous agents carry out jobs, such as searching the WWW,

arranging meetings or compiling music recommendations, more or less anonymously and act on behalf of a single user or an organization. Non-human agents as well as more hostile viruses and worms compete with humans for bandwidth, cpu cycles, and storage space. Autonomous agents possessing the capacities to do things that are useful to humans also have the capacity to do things that are harmful to humans and other entities [Helmers et al]. Ethics provide behavioral guidelines in the competition for and consumption of resources. As artificial intelligence has moved closer to the goal of producing fully autonomous software agents, ethical issues in the interaction between and among humans and their autonomous agents increase in importantance.

Just as human beings differ in their skills and ethical capabilities, so autonomous software agents differ in their skills and ethical[1] capabilities. And we have no reason to expect autonomous agents to be any more uniform in their decisions then an arbitrary collection of humans. Further, there is no reason to believe that even if moral perfection in machines were computationally attainable[2] [Moore, Allen et al] that all autonomous agents would be constructed with perfect ethical capabilities.

The existence of a wide variety of autonomous software agents and anti virus software which provides in effect, private security guards is evidence enough that we are far past having the luxury of theorizing about what ethical values autonomous agents should have as Asimov has done with his *Three Laws of Robot Ethics*[3] [Asimov]. We now have, on the internet, a heterogeneous mix of human and non-human agents with a wide variety of ethical standards and abilities. The issue interest is not the reasoning power of computers but the evolution of proactive and reactive ethical behavior in autonomous agents.

Martijn Koster created *Guidelines for Robot Writers* and *A Standard for Robot Exclusion*. The latter describes the mechanisms for WWW servers to indicate to robots which parts of their server should not be accessed and the former are suggestions for the design and management of software agents which involve voluntary compliance with the robot exclusion mechanisms.

How should ethical components be constructed for such agents? Computationally, the call by John Stuart Mill and Jeremy Bentham for the greatest good for the greatest number, seems to be a natural starting point. Just as Horn clause logic and the unification algorithm has provided a computational approach to reasoning suitable for use by a machine, so we must devise a computational approach to ethical behavior suitable for autonomous agents. Koster's approach is a beginning.

The language of ethics structures the values of the real world just as the language of geometry structures the spatial aspect of the real world. However, ethics differs from geometry in two significant ways. The the concepts of ethical language can slip and slide and sometimes one ethical principle will conflict with or override another [Maurice Stanley]. These differences suggest that a nontraditional logic be used (such as a *multivalued default logic* or a *fuzzy logic*). In this paper I confine myself to the language and leave a discussion of the logic for a later paper.

There are several alternate approaches. One is to use genetic algorithms and the methods of evolutionary programming to create an artificial life community with emergent ethical behaviors. Another is that of evolutionary ethics and sociobiology.

Ethics must be grounded in metaphysics [Miculan]. The ethics developed here recognizes the nature of reality rather than attempts to prescribe morality. Since the ethical code is derived from fundamental metaphysical principles, it will be suitable for any natural or artificial community of interacting entities. The code is developed as far as the principle of reciprocity.

The remainder of this paper is structured as follows.
Section 2 is an overview of artificial societies including cooperating and competing processes in operating systems, the internet, and social simulations.
Section 3 is the core of the paper and it presents the metaphysical foundations and the emergent ethics.
Section 4 presents the conclusions.

2 Artificial societies

In a operating system environment programs form a community of cooperating and competing processes which compete for access to a variety of scarce resources some of which can be shared, others to which a process must have exclusive access. Processes

- execute at a non-zero speed but no assumption can be made regarding relative speeds, and
- request resources at unpredictable times and in unpredictable amounts.

In the environment the utilization of the resources must include:

- *Efficiency* - Resources should be used as much as possible.
- *Fairness* - Processes should get the resources they need.
- *Absence of deadlock or starvation* - No process should wait forever for a resource.
- *Protection* - No process should be able to access a resource with out permission.

Efficiency, fairness, absence of deadlock or starvation and protection are designed into the operating system.

As an example of difficulties that arise with multiple processes, consider two individuals attempting to cross a stream from opposite sides where the set of stepping stones will support only one person at a time. It is easy to imagine a situation where the two become deadlocked. The necessary conditions for deadlock are:

- *Mutual exclusion:* once a process obtains a particular resource, it has exclusive resource.
- *Hold and wait:* a process may hold a resource at the same time it requests another one.
- *Circular waiting:* each process holds a resource while waiting for a resource held by another process.
- *No preemption:* resources can be released only by action of the resource holding the process.

The usual solution is to implement a resource manager (the operating system) from which processes request resources. While it is possible to construct an environment where deadlock cannot occur, the

resulting solution is not considered efficient enough to be practical. Instead the operating system implements several techniques to reduce the likely hood of deadlock by insuring that one or more of the necessary conditions for deadlock cannot be met. Figure 1 summarizes methods for preventing and avoiding deadlock.

Figure 1: Methods for prevention and avoidance of deadlock

Deadlock prevention - prevent one of the necessary conditions for deadlock from holding.

- *Mutual exclusion*
  - Create virtual resources.
- *Hold and wait*
  - Require a process to request all of its resources at once or
  - to release all currently held resources prior to requesting any new resources.
- *Circular wait*
  - Establish a total order on all resources in the system and allow processes to acquire a resource only if it's index is greater than all the indices of the resources it already has.
- *Preemption*
  - Implement round robin sharing.

Deadlock avoidance: Use the Banker's algorithm to allocate resources.

A computer network consists of a community of a large number of nodes (computers) in an environment where the network changes in topology, in the underlying technologies upon which they are based, and in the demands placed on them by application programs. The network must provide general, cost-effective, fair, robust, and high-performance connectivity among the nodes in the network. The individual nodes and application programs may engage in hostile, uncooperative behavior. However, key nodes in the network utilize algorithms to minimize the negative effects of hostile behavior. A network environment differs from an operating system environment in that there is no centralized control or management.

A significant amount of research has gone into the study of artificial societies and the simulation of social environments. Perhaps the most well known is the research of Robert Axelrod who has studied the Iterated Prisoner's Dilemma problem. In most cases the result has been the emergence of a cooperative society based on some variant of the Tit-for-Tat strategy suggested by Anatol Rapoport. In addition research into artificial life with evolutionary programming techniques and genetic algorithms paves the way for improved understanding of evolution and social behavior.

3 Computational ethics

Miculan has proposed to ground ethics in Whitehead's metaphysics. Her approach is sumarized in Figure 2.

Figure 2: from Alison Roberts Miculan's *Ethics and Reality*

Metaphysical principles	
Principle of interrelation	The universe is completely interrelated.
Principle of novelty	Creativity allows disjunctive elements to form a conjunctive new entity.
Ethical principles	
Ethical Principle of Value Ontological ethical principle	All existents have value.
Goodness	Goodness is that which maintains and enhances existence.
Evil	Evil is that which destroys, degrades or undermines existence.

While the end results are similar, I use a different formulation. I begin with entities and actions.

> **Terms:** The metaphysical universe consists of entities which engage in actions which change the state of the universe.

The terms *entity, actions,* and *state* are left undefined and undifferentiated so that the theory may include both animate and inanimate entities and be applicable to both.

The relationships between the entities in the universe are described by two axioms - the axioms of dependence and independence. They apply to both to animate and inanimate entities, to entities with and without free will. The first axiom, independence, describes what range of actions are available to an entity.

> **Axiom of independence:** Every entity has the right to do whatever it wants (Smullyan 1977).

The relevant phrase is "the right to do whatever it wants". The behavior of entities is determined by the laws of nature which define its behavior. In addition, the behavior of entities with free will, is determined by both the laws of nature and the free will choice of the entity. The mechanism that determines the behavior is immaterial. It is only the behavior that is of interest. As Miculan says,

> Ethical actions must take place in a context in which many (at least two) possible actions could occur. That is to say, actions which could not have been made otherwise are not ethical decisions (in fact, they are not decisions at all).

Different entities may have rights to do incompatible acts. The consequence of an action is as described in

> **Axiom of dependence:** Every action of an entity affects all other entities.

Some alternate formulations of the axiom of dependence include

- The universe is completely interrelated (Whitehead 1974 & 1978) and
- Entities are interdependent.

However, an entity's "right to do whatever it wants" does not necessarily prevent another entity's right to interfere with the rights of the other as Smullyan (1977) points out,

> If the people want laws, they have a perfect right to pass them. The criminal has a perfect right to break them, the police have a perfect right to arrest him, the judge has a perfect right to sentence him to jail, and so on.

Emerging computational ethics

Normative ethical systems (both religious and legal) as well as organizations of all types exist to guide individuals through the choices between the incompatible rights of distinctive entities. In order to compare possible actions with the intent of determining the preferred ethical course we need an axiom that provides a mechanism for assigning a value to an action. I find this in the nature of entities themselves. The axiom of integrity describes the goal of each entity which is necessary to provide both motivation and value for actions.

> **Axiom of integrity:** Every entity wants to maintain its integrity.

Some alternative axioms include

- Axiom of identity: Every entity wants to maintain its identity.
- Axiom of existence: All existents have value [Miculan].

While I would not want to argue that integrity, identity, and existence are synonyms, they are close enough in this context.

Where an entity has a choice of behaviors, ethical systems suggest that some behaviors are preferred over others labeling some a "good" or "right" others as "bad" or "wrong". I use the following definitions of goodness and evil [adapted from Miculan].

> **Definition:** *Goodness* is that which maintains and enhances integrity.

> **Definition:** *Evil* is that which destroys, degrades or undermines integrity.

Using these definitions, actions may be identified as good or evil and by extension, entities which engage in good or evil actions may by association be identified as good or evil. The computation of the goodness value of an action *A* will depend on all future actions that result from action *A*. Since there are no doubt many alternative actions that could result the situation is much like a predicting the

outcome of a chess game and in principle is the same. These definitions place this theory among the teleological theories as they essentially say that an action is morally right if the consequences of that action are more favorable than unfavorable.

In this development I differ with Miculan. I focus on integrity while Miculan focuses on existence. For example, Miculan's Ontological Ethical Principle states that as a fact of our very existence, we have value. The following table summarizes Miculan's formalization.

Figure 3: Computational ethics

Definition

The metaphysical universe consists of *entities* which engage in *actions* which change the *state* of the universe.

Axioms

The universe is completely interrelated.

Every entity has the right to do whatever it wants.

Every entity wants to maintain its *integrity*.

Definitions

Incompatible actions are actions by two entities which result in a state in which either entity is unable to maintain its integrity.

Goodness is that which maintains and enhances integrity.

Evil is that which destroys, degrades or undermines integrity.

mutually assured destruction

Proposition

If two entities have incompatible wants, they may interfere with each other.

Two entities of equal strength and incompatible wants, can survive only through the fear of mutually assured destruction.

The principle of rights is necessary for an objective code of morality.

4 Conclusions

Patricia Williams would argue that while the weak interpretation (love self, kin, and friend) of the Love Command is likely to be computable, the strong interpretation (love of neighbor as oneself) is not likely to be computable.

Acknowledgments

Notes

1 ... possible behavior ...
I use *ethic*, *ethics*, and *ethical* instead of the more accurate *social laws* or *socially acceptable behavior* because they are shorter.

2 I find discussion of the possible incompleteness of ethical systems and whether a computer program could pass the Turing test and be perceived as ethical irrelevant for two reasons. First, a system is only interesting if it is incomplete i.e., ethics is interesting because of the existence of ethical dilemmas. And second, whether or not computer programs can be made perfectly ethical is not relevant because humans are not uniformly ethical and are unlikely to be perfectly ethical with the result that ethically imperfect (hostile) programs have and will continue to be constructed.

I believe that the most fruitful approach is to recognize the reality of this environment. If robots can and do evolve beyond the capabilities of humans, I hope that they will be gentle on us, the lower species. If they cannot, then I hope that we should have a class of gentle robots that help enrich our lives.

For those who disagree, tell me how the human mind works and then we can decide whether machines can think.

3 Asimov has proposed three laws for robots which impose ethical behavior on robots which illustrate both the Ethical Principle of Value and the definitions of Goodness and Evil.

Isaac Asimov's Three Laws of Robot Ethics

1. A robot may not injure a human being, or, through inaction, allow a human being to come to harm.
2. A Robot must obey the orders given it by human beings except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

His laws conform to the ethical principle of value in protecting human life and robot existence. The laws are asymmetric with respect to humans and robots making robots second class citizens with respect to humans. Of course an unscrupulous manufacturer of robots is likely to ignore these laws.

4 Ethical rules have been proposed for information gathering robots (bots) loosed on the internet. The rules are based on the traditional rules of Netiquette [Koster, Helmers et al]:

Traditional rules of Netiquette

1. Never disturb the flow of information!
2. Help yourself, this is an expression of decentralized organization.
3. Every user has the right to say anything and to ignore anything.

References

Allen, Varner, and Zinser *Prolegomena to any future artificial moral agent* Journal of Experimental & Theoretical Artificial Intelligence Volume: 12 Number: 3 Page: 251 -- 261
Asimov, I. (1968). *The rest of robots*. London: Granada 1968.
Gorniak-Kocicowska, Krystyna *The Computer Revolution and the Problem of Global Ethics* The Research Center on Computing & Society at Southern Connecticut State University 2000.
Helmers, Hoffmann, and Stamos-Kaschke *(How) Can Software Agents Become Good Net Citizens?* CMC Magazine, Vol. 3, No. 2, Feb. 1997
Hoffmann, Robert (2000) *Twenty Years on: The Evolution of Cooperation Revisited* Journal of artificial Societies and Social Simulations vol. 3 no. 2,
Koster, Martijn (1993) *Guidelines for Robot Writers* URL: info.webcrawler.com/mak/projects/robots/robots.html
Miculan, Alison R. *Ethics and Reality* 20th World Philosophical Congress
Moor, James. *Is Ethics Computable?* Metaphilosophy 26, nos. 1-2 (January-April): 1-21.
Sandip Sen, `` Reciprocity: a foundational principle for promoting cooperative behavior among self-interested agents"" , in *Proc. of the Second International Conference on Multiagent Systems*, pages 322--329, AAAI Press, Menlo Park, CA, 1996.
Shoham, Yoav and Tennenholtz, Moshe. "On social laws for artificial agent societies" Artificial Intelligence vol 73.
Smullyan, Raymond (1977). *The Tao is silent*. Harper & Row 1977.
Smullyan, Raymond (1983). *5000 B. C. and other Philosophical Fantasies* St. Martins Press 1983.
Stanley, Maurice F. *The Geometry of Ethics* 20th World Philosophical Congress
Whitehead, A. N. *Process and Reality* Macmillan 1978.
Whitehead, A. N. *Religion in the Making* New American Library 1974.
Williams, Patricia A. *Christianity and Evolutionary Ethics: Sketch Toward a Reconciliation* Zygon, vol. 31, no. 2 (June 1996)

- Last Modified - . Comments and content invited aabyan@wwc.edu

---

Is ethics resource management?

Anthony Aaby
Walla Walla College
aabyan@wwc.edu
Last Modified: .
Comments and content invited: aabyan@wwc.edu

ABSTRACT: Competition for and consumption of resources is at the core of ethical issues. Solutions to these problems have been at the core of both operating system design and internet algorithms. The solutions are traditionally integrated into the software. However, the emergence of intelligent autonomous agents such a bots and spiders which compete with human users for resources on the internet have introduced unpredictable and uncontrollable elements into the environment.

Wonder if in the emerging complexity of ... Hope that ethicists will recognize elements of their own discipline and perhaps find a new language in which to cast their problems.

Ethical theories can be put to use in OS/Networking design.

resource management, improve the predictability of our environment, ...

Introduction

Applied ethics ...
ethics in human resource management ...
ethics of natural resource management (environmental ethics) ...
ethical rules are in essence, rules for resource management ...

Consider six principles that the vast majority of ethicists and moral agents generally would accept.

Figure 1: Ethical domains and resources

Ethical domain	**rule for managing**
Honesty and Promise-Keeping	Truth and information

Nonmaleficence and Beneficence	Life
Autonomy	Individuality
Justice	Order

1. Principle of Autonomy: Generally, people have the right to live their lives as they see fit so long as doing so does not interfere with the correlative rights of others.
2. Principle of Equality (Justice): Generally, people should be treated in a manner that accords to each an equality of respect.

Look at resource management from the perspective of operating system design and computer network design to provide terminology and to survey some of what computer scientists know about resource management.

Examine some ethical rules from the perspective of resource management.

Suggest directions for further research into the idea of ethics as resource management.

Operating Systems

An operating system is a collection of processes and resources. The processes form a community of cooperating and competing processes which compete for access to a variety of resources some of which can be shared, others to which a process must have exclusive access.

The metaphysical world

Resources

consumable
shareable

- simultaneously
- serially

Processes

A *process* is a program in execution. Processes have a *life cycle* which begins with its *creation*. It then aternates between *running* and *waiting* for a *resource*. Its final state is when it is finished or is "killed" and *exit*s. During its life cycle it *consumes*, *shares*, and *creates* resources. Its use of resources may be *cooperative* and *competitive*.

independent, hostile

In a operating system environment processes form a community of cooperating and competing processes which compete for access to a variety of scarce resources some of which can be shared, others to which a process must have exclusive access. Processes

- execute at a non-zero speed but no assumption can be made regarding relative speeds, and
- request resources at unpredictable times and in unpredictable amounts.

Design considerations:

Independent: process cannot affect or be affected by the other processes
Dependent: processes can affect or be affected by the other processes. Possible to deadlock or starve. There are several subcategories
cooperating -- shared task and possibly shared resources
competing -- may starve opponent
hostile -- attempt to destroy another's resources

Laws which

describe possible actions
prevent ...
guarantee the successful coexistence of multiple agents

Processes interact with each other through shared resources.

Bad things

Race conditions

Starvation

Deadlock

Safety property: nothing bad will happen (negative duties).

Safety properties can always be satisfied by processes that do nothing.

Good things

Liveness property: something good will happen (positive duties).

Liveness properties specify things that must be done.

In the environment the utilization of the resources must include:

- *Efficiency* - Resources should be used as much as possible.
- *Fairness* - Processes should get the resources they need.
- *Absence of deadlock or starvation* - No process should wait forever for a resource.
- *Protection* - No process should be able to access a resource without permission.

Operating system designers try to guarentee that the resulting operating satisfies these design goals.

Traditionally, efficiency, fairness, absence of deadlock or starvation and protection are designed into the operating system.

As an example of difficulties that arise with multiple processes, consider two individuals attempting to cross a stream from opposite sides where the set of stepping stones will support only one person at a time. It is easy to imagine a situation where the two become deadlocked. The necessary conditions for deadlock are:

- *Mutual exclusion:* once a process obtains a particular resource, it has exclusive resource.
- *Hold and wait:* a process may hold a resource at the same time it requests another one.
- *Circular waiting:* each process holds a resource while waiting for a resource held by another process.
- *No preemption:* resources can be released only by action of the resource holding the process.

Scheduling Criteria (Goals)

- Fairness: each process gets its fair share
- Efficiency: CPU utilization
- Throughput: number of processes/time unit
- Turnaround: time it takes to execute a process from start to finish
- Waiting time: total time spent in the ready queue
- Response time: amount of time it takes to start responding (average, variance)

It is desireable

- to ensure that all processes get the CPU time they need and
- to maximize CPU utilization and throughput, and
- minimize turnaround time, waiting time, and response time.

And may want to

- optimize the minimum or maximum (minimize maximum response time)
- minimize variance in response time (i.e. predictable response time)

Managers

In order to solve the coordination problems, operating systems are designed around a variety of

managers. There are managers for devices, file systems, the memory system, the central processor, and the processes.
The core or kernel of the operating system provides protection services ...

Device drivers
File system manager
Memory manager
Process manager/scheduler
life cycle
create
death - kill

Design Principles

The usual solution is to implement a resource manager (the operating system) from which processes request resources. While it is possible to construct an environment where deadlock cannot occur, the resulting solution is not considered efficient enough to be practical. Instead the operating system implements several techniques to reduce the likely hood of deadlock by insuring that one or more of the necessary conditions for deadlock cannot be met. Figure 2 summarizes methods for preventing and avoiding deadlock.

Figure 2: Methods for prevention and avoidance of deadlock

Deadlock prevention - prevent one of the necessary conditions for deadlock from holding.

- *Mutual exclusion*
  - Create virtual resources so that each process appears to own the resource. Typically done for printers.
- *Hold and wait*
  - Require a process to request all of its resources at once or
  - to release all currently held resources prior to requesting any new resources.
- *Circular wait*
  - Establish a total order on all resources in the system and allow processes to acquire a resource only if it's index is greater than all the indices of the resources it already has.
- *Preemption*
  - Implement round robin sharing.

Deadlock avoidance: Use the Banker's algorithm to allocate resources.

Computer Networks

A computer network consists of a community of a large number of nodes (computers) in an environment where the network changes in topology, in the underlying technologies upon which they are based, and in the demands placed on them by application programs. The network must provide general, cost-effective, fair, robust, and high-performance connectivity among the nodes in the network. The individual nodes and application programs may engage in hostile, uncooperative behavior. However, key nodes in the network utilize algorithms to minimize the negative effects of hostile behavior. A network environment differs from an operating system environment in that there is no centralized control or management.

Ethical Rules as Resource Management Rules

If every entity gets whatever resources it wants/needs, there are no ethical issues of importance.

Consider a static universe where nothing ever changes. Such a universe can have no need of ethics. Consider a second universe where there is only one active entity and everything else is static. If ethics exists in this second universe, they are completely determined by the single active entity.

Ethics is unnecessary

- Zero entities
- One entity
- n+2 entities
    - independent
    - dependent

Internal state

- wants vs needs
- beyond capabilities - sunlight, sky color, grass color

Ethics becomes an issue only when entities are capable of conflict over resources.

Conclusions

further research, open questions

References

Operating systems texts

Can ethics contribute to system design?

Anthony Aaby
Walla Walla College
aabyan@wwc.edu

Document purpose: *This paper is the result of attempting to answer the questions "Is the language of ethics a specialized language of a limited domain or is it a language that can be adapted to meet the needs of areas such as resource management, decision theory, and system requirements, verification, and validation? And if so, what is the potential for cross-fertilization between the areas?".*
Document status: *rough, working draft*
Document use: *no warranty is expressed or implied for the use of this material.*

Last Modified: .
Comments and content invited: aabyan@wwc.edu

Abstract: This paper explores the possibility that the vocabulary of ethics can provide a useful vocabulary for system life cycle processes especially in the areas of structuring requirements and in the interaction between requirements and the activities of verification and validation.

Introduction

This paper is the result of attempting to answer the question "Is the language of ethics a specialized language of a limited domain or is it a language that can be adapted to meet the needs of software engineering specifically, system requirements, verification, and validation?".

The remainder of this paper is structured as follows.
Section two is a short review of ethics.
Section three is a short review of the system life cycle processes.
Section four is an examination of operating system design from the view of ethical theories.
Section five is a summary and conclusion.

A short review of ethics

The field of ethics involves systematizing, defending, and recommending concepts of right and wrong behavior. Ethics is divided into three general areas, metaethics, normative ethics, and applied ethics. *Metaethics* investigates the source and meaning of ethical principles. *Normative ethics* describe the ethical standards that regulate right and wrong. *Applied ethics* resolve specific ethical controversies using the tools of metaethics and normative ethics. The next two subsections review metaethics and normative ethics.

Metaethics

The task of metaethics is to determine the set of entities and behaviors of interest and to determine the source of ethical values. Traditional sources of ethical values identified in metaethical argument may be divided into objective or subjective sources. Objective sources include divine commands and part of the fundamental nature of the universe. Subjective sources include the individual and culture.

... duties and right vs consequences and good ...

Normative ethics

Normative ethics involves arriving at the ethical standards that regulate conduct. The key assumption is that there is only one ultimate criterion of ethical conduct whether it is a single rule or set of principles. It is common to classify ethical theories into several categories:

1. virtue theory,
2. deontological theories,
3. consequentialist theories, and
4. relativistic theories.

In addition to determining ethical behavior, an ethical theory should prescribe a method for resolving ethical conflicts should any arise in an application of the theory.

Virtue theory suggests that ethic behavior is the result of good habits of character or virtues. Suggested virtues include: wisdom, courage, temperance, justice, fortitude, generosity, self-respect, good temper, and sincerity. Vices or negative virtues include cowardice, insensibility, injustice, and vanity.

Deontological theories variously identify duties, rights, obligations, and categorical imperatives. Duties and obligations have been classified under several categories including

- duties to God,
- duties to oneself, and
- duties to others which include
    - duties to family,
    - social duties, and
    - political duties.

The basic rights include life, liberty and the pursuit of happiness and are natural, universal, equal, and inalienable (following John Locke and Thomas Jefferson). A basic formulation of the categorical imperative is: actions toward another entity should reflect the value of that entity (Kant).

The focus is on moral duties or obligations rather than on moral value or goodness.

Intentions play a significant role in determining whether an act is ethical.

Consequentialist (teleological) theories determine ethical behavior by weighing the consequences of an action. The good and bad consequences of an action are tallied and if the total good consequences outweigh the total bad actions, then the action is ethically proper. Thus an action is ethical if the consequences of that action are more favorable than unfavorable with respect to some criteria. Criteria include affected groups and the dimension of time. Three criteria with respect to agents have been suggested

- *Ethical egoism:* only consequences to the entity performing the action are considered.
- *Ethical altruism:* only consequences to everyone except the agent performing the action are considered.
- *Utilitarianism:* the consequences to everyone of an act or rule are considered (Bentham, Mills).

In the dimension of time, the influence of action may extend beyond the immediate consequences of the act. The focus on consequences is problematic since consequences are, in almost all cases, outside the agent's immediate and direct control.

The focus is on moral value or goodness rather than on moral duties or obligations. An action's consequences (what is good) are more important than on moral obligations (what is right). Human nature and experience determine what the good is.

The following table contrasts the features of deontological and teleological theories.

DEONTOLOGICAL THEORIES	TELEOLOGICAL THEORIES
1. The focus is on moral duties (what is right) rather than on an action's consequences (what is good). 2. Considerations about moral duties are more important than considerations about moral value. 3. Since the focus is on moral duties, the individual's intentions have a substantial role in a situation's moral evaluation and consequences that arise through the individual's actions have no relevance.	1. The focus is on an action's consequences (what is good) rather than on moral duties (what is right). 2. Considerations about moral value are more important than considerations about moral duties. 3. Since the focus is on moral value, the consequences that an individual's actions produce have a substantial role in a situation's moral evaluation and the individual's intentions have no

4. There is no one specifiable relation between good and right.
5. Concepts about moral value (i.e., what is good) are definable in reference to concepts about moral duties (i.e., what is right).
6. The right is prior to the good.
7. An action's goodness (or value) depends upon the action's rightness.
8. It is the individual's moral status that is important.
9. The statement 'x is a moral individual' means 'x did what was right with the right intention'.
10. Deontological ethics stresses that reason, intuition or moral sense reveals what is right.
11. There are some acts that are moral or immoral in themselves.
12. Moral duties have a negative formulation.
13. Other's personal interests or happiness have no relevance in one's moral considerations or evaluations, one's own moral duties have precedence over all other considerations.
14. To do what is moral (i.e., right) requires that one observe one's moral duties, possess the right intentions and avoid those actions that are immoral in themselves.

relevance.
4. There is a specifiable relation between good and right.
5. Concepts about moral duties (i.e., what is right) are definable in reference to concepts about moral value (i.e., what is good).
6. The good is prior to the right.
7. An action's rightness depends upon the action's goodness (or value).
8. It is the action's moral status that is important.
9. The statement 'x is a moral action' means 'x produces at least as good consequences as all other possible actions'.
10. Teleological theories argue that experience, rather than reason, reveals what is good.
11. There are no actions that are moral or immoral in themselves.
12. Moral duties have a positive formulation.
13. One must give equal and impartial consideration to other's interests and happiness, as well as one's own, in all moral considerations and evaluations.
14. To do what is moral (i.e., good) requires that one acts so as to maximize the happiness that one's action produce.

Relativistic theories reject any ethical rule as universal or absolute. Ethical beliefs and practices vary from culture to culture. There is no objective way to assess the validity of ethical principles.

Comment: Each category of ethical theories has something to contribute, an approach to ethical decision making. They are not necessarily mutually exclusive theories. The collection of theories is a resource, a collection of tools, to be used as needed and when appropriate. The designer of an artificial society is free to select any ethical system for the society. In societies with a mix of human and autonomous agents, ...

System life cycle processes

The system life cycle processes

- Problem statement (needs purpose)
- Requirements - define right (or is it good?), in addition to constituting a contract between the client and the programmer.
    - specification
- Design - how the requirements will be met
- Implementation -
    - verification - checking that the implementation meets the specification (Are we building the product right?)
    - validation - checking that the implementation meets the expectations of the customer is suitable for its intended purpose. (Are we building the right product?)
- Retirement

Software engineering

Pre and post conditions

In the design of functions the contract between the user and the function is given by the pre and post conditions. Pre and post conditions are also the *specification* of the function.

- *pre-condition:* The conditions the user must meet in order to receive the service provided by the function.
- *post-condition:* The service the function guarantees to provides.

In deontological terms, the pre-condition describes the duty of the user and the post-condition describes the duty (obligation) of the function (or we may say the rights of the user provided the user fulfills its obligation). So for example, the contract between a user and an implementation of the factorial function guarantees to provide the user with the value *n!* provided the user supplies a natural number *n* between 0 and *a* inclusive, where *a* is some implementation defined limit, i. e.,

f(n) : if n=0 then 1 else n*f(n-1) where

- *pre-condition:* ***n*** must be a natural number greater than or equal to 0 and less then ***a*** (the implementation defined limit),
- *post-condition:* the value returned is ***n!***.

The term used in software design is *correctness* which corresponds to the ethical terms of right or good.

Software verification and validation

Verification, checking that the implementation meets the specification (Are we building the product right?), corresponds to the approach of deontological ethics.

Validation, checking that the implementation meets the expectations of the customer (Are we building

the right product?), corresponds to the approach of consequentialist ethics.

Safety and liveness properties

... two classes of behavioral properties: safety and liveness properties.

Safety properties assert what the entity (or system) is allowed to do, or equivalently, what it may not do. *Liveness properties* assert what the entity (or system) must do.

For example, asserting that an entity may not tell falsehoods is a safety property. Asserting that an entity must eventually tell the truth or that a system must be fair are examples of liveness properties. In the specification of systems of concurrent entities, safety and liveness properties are specified separately.

... positive and negative duties ...

... invariant ... variant ...

> **Safety property:** nothing bad will happen (negative duties).

Safety properties can always be satisfied by processes that do nothing. []p

> **Liveness property:** something good will happen (positive duties).

Liveness properties specify things that must be done. Liveness properties ... termination, fairness ... <>p

All properties of concurrent systems are describable as a conjunction of safety and liveness properties.

Safe liveness []<>p

Positive Duties - duties to do something.
Negative Duties - duties to refrain from doing something.

Fairness is a liveness property.
weak fairness

> **Weak fairness** on *A* asserts that if *A* eventually becomes enabled forever, than infinitely many *A* steps must occur.

Strong fairness

> **Strong fairness** on *A* asserts that if *A* is infinitely often enabled, than infinitely many *A* steps must occur.

Weak fairness:	<>[] (enabled *A*) => []<>*A*
Strong fairness:	[]<> (enabled *A*) => []<>*A*

Artificial societies

Artificial societies consist of a collection of resources and autonomous agents. Agents have a life cycle which begins with its creation. It then alternates between doing internal computation, waiting for a resource, and engaging in communication. Its final state is when it is finished or is "killed" and exits. During its life cycle it consumes, shares, and creates resources. Its use of resources may be cooperative or competitive. Agents may be independent, cooperative, or competitive (even hostile) toward other agents. For the purposes of this paper, operating systems with their mix of processes and resources and computer networks with their nodes, connections, and packets flowing through the network are considered as examples of artificial societies.

Agents which by design are intended to work independently of other agents may, inadvertently, through their use or need for a resource cause inconvenience or even fatal damage to another agent. Readers familiar with personal computer systems should be familiar with system crashes. When a system crashes or hangs, it is an indication that an agent has interfered with another process. A system's integrity can be compromised by even well intentioned agents.

There are two primary concerns in the design of an artificial society. *Safety* is the property that nothing bad happens. *Liveness* is the property that something good will happen.

The customers and designers of systems for artificial societies are in the position of determining the ethics of the society. They determine what behaviors are good and bad and put into place mechanisms for insuring that generally good behavior occurs and bad behavior is minimized. The system goals (requirements, specifications) are the standard by which the behavior in and of the system is evaluated. In this context then, system designers are applied ethicists.

Operating system design and Thomas Hobbes (1588-1679)

Operating systems are designed as a Hobbesian society with an absolute monarch.

... social contract theory ...

Processes can make requests which if granted would violate any of the management goals. The requirements are incompatible. Thus the OS is the absolute monarch of Hobbes.

A computer system

... simplified view of a computer ...

A computer system consists of resources and processes.

Resources

The major hardware components of a computer are the central processing unit (CPU), a memory hierarchy (RAM, hard drives, tape backup), and various input/output (I/O) devices. The components run at different speeds. The speed differential between the CPU the memory hierarchy and between each level of the memory hierarchy can be several orders of magnitude. Bare hardware is unusable without significant software support. For example, software support is necessary to provide a file system on secondary storage. These hardware resources are heterogeneous differing in type, speed, number, and availability (simultaneously or serially), and are usually scarce relative to demand.

Hardware resources	Computer resources are heterogeneous differing in
• CPU • Memory hierarchy • I/O devices	• type, • speed, • number, and • availability (simultaneously or serially), and • are usually scarce relative to demand.

Figure n: Hardware components

Processes

A process is a program in execution. It consists of the executable code (the program), data, contents of various registers in the CPU, and files on secondary storage. Processes are independent in the sense that most are, by design, noncooperative, lacking in awareness of the existence of other programs. Processes differ in

    - their resource requirements, and
    - dependence on other processes,
- have unpredictable resource requirements
- may not be tolerant of hardware failures
- A process may cause another process to fail to satisfy its postcondition, or one or more of its safety or liveness conditions.
- Natural behavior of processes - request and release resources without predictable limits.
- Use of resources is undecideable. Termination is not a decideable property of programs.
- require unpredictable amounts of resources. For example, termination (use of the CPU) is not a decideable property of programs.
-

Processes are amoral. Any system morality is the result of operating system action.

Program behavior may be described by a combination of formulas of the form:

- {Precondition} *P* {Postcondition} which means that if a program *P* is started in a state satisfying the precondition and the process terminates, then it terminates in a state satisfying the postcondition,
- [] *S*, which describe the safety properties satisfied by the program (nothing bad happens), and
- <>*L*, which describe the liveness properties satisfied by the program (something good happens).

Figure n: Program specification

In early computers, each process had exclusive access to the computer (the computer was shared sequentially). Each program had to include the necessary supporting software. Soon libraries of supporting software appeared, quickly followed by a rudimentary operating system where the supporting software stayed resident on the computer. Both with the shared libraries and the operating system, it became necessary to agree on the starting location of the program in memory. Even in such simple systems problems bad behavior occurs. Poorly designed programs often overwrote portions of the operating system with the result that the system crashed necessitating the reloading of the operating system. With the high cost of computers, the large speed differential between the CPU and the file system, and larger memories, it became possible to load multiple programs into memory and switch execution between processes whenever a process needed file access.

The fundamental features of computer systems are:

- The environment consists of
  - a heterogeneous collection of processes and
  - a heterogeneous collection of resources provided by the hardware.
- Processes need resources to complete their tasks.
- Processes interfere with each other in their use of resources.
- Resources are scarce with respect relative to the demand by the number of processes.

These fundamental features lead to the following require an operating system to manage system resources with

- efficiency (resources should be used as much as possible),
- fairness (processes should get the resources they need),
- absence of deadlock or starvation (no process should wait forever for a resource), and
- protection (no process should be able to access a resource with out permission).

An operating system must manage system resources with

- efficiency (resources should be used as much as possible),
- fairness (processes should get the resources they need),
- absence of deadlock or starvation (no process should wait forever for a resource), and
- protection (no process should be able to access a resource with out permission).

Figure n:OS Requirements

The resource management tasks of the operating system are

- Allocation - assign resources to processes needing the resource
- Accounting - keep track of resources - knows which are free and which process the others are allocated to.
- Scheduling - decide which process should get the resource next.
- Protection - make sure that a process can only access a resource when it is allowed

Hardware component	**software component**
CPU System clock and interrupts	interrupt handler process scheduler
Memory hierarchy	memory manager file system manager
I/O devices	device drivers

Figure n: Operating System components

The operating system itself is structured and a collection of processes. ...

OS environment includes

- processes whose intent is determined by its code, designer, and user. None of which may be accessible to the OS or other processes. Only process behavior or process history.
- OS is responsible for system performance which is affected by the consequences of process actions.

References

- Hobbes, Thomas. *Leviathan* 1651
- Locke, John. *The Second Treatise of Government 1764*

Levels

- OS designer & users
- OS evolution
- OS & processes - an ethical model
    - prescriptive ethics: ethics to os
    - descriptive ethics: os to ethics

Computer Networks

Requirements: A computer network must provide general, cost-effective, fair, robust, and high-performance connectivity among a large number of computers in an environment where the network changes in topology, in the underlying technologies upon which they are based, and in the demands placed on them by application programs.

Just as in operating systems where each processes computation is broken in a sequence of small quanta to maximize efficient use of the CPU, so in networks, in order to provide high-performance, communication is broken into a stream of packets. However, the network does not guarantee packet delivery - the packets may be lost, duplicated, corrupted, or delivered out of order.

Discussion

Take One!

The Ethical Universe

Document status: Underdevelopment
Author: Anthony Aaby
Date: Febuary 2001

Introduction

Metaethical discussion focuses on the source of ethics ...
Ethical rules are similar to the rules used to describe the properties of concurrent systems. Concurrent systems are described in terms of safety and liveness properties. A safety property asserts that nothing bad will happen and a liveness property asserts that something good will happen. These properties correspond to negative and positive duties. ...
This paper proposes to recast metaethical discussion in terms of resource management. I leave a number of terms undefined and require a temporal logic to produce a formal treatment ...

Temporal logic

Temporal logic is ordinary logic extended with temporal operators [] (read *henceforth*) and <> (read *eventually*). The formula []*P* asserts that *P* is true now and at all future times, and the formula <>*P* asserts that *P* is true now or at some future time. Since *P* is eventually true if and only if it is not always false, <>*P* is equivalent to ~[]~*P*.

Temporal logic, as it has been defined here, cannot formally specify things like average response time and probability of failure. However, it is useful for the specification of safety and liveness properties. *Safety properties* assert what the system is allowed to do, or equivalently, what it may not do. Safety properties are satisfied by a system which does nothing. Restriction to only producing correct answers is an example of a safety property. *Liveness properties* assert what the system must do. Termination is an example of a liveness property.

As an example of temporal specifications and safety and liveness specifications in particular, we provide a specification of the *The Dining Philosophers Problem.* Five philosophers spend their lives seated around a circular table thinking and eating. Each philosoper has a plate of spaghetti and, on each side, shares a fork his/her neighbor. To eat, the philosopher must aquire two forks. The problem is to prevent deadlock or starvation i. e. insure that each philosopher gets to eat.

Figure 1: Safety and Liveness Specifications: Philosopher P(i)

Safety Properties	[]($eating(i) \vee thinking(i)$)	Philosophers either eating or think

	[]~(eating(i) \/ eating(i+1))	Adjacent philosophers cannot eat simultaneously
Liveness Properties	[]<>(thinking(i) -> <>eating(i)) []<>(eating(i) -> <>thinking(i))	Philosophers alternate between eating and thinking

Fairness is a desirable property of a concurrent system and is definable as a liveness property.

Generalized weak fairness <>[]P => []<>A

Generalized strong fairness []<>P => []<>A

Formally, an *action system* consists of an initial state predicate *Init* and a set of predicates A_i on pairs of states. The A_i are called *system actions*. An action system expresses the safety propery consisting of every behavior $<s_0, s_1, ... >$ whose initial state s_0 satisfies *Init* and whose every pair $<s_i, s_{i+1}>$ of successive states satisfies some system action.

The nature of an ethical universe

The ethical universe is composed of a dynamic set of

- *entities, E* = $\{e_0, e_1, ... \}$.

The set of entities is composed of a set of

- *processes, P* = $\{p_0, p_1, ...)$, and a set of
- *resources, R* = $\{r_0, r_1, ... \}$,

i.e., *P*u*R* = *E*. However, the intersection is not necesarily empty *P*n*R* != {}.

> **Principle of interaction** A process may *wait for*, *acquire*, *consume*, and *produce* a resource.

As a resource, a process may itself be waited for, acquired by, consumed by, and produced by another process. The set of predicates for these actions are found in the following table.

Predicates for an ethical universe

	Predicates
$T(e_i)$, $e_i : T$	of type *T*

$E(e_i)$ exists, is available

$W(e_i, e_j)$ waiting for ej

$A(e_i, e_j)$ acquired (held)

$P(e_i, e_j)$ produced, released

$C(e_i, e_j)$ consumed

The language of the ethical universe contains, entity, process, and resource constants, and the predicates of type, existence, waiting, acquisition, production, and consumption. The formulas are those of first-order temporal logic. In addition to the logical axioms and rules of inference the non-logical axioms of the ethical universe are:

The Axioms

Formula	***Meaning***
$/\backslash e_i, e_j[](P(e_i, e_j) \text{ -> } E(e_j))$	Once a resource is produced or released, it gains existence.
$/\backslash e_i, e_j[](A(e_i, e_j) \text{ -> } \sim E(e_j))$	Once acquired, a resource is not available.
$/\backslash e_i, e_j[](C(e_i, e_j) \text{ -> } []\sim E(e_j))$	Once a resource is consumed, it ceases to exist.

The principle of interaction tells us that entities emerge, change, and disappear over time.

> **Principle of ethical reality** Each entity has the right to do whatever it wants.

Without some freedom of choice, questions of ethics becomes uninteresting. "Wants", of course, is nebulous. My intention is to include both animate and inanimate entities in this discussion because the border between animate and inanimate is not distinct. ... Thus, a better statement is

> **Principle of integrity** Each entity wants to maintain its integrity which consists of behaving in a manner consistent with its attributes and properties.

Thus I don't expect stones to discuss ethics or monkeys to assemble themselves into a mountain.

Entities are not all simple, most are composite, composed of other entities. And as a consequence, composite entities have properties that are emergent - properties that are not predictable from the properties of the constituent parts. The principle of connection applies both to compound entities and the entire ethical universe.

> **Principle of interference** Every entity is connected to every other entity which both constrains and enhances its ability to do whatever it wants.

Ethics in an ideally ethical universe

The characteristics of an ideal ethical world include:

- *Efficiency* - resources should be used as much as possible.
- *Fairness* - entities should get the resources they need.
- *Absence* of deadlock or starvation - no entity should wait forever for a resource.
- *Protection* - no entity should be able to access a resource without permission.

The corresponding formalization of these charactoristics is given in the following table.

Axioms for an ideal ethical world

Formula	***Property***
$[]\forall e_i \forall e_j \{(W(e_i, e_j) \wedge E(e_j)) \rightarrow \langle\rangle A(e_i, e_j)\}$	Efficiency
$\forall e_i \forall e_j \{[](W(e_i, e_j) \wedge E(e_j)) \rightarrow \langle\rangle A(e_i, e_j)\}$	Fairness
$\wedge e_i, e_j \sim[]\{W(e_i, e_j) \wedge \sim E(e_j)\}$	Liveness
$\wedge e_i, e_j, e_j[](A(e_i, e_j) \rightarrow \sim A(e_k, e_j))$	Protection

The principle of connection implies that the actions of one entity may interfere with the rights of another. Hydrogen atoms remain hydrogen atoms unless disassembled and reassembled into atoms of iron by some force external to the atoms. This situation leads to the fundamental problem of ethics.

> **The fundamental problem of ethics** What constraints on individual behavior are necessary to preserve the nature of universe. How to maximize the freedom of the individual to do whatever it wants while minimizing the negative consequences. More formally, what ethical axioms imply the world described by the axioms for an ideal ethical world.

The solution to the fundamental problem of ethics requires a determination ...

> **The goal of individual ethics** Individual ethics is concerned with access to resources and freedom to do whatever it wants.

> **The goal of universal ethics** The fair distribution of resources and the protection of individuals from each other.

Deontological
Consequentialism
Kant's categorical imperative.

Ethical beings are entities which engage in universal ethical behavior without coercion.

Principle of ethical optimism Ethical behavior can be learned, practiced, and habituated.

There are two approaches to producing a ethical world. A centralized approach manages all entities and a distributed approach which requires each entity to exchange messages with other entities to come to agreement. Intermediate solutions are possible as well.

Traditional ethical principles, family, communities, government, and religion are attempts to provide distributed solutions to the ethical problem.

Byzantine consensus protocols may be used in a heterogeneous environment to coordinate the behavior of a group.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

When is ethics interesting?

Anthony Aaby
Walla Walla College
aabyan@wwc.edu

Document status: under development; a very rough draft
Last Modified: .
Comments and content invited: aabyan@wwc.edu

> 'Fascinating' is a word I use for the unexpected. In this case, I should think 'interesting' would suffice.
> -- Mr. Spock in *The Squire of Gothos*

Many an ethical system has fallen to pathological cases -- cases that push the system to the extreme, to the boundary conditions. The focus of this paper is to identify pathological cases based on the size or complexity of the universe. In some of these situations no ethical system can exist, in others, any possible ethical system is trivial, and in others the ethical system presents little intellectual challenge.

0 <= *n* < 2

The first such universe is an empty universe. In an empty universe there are no entities. There is no behavior in this universe. With no behavior of interest there can be no ethics.

The second universe is a universe with one and only one entity. As behavior is perceptible only by reference to other entities, there can be no behavior in this universe and therefore no ethics. It may be argued that an entity may be able to perceive its own behavior. This is to suggest that it is aware of its parts which implies that it is composed of multiple entities - a situation ruled out by the definition of this universe.

2 <= *n*

The third universe is a universe with *n*+2 entities. There are several cases.

Independent behavior

Suppose the behavior of each entity is fully independent of each other entity, then this degenerates to

the universe with only one entity and again there are no ethical issues.

Dependent behavior

Deterministic

Suppose the behavior of some entities is dependent on other entities but the universe is completely deterministic, then as the entities have no choice there are no ethical issues.

Nondeterministic

Suppose the behavior of some entities is dependent on other entities but the universe contains some nondeterminism, then there are two possibilities.

Randomness

If the nondeterministic behavior is the result of innate randomness, then there are no ethical issues.

Choice and computational complexity

If the nondeterministic behavior is the result of choice then the question of interest changes from whether the universe can have an ethical system to what makes an ethical system interesting. I suggest that the answer depends on whether the ethical system is decideable.

Decideable

Suppose the ethical system is so simple as to be completely decideable. This means that any ethical question posed to the system can be answered by the system. In such a system there can be no ethical dilemmas. Whether such an ethical system is interesting depends on the computational complexity of the decision process. Boolean algebra is an example of a decideable theory that is no longer of interest to mathematicians but continues to be of interest (because of applications) to engineers in the form of digital logic.

Computer scientists consider an algorithm that runs in polynomial time as a practical algorithm. So, if the computational complexity of the ethical system is in polynomial time, then entities can be expected to figure out the answer to any ethical question.

If, on the other hand, it requires, say, exponential time, then it is considered to be impractical for all but the smallest data sets. In such instances, heuristics which run in an acceptable amount of time are developed which give acceptable approximations to the correct result. Entities would not be expected to be able to figure out answers to some ethical questions and ethical dilemmas could exist.

Where the decision algorithm is an exponetial algorithm, computer scientists use heuristics to find an approximate or "good enough" algorithm. The search for such algorithms maintains interest in this case. This is the case with propostional temporal logic. Proof construction is decideable in exponential

time but because of its applicability to the specification of reactive systems, work continues to find ways of minimizing the inevitable state explosion of temporal logic proofs.

Undecideable

Suppose the ethical system is undecideable in the sense that first order arithmetic is undecideable (incomplete). There are several interesting parts to this sort of ethical system.

- What questions can be answered with the ethical system?
- Are there useful decideable fragments of the ethical system?
- And of course, the standard questions in which theoretical ethicists would be interested.

So, when is ethics interesting?

I would expect that Mr. Spock would have found all of the these scenarios interesting but none fascinating.

How many minds?

Anthony Aaby
Walla Walla College
aabyan@wwc.edu
Last Modified: .
Comments and content invited: aabyan@wwc.edu

> Love is composed of a single soul inhabiting two bodies. *Aristotle*
> **Principle of reciprocity:** Love your neighbor as yourself.

Discussion: Entities are separated into equivalent classes by a variety of equivalence relations. Membership in an equivalence is dependent on the behaviors of the entities. For example, an entity that engages in criminal behavior may be placed in the criminal equivalence class.

Many ethical systems are include some form of the principle of reciprocity which grants to members of the same equivalence class the same rights and privileges. For example, in Christianity, it is expressed as "Love your neighbor as yourself". The choice of equivalence class is the problem as indicated in the Biblical question "Who is my neighbor?"

Biologically, the answer seems to be the immediate family and first cousins. Biblically, the answer seems to be anyone in need. Animal rights activists extend it to include all animal species. Ecological activists extend it to include all animate nature. In the future it might be extended to include robotic entities.

Several possibilities

1. I am my only neighbor.
2. Members of some group are my neighbors.
3. My neighbor is myself.
4. My neighbor is my past, present, and future self.

If minds are distinct from entities, then the possible relationships between the set of entities and set of minds include:

1. Each entity is successively inhabited by each mind, i.e., the minds rotate through entities.
2. Each entity has is own unique mind.
3. There is one mind which inhabits each entity successively, i.e., the mind rapidly rotates through the entities giving the appearance of multiple intelligent entities. This is the case in the typical operating system.

References

Allen, Varner, and Zinser *Prolegomena to any future artificial moral agent* Journal of Experimental & Theoretical Artificial Intelligence Volume: 12 Number: 3 Page: 251 -- 261
Asimov, I. (1968). *The rest of robots*. London: Granada 1968.
Gorniak-Kocicowska, Krystyna *The Computer Revolution and the Problem of Global Ethics* The Research Center on Computing & Society at Southern Connecticut State University 2000.
Helmers, Hoffmann, and Stamos-Kaschke *(How) Can Software Agents Become Good Net Citizens?* CMC Magazine, Vol. 3, No. 2, Feb. 1997
Hoffmann, Robert (2000) *Twenty Years on: The Evolution of Cooperation Revisited* Journal of artificial Societies and Social Simulations vol. 3 no. 2,
Koster, Martijn (1993) *Guidelines for Robot Writers* URL: info.webcrawler.com/mak/projects/robots/robots.html
Miculan, Alison R. *Ethics and Reality* 20th World Philosophical Congress
Moor, James. *Is Ethics Computable?* Metaphilosophy 26, nos. 1-2 (January-April): 1-21.
Sandip Sen, `` Reciprocity: a foundational principle for promoting cooperative behavior among self-interested agents"" , in *Proc. of the Second International Conference on Multiagent Systems*, pages 322--329, AAAI Press, Menlo Park, CA, 1996.
Shoham, Yoav and Tennenholtz, Moshe. "On social laws for artificial agent societies" Artificial Intelligence vol 73.
Smullyan, Raymond (1977). *The Tao is silent*. Harper & Row 1977.
Smullyan, Raymond (1983). *5000 B. C. and other Philosophical Fantasies* St. Martins Press 1983.
Stanley, Maurice F. *The Geometry of Ethics* 20th World Philosophical Congress
Whitehead, A. N. *Process and Reality* Macmillan 1978.
Whitehead, A. N. *Religion in the Making* New American Library 1974.
Williams, Patricia A. *Christianity and Evolutionary Ethics: Sketch Toward a Reconciliation* Zygon, vol. 31, no. 2 (June 1996)

- Last Modified - . Comments and content invited aabyan@wwc.edu

summary - ethics vs mathematics

Ethics: survey and observations

Anthony Aaby
Walla Walla College
aabyan@wwc.edu

Document status: working notes
Distribution: private

Last Modified: .
Comments and content invited: aabyan@wwc.edu

0 Introduction

The purpose of this document is to provide a personalized summary of ethical theory in order to form a base for answering the question "What does ethical theory have to contribute to software design and vice versa?". In software engineering terms, this summary focuses on an "architectural view" rather than an "implementation view" of ethics.

The remainder of this paper is structured as follows. Section 1 is a short review of ethics with short descriptions of metaethics and the prominent categories of normative ethics. Section 2 is a short review of the fundamental principles of theories. Section 3 contains some observations on ethical theories from the perspective of mathematical logic. Section 4 outlines some ideas on how to construct ethical systems for artificial societies. An Appendix follows with a variety of short notes supplying additional information on a variety of ethical theories.

1 A short review of ethics

The field of ethics involves systematizing, defending, and recommending concepts of right and wrong behavior. Ethics is divided into three general areas, metaethics, normative ethics, and applied ethics. *Metaethics* investigates the source and meaning of ethical principles. *Normative ethics* describe the ethical standards that regulate right and wrong. *Applied ethics* resolve specific ethical controversies using the tools of metaethics and normative ethics. The next two subsections review metaethics and normative ethics.

Metaethics

The task of metaethics is to determine the set of entities and behaviors of interest and to determine the source of ethical values. Traditional sources of ethical values identified in metaethical argument may be divided into objective or subjective sources. Objective sources include divine commands and part of the fundamental nature of the universe. Subjective sources include the individual and culture. The fundamental problem of metaethics is to provide a complete explanation of both ethical and unethical behavior and the historical and contemporary variations in ethical values and practice.

The fundamental problem of metaethics

> The fundamental problem of metaethics is to provide a complete explanation of both ethical and unethical behavior and the historical and contemporary variations in ethical values and practice.

Normative ethics

Normative ethics involves arriving at the ethical standards that regulate conduct. The key assumption is that there is only one ultimate criterion of ethical conduct whether it is a single rule or set of principles. It is common to classify ethical theories into several categories:

1. axiological (virtue) theories,
2. deontological theories, and
3. consequentialist theories.

In addition to determining ethical behavior, an ethical theory should prescribe a method for resolving ethical conflicts/dilemmas should any arise in an application of the theory.

Virtue theory suggests that ethical behavior is the result of virtues or good habits of character. Commonly suggested virtues include: wisdom, courage, temperance, justice, fortitude, generosity, self-respect, good temper, and sincerity. Vices or negative virtues include cowardice, insensibility, injustice, and vanity.

Deontological theories identify various duties and rights. Duties and obligations have been classified under several categories including

- duties to God,
- duties to oneself, and
- duties to others which include
    - duties to family,
    - social duties, and
    - political duties.

The basic rights include life, liberty and the pursuit of happiness and are considered to be natural, universal, equal, and inalienable. The focus of deontological theories is on moral duties or obligations rather than on moral value or goodness. Intentions play a significant role in determining whether an act is ethical.

Categorical imperative: Kant formulated the notion of a categorical imperative which requires that actions toward another entity should reflect the value of that entity. There are three forms of the categorical imperative.

1. *Categorical Imperative:* Act only on that maxim whereby you can at the same time will that it would become a universal law.
2. *Principle of ends:* Act so that you treat humanity never as a mere means to an end, but always as an end in themselves.
3. *Principle of Autonomy:* Every rational being is able to regard itself as a maker of universal law and everyone who is ideally rational will legislate exactly the same universal principles.

Consequentialist (teleological) theories determine ethical behavior by weighing the consequences of an action. The good and bad consequences of an action are tallied and if the total good consequences outweigh the total bad consequences, then the action is ethically proper. Thus an action is ethical if the consequences of that action are more favorable than unfavorable with respect to some criteria. Criteria include affected groups and the dimension of time. Three criteria with respect to agents have been suggested.

- *Ethical egoism:* only *consequences to the entity* performing the action are considered.
- *Ethical altruism:* only *consequences to everyone except the agent* performing the action are considered.
- *Utilitarianism:* the *consequences to everyone of an act or rule* are considered.

In the dimension of time, the influence of action may extend beyond the immediate consequences of the act. The focus on consequences is problematic since consequences are, in almost all cases, outside the agent's immediate and direct control and may be unanticipated due to lack of omniscience.

The focus is on moral value or goodness rather than on moral duties or obligations. An action's consequences (what is good) are more important than on moral obligations (what is right). Human nature and experience determine what the good is. *Social contract theory* is a consequentialist theory in which morality is defined by a set of rules accepted by rational people for their mutual benefit.

2 Fundamental principles of theories

...
understanding ... organization ...
consensus ... impartial ... consistent ...
... through careful use of generalization and atraction ...
... distinguish between particular cases and universal ...
... careful exploration of boundary conditions and exceptional situations ...
... conclusions are reproduceable since they are rationally derived from a commonly accepted base ...
... meaning ... a mapping between syntactic statements and a semantic structure.

A theory provides systematic body of knowledge that otherwise exists as a disorganized collection of

rules and an objective rather than subjective system where rational agents will arrive at the same conclusions.

Each theory consists of a language suitable for the domain of interest, a logic with rules of inference, and a collection of domain assumptions (axioms).

Questions to ask about any theory.

- Is the set of axioms independent (each axiom is necessary)?
- Are the inference rules sound (if *a* results from an inference, then *a* is in fact true)?
- Is the theory
  - consistent (cannot conclude both *a* and ~*a*) and
  - complete (for any formula *a*, ether *a* or ~*a* holds)?
- What is the complexity of the theory?
  - Is it decidable?
- What does the theory mean? i.e. what is the correspondence between the theory and some world?

Principles that are prerequisite to any theory.

1. *Principle of rationality:* all conclusions must be supported by generally accepted reasons.
2. *Principle of soundness:*
3. *Principle of consistency:* theory should be consistent - don't conclude both *a* and ~*a*.
4. *Principle of least harm:* choose the lesser of two evils
5. *Principle of impartiality:* theory should provide equal treatment for equal situations - if *a* and *a'* are equivalent and *P*(*a*) holds, then so does *P*(*a'*).
6. *Principle of substitution:* if *a* and *a'* are equivalent and *P*(*a*) holds, then so does *P*(*a'*).

3 Observations

General observations

Ethical theories have not reached the level of formalism present in mathematical theories. This lack of formalization makes it difficult to assess the applicability of an ethical theory in novel situations and increases the difficulty in identifying the similarity and differences with other (non ethical) theories.

Virtue theory

Virtue theory bears some resemblance to high level attributes of well-engineered software. The attributes of well engineered software include: maintainability, dependability, efficiency, and usability.

Deontological theory

Deontological theories with lists of rights and duties bear some resemblance to safety and liveness specifications. Kant's categorical imperative seems to be of little use in software design as often the design consists of heterogeneous processes with little behavior in common.

Teleological theory

Consequentalist theories bear some resemblance to the software engineering practice of data collection and testing to determine an acceptable solution. Social contract theory bears a strong resemblance to the requirements and specification documents produced in the software engineering process.

Other observations

Normative ethics are

- rules for resource mangement
- rules to maximize predictability
- rules to define a society or group

... Christian ethics ...

4 Ethical systems for artificial societies

Comment: Ideal ethical systems are assumed to be universal and eternal rather than local and evolving.
Comment: System requirements and specifications roughly correspond to an ethical theory.
Comment: Virtue theory lacks prescriptive value for software design i.e., its not clear how virtues translate to behaviors in computer systems.. Some commonly accepted virtues for software include:

- correctness,
- efficiency,
- modularity
- robust (fault tolerant)
- ...

Correctness	Completeness and correctness of solution Static type safety, dynamic type safety Multithreaded safety, liveness Fault tolerance, transactionality Security, robustness

Resources	Efficiency: performance, time complexity, number of messages sent, bandwidth requirements Space utilization: number of memory cells, objects, threads, processes, communication channels, processors, ... Incrementalness (on-demand-ness) Policy dynamics: Fairness, equilibrium, stability
Structure	Modularity, encapsulation, coupling, independence Extensibility: subclassibility, tunability, evolvability, maintainability Reusability, openness, composibility, portability, embeddability Context dependence Interoperability ... other ``ilities'' and ``quality factors''
Construction	Understandability, minimality, simplicity, elegance. Error-proneness of implementation Coexistence with other software Maintainability Impact on/of development process Impact on/of development team structure and dynamics Impact on/of user participation Impact on/of productivity, scheduling, cost
Usage	Ethics of use Human factors: learnability, undoability, ... Adaptability to a changing world Aesthetics Medical and environmental impact Social, economic and political impact ... other impact on human existence

Comment: For a deontological theory to be useful for software design, an appropriate list of duties rights and obligations include:

- acceptance of the decisions of the OS,
- faithful to specification, and
- non interference with other processes.

Comment: Teleological (consequentialist) theories are used in operating system design for determining a scheduling policy.

- efficiency
  - maximize throughput
  - minimize waiting time

Comment: The values approach is used in user interface design - user friendly ...

Comment: None of the ethical theories is as compelling with respect to truth and universality as mathematics. Each category of ethical theories has something to contribute to ethical decision making. They are not necessarily mutually exclusive theories. The collection of theories is a resource, a collection of tools, to be used as needed and when appropriate. The designer of an artificial society is free to select any ethical system for the society. In societies with a mix of human and autonomous agents, **...** From virtue theories we learn to have an even broader perspective than behavior. From consequentialist theories we learn to consider the consequences of an action. From deontological theories we learn to consider rights, duties, and obligations. From relativistic theories we learn how to construct purpose built ethical systems.

From deontological ethics, the language of "rights" and "obligations" in the context of the process of the design of a cooperative society seems to capture ...

From teleological ethics, the language of "consequences" ...

formal specifications and proofs of correctness

deterministic behavior of agents

ability to collect information from carefully designed experiments and simulations ameliorates the lack of omniscience.

For the purposes of this section, general purpose operating systems and computer networks are taken to be examples of primitive artificial societies.

Metaethics

The task of metaethics is to determine the set of entities and behaviors of interest and to determine the source of ethical values. For software systems, the customer and software engineer determine the entities and behaviors of interest thus are the source of ethical values.

The high rate of evolution of software systems requires the system to be flexible

Normative ethics

Formally, an ethical system is a mapping between *V*: *F*x*S*-> {bad, good} or {wrong, right}

Rights and liabilities

Rights, liberties, powers, and immunities are all kinds of "rights." Any right implies certain duties, liabilities, or disabilities in others. Each kind of right implies a certain kind of liability in others, and each kind of right also has its opposite form of liability.

Right: to goods and services.
Liberty: a right to act without restraint.
Power: the ability to change the status of something or force a compliance in another.
Immunity: is an exemption from being subject to someone else's powers.

Duty: an obligation to act
No rights:
Liability: must recognize or comply with the power exercised upon them.
Disability: without a power to affect the immune person.

The opposite of a "duty" is a *liberty*, which means that there are no rights of others that need be observed in a particular case. A "power" is the ability to change the legal status of something or force a legal compliance in another. A power thus implies a *liability* in another, that they must recognize or comply with the power exercised upon them. The opposite of a "liability" is an *immunity*, which is an exemption from being subject to someone else's powers. An "immunity" implies a *disability* in another, that they are *without* a power to affect the immune person in that case.

Rights			
Right (may do)	Liberty (may do)	Power (can force)	Immunity (can resist)
Duty (must do)	No right (may not do)	Liability (can not resist)	Disability (cannot force)
Liabilities			

	Negative	**Positive**
Rights	?	liberty
Duties	disability (must not do)	liability (must do)

Rights	
Right	Liberty
Duty	No right
Liabilities	

Rights	
Power	Immunity
Liability	Disability (cannot do)
Liabilities	

DUTIES - Rights are often discussed in terms of being entailed by duties. So that for every right, there is usually either a positive duty or a negative duty (or both) that comes with having the right. For instance, the right to life might be said to have the negative duty to refrain from taking other people's lives, or maybe even the positive duty to help protect people's lives.

- Positive Duties - duties to do something.

- Negative Duties - duties to refrain from doing something.

Appendix - miscellaneous notes

Moral dilemma

A *moral dilemma* involves a situation in which the agent has only two courses of action available, and each requires performing a morally impermissible action.

Situation *S* requires action $A \vee B$ but neither is morally permissible i.e.
$E \vdash \sim A$ and $E \vdash \sim B$

Communitarianism

The dominant themes of communitarianism are that individual rights need to be balanced with social responsibilities, and that autonomous selves do not exist in isolation, but are shaped by the values and culture of communities.

Deontological vs teleological theories

The following table contrasts the features of deontological and teleological theories.

DEONTOLOGICAL THEORIES	TELEOLOGICAL THEORIES
1. The focus is on moral duties (what is right). 2. Moral duties are more important. 3. The individual's intentions have a substantial role in a situation's moral evaluation and consequences that arise through the individual's actions have no relevance. 4. There is no one specifiable relation between good and right. 5. Concepts about moral value (i.e., what is good) are definable in reference to concepts about moral duties (i.e., what is right). 6. The right is prior to the good. 7. An action's goodness (or value) depends upon the action's rightness. 8. It is the individual's moral status that is important. 9. The statement 'x is a moral individual' means 'x did what was right with the right	1. The focus is on an action's consequences (what is good). 2. Moral values are more important. 3. The consequences that an individual's actions produce have a substantial role in a situation's moral evaluation and the individual's intentions have no relevance. 4. There is a specifiable relation between good and right. 5. Concepts about moral duties (i.e., what is right) are definable in reference to concepts about moral value (i.e., what is good). 6. The good is prior to the right. 7. An action's rightness depends upon the action's goodness (or value). 8. It is the action's moral status that is important.

intention'. 10. Reason, intuition or moral sense reveals what is right. 11. There are some acts that are moral or immoral in themselves. 12. Moral duties have a negative formulation. 13. One's own moral duties have precedence over all other considerations. 14. To do what is moral (i.e., right) requires that one observe one's moral duties, possess the right intentions and avoid those actions that are immoral in themselves.	9. The statement 'x is a moral action' means 'x produces at least as good consequences as all other possible actions'. 10. Experience reveals what is good. 11. There are no actions that are moral or immoral in themselves. 12. Moral duties have a positive formulation. 13. One must give equal and impartial consideration to other's interests and happiness, as well as one's own, in all moral considerations and evaluations. 14. To do what is moral (i.e., good) requires that one acts so as to maximize the happiness that one's action produce.

Adapted from a lost source.

Evolutionary ethics

Evolutionary ethics seeks to ground ethical behavior in evolutionary theory. For example, sociobiology finds evolutionary evidence for altruism toward kin and reciprocity toward non kin.

Relativism

The most famous statement of relativism in general is by the ancient Greek sophist Protagoras (480-411 BCE.): "A human being is the measure of all things - of things that are, that they are, and of things that are not that they are not." This reflects the view of many of the sophists that social convention has a status above nature. Although Protagoras's claim applies to any proposed standard of knowledge, moral values are at least part of his position. Most philosophers have assumed that there is some standpoint--for example, that of God--in relation to which our judgments are definitively true or false. Relativism is sometimes identified (usually by its critics) as the thesis that all points of view are equally valid. In ethics, this amounts to saying that all moralities are equally good; in epistemology it implies that all beliefs, or belief systems, are equally true.

Relativistic ethical theories overlap the previous three categories of ethical theories. They reject any ethical rule as universal or absolute, assert that ethical standards are grounded only in social custom, and that there is no objective way to assess the validity of ethical principles. In particular,

1. ethical value are relative to some particular framework or standpoint (e.g. the individual subject, a culture, an era, a language, or a conceptual scheme), and
2. no standpoint is uniquely privileged over all others.

Relativism and its critics

Relativism	**Its critics.**
The key features of relativism are 1. *Something* (e.g. moral values, beauty, knowledge, taste, or meaning) *is relative* to some *particular framework* or standpoint (e.g. the individual subject, a culture, an era, a language, or a conceptual scheme). 2. No standpoint is uniquely privileged over all others. Relativism is sometimes identified (usually by its critics) as the thesis that all points of view are equally valid.	There is some standpoint -- for example, that of God or science -- in relation to which our judgments are definitively true or false.
It seems to me that while the first point is worthwhile, the second is too strong. It seems that it should be possible through comparative ethical studies to show some sort of ordering among systems is possible and that one system is perhaps better for some purpose than another.	
Counter argument: While relativism may indeed be false from certain perspectives, these are not perspectives that consistent relativists will be committed to. In fact, of those who accept the major paradigm shifts that have characterized philosophy over the last two centuries, relativists can claim to be the most consistent, since they alone accept the full implications of these shifts for our notions of truth and rationality.	*Objection:* Relativism is incoherent and self-refuting. It is pernicious since it undermines the enterprise of trying to improve our ways of thinking. • The relativist (from a relativist argument) must concede that from some points of view relativism will appear false. • Moreover, since no standpoint is uniquely privileged, these standpoints, and the views they encompass or imply, are equally worthy of our respect. • The relativist must therefore hold that relativism is both true and false.
The problem with the objection seems to stem from the use of two valued logic. An infinite valued logic permits statements to have intermediate truth values.	
Counter argument: Relativists do not have to commit themselves to any non-relativistic notion of truth. It is possible to advance a claim and hold it to be true relative to a given set of norms, without committing oneself to the view that it is true, or that the norms in question are valid, in some further, non-relativistic sense.	*Objection:* If all judgments are only true relative to some non-privileged standpoint, then relativism is true in some non-relativistic sense.

Moral relativism: morality is grounded in social custom rather than some prescriptive ideal. 1. Primacy of De Facto Values - morality should be based on how people actually behave. 2. Cultural variation - main moral values vary from culture to culture .	*Objection:* There is a core set of values that is common to all societies and is in fact necessary for any society to exist. These values are that 1. we should care for children, 2. we should tell the truth, and 3. we should not murder. If moral relativism is true, we cannot argue that customs such as slavery are morally inferior. (James Rachels)
It seems to me that the proper starting point is to describe the range of behaviors available to an entity in a society and the necessary and sufficient conditions for the society to exist. When societies come in contact with each other, ... conflict	
Cognitive relativism: truth is relativized is usually understood to be a conceptual scheme. No one set of epistemic norms is metaphysically privileged over any other. *Counter argument:* prove the superiority of the preferred epistemic norm.	*Objection:* The epistemic norms such as employed by modern science enjoy special status and can serve as objective, universally valid, criteria of truth and rationality. *Counter counter argument:* The success of modern science is sufficient proof.
It seems that it should be possible through comparative studies to show some sort of ordering among systems is possible and to demonstrate that one system is perhaps better for some purpose than another. In fact, there are contemporary and historical variations within the scientific community.	

Adapted from the Internet Encyclopedia of Philosophy

Social contract/Contractarian theory principles

Fundamental principle: Ethics is founded solely on uniform social agreements that serve the best interests of those who make the agreement.

Initial state:

Hobbes	**Locke**	**Rousseau**	**Rawls**

constant war	pre-political, moral, bound by natural law with inalienable rights - life, liberty, & property	individual freedom & creativity

Nature of mankind

Hobbes	Locke	Rousseau	Rawls
motivated by selfish interest		social person	rational and impartial

Resources

Contract constructed by

Hobbes	Locke	Rousseau	Rawls
the rational and self-motivated		the social persons	the rational and impartial

Purpose of contract (good is ...)

Hobbes	Locke	Rousseau	Rawls
to preserve peace	security in life, liberty & property	regulate social interactions	mutual benefit

Type of government

Hobbes	Locke	Rousseau	Rawls
absolute monarchy	democracy	absolute democracy	

http://www.opencontent.org).

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Some ethical principles

Anthony Aaby
Walla Walla College
aabyan@wwc.edu

Document status: working notes
Distribution: private

Last Modified: .
Comments and content invited: aabyan@wwc.edu

1. Principle of Autonomy: freedom of choice and action.
2. Principle of Beneficence: obligation to do good for others.
3. Principle of the Categorical Imperative: Kant; There are three forms of the categorical imperative.
    1. Categorical Imperative: Act only on that maxim whereby you can at the same time will that it would become a universal law.
    2. Principle of ends: Act so that you treat humanity never as a mere means to an end, but always as an end in themselves.
    3. Principle of Autonomy: Every rational being is able to regard itself as a maker of universal law and everyone who is ideally rational will legislate exactly the same universal principles.
4. Principle of Equality (Justice): accord to each an equality of respect/treatment.
5. Principle of Fidelity: obligation to keep promises.
6. Principle of Honesty: obligation to tell the truth.
7. Principle of Justice: John Rawls; Fair distribution of benefits and burdens.
    - Principle of Equal Liberty:
    - Principle of Difference: There will be inequalities, but we are morally obligated to improve the worst off unless it would make everyone worse off.
    - Principle of Fair Equality of Opportunity: Requires that job qualifications be related to the job.
8. Principle of Least Harm: choose the lesser of two evils.
9. Principle of Nonmaleficence: obligation to do no harm.
10. Principle of Rights: Immanuel Kant; Right to free and equal treatment.
11. Principle of Social Contract: morality consists in a set of rules, that rational people will agree to accept, for their mutual benefit, on the condition that others follow those rules as well and that these rules benefit the least advantaged in society.
    1. Principle of liberty: equal right to the most extensive scheme of liberties compatible with a similar scheme of liberties for all.

2. Principle of opportunity: there must be equality of opportunity for individuals in compostion for those positions in society that bring greater rewards.
3. Principle of distributive justice: basic goods are distributed so that the least advantaged members of society benefit as much as possible.
4. Principle of justice: each possesses an inviolability founded on justice that society cannot override.
5. Principle of need: each is guaranteed the primary goods that are necessary assuming that there are sufficient resources to maintain the guaranteed minimum.

12. Principle of Utility: John Stuart Mill; the value of an act depends on whether it increases or decreases the amount of happyiness/good/pleasure of the party whose interest is in question.
    - Act-utilitiarianism: An act is right iff it results in as much good as any viable alternative.
    - Rule-utilitarianism: An act is right iff it is required by a rule that itself is a member of a set of rules, the acceptance of which would lead to greater good for society than any available alternative.
    - Principle of non-interference: Society is justified in coercing the behavior of an individual in order to prevent her/him from injuring others; it is not justified in coercing her/him simply because the behavior is immoral or harmful to herself/himself.
    - Principles of Consequences: In assessing consequences, the only thing that matters is the amount of happiness/good or unhappiness/bad that is caused. The right actions are those that produce the greatest amount of good over bad.

Commonly expressed unethical rules

- Principle of Ends: The end justifies the means.
- Principle of Might: Might makes right.
- Principle of Rights: Everyone has the right to do whatever s/he wants.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Miscellaneous notes

Anthony Aaby
Walla Walla College
aabyan@wwc.edu

Document status: working notes
Distribution: private

Last Modified: .
Comments and content invited: aabyan@wwc.edu

Distinguish between:

- What we can do.
- What we desire to do.
- What we ought to do.

Autonomous software agents

Design considerations:

- Independent: process cannot affect or be affected by the other processes
- Dependent: processes can affect or be affected by the other processes. Possible to deadlock or starve. There are several subcategories
    - cooperating -- shared task and possibly shared resources
    - competing -- may starve opponent
    - hostile -- attempt to destroy an other's resources

Laws which

- describe possible actions
- prevent ...
- guarantee the successful coexistence of multiple agents

> **Proposition 1:** If two entities have incompatible wants, they may interfere with each other.

Proof:

Proposition 2: Two entities of equal strength and incompatible wants, can survive only through the fear of mutually assured destruction.

Proof:

...what is good for the individual and for society...

Proposition 3: The principle of rights is necessary for an objective code of morality.

Proof: Suppose there is an entity which does not have the right to do whatever it wants. I argue that the previous sentence is a contradiction. As has been pointed out in the discussion of the principle of rights, an entity's behavior may be constrained by other entities but cannot be the case here. It must refer to a behavior that it wants to do and is capable of but cannot do independently of constraints that might be imposed on it by other entities. That is, its behavior is self constrained. It both wants to do something and does not want to do the same thing. A contradiction.

Discussion: It might be argued that both wanting and not wanting to do something is not a contradiction it is just the natural aspect of exercising choice between mutually exclusive alternatives. If so, then we must adopt a multivalued logic to describe behavior rather than the traditional two valued logic.

It also might be argued that contradiction is necessary for the existence of free will. And therefore both good and evil must coexist. Some religious traditions recognize this and argue that either the coexistence of good and evil is the natural state of the universe or that at some future point, all evil will be eliminated and with it, all free choice.

It might be argued that the proof is bogus because it permits only external constraints to constrain an entity from certain behaviors that is, an entity wants to do something but chooses not to do it. I argue that its choice demonstrates what it wants.

Last Modified - . Comments and content invited aabyan@wwc.edu

CS Body of Knowledge

from: Computing Curricula 2001

Computer science body of knowledge with core topics underlined and minimum number of lecture hours in parentheses. Select *Details* links to see content.

DS. Discrete Structures (43 core hours)

- DS1. Functions, relations, and sets (6)
- DS2. Basic logic (10)
- DS3. Proof techniques (12)
- DS4. Basics of counting (5)
- DS5. Graphs and trees (4)
- DS6. Discrete probability (6)

PF. Programming Fundamentals (38 core hours)

- PF1. Fundamental programming constructs (9)
- PF2. Algorithms and problem-solving (6)
- PF3. Fundamental data structures (14)
- PF4. Recursion (5)
- PF5. Event-driven programming (4)

AL. Algorithms and Complexity (31 core hours)

- AL1. Basic algorithmic analysis (4)
- AL2. Algorithmic strategies (6)
- AL3. Fundamental computing algorithms (12)
- AL4. Distributed algorithms (3)
- AL5. Basic computability (6)
- AL6. The complexity classes P and NP
- AL7. Automata theory
- AL8. Advanced algorithmic analysis
- AL9. Cryptographic algorithms
- AL10. Geometric algorithms
- AL11. Parallel algorithms

AR. Architecture and Organization (36 core hours)

- AR1. Digital logic and digital systems (6)
- AR2. Machine level representation of data (3)
- AR3. Assembly level machine organization (9)
- AR4. Memory system organization and architecture (5)
- AR5. Interfacing and communication (3)
- AR6. Functional organization (7)

HC. Human-Computer Interaction (8 core hours)

- HC1. Foundations of human-computer interaction (6)
- HC2. Building a simple graphical user interface (2)
- HC3. Human-centered software evaluation
- HC4. Human-centered software development
- HC5. Graphical user-interface design
- HC6. Graphical user-interface programming
- HC7. HCI aspects of multimedia systems
- HC8. HCI aspects of collaboration and communication

GV. Graphics and Visual Computing (3 core hours)

- GV1. Fundamental techniques in graphics (2)
- GV2. Graphic systems (1)
- GV3. Graphic communication
- GV4. Geometric modeling
- GV5. Basic rendering
- GV6. Advanced rendering
- GV7. Advanced techniques
- GV8. Computer animation
- GV9. Visualization
- GV10. Virtual reality
- GV11. Computer vision

IS. Intelligent Systems (10 core hours)

- IS1. Fundamental issues in intelligent systems (1)
- IS2. Search and constraint satisfaction (5)
- IS3. Knowledge representation and reasoning (4)
- IS4. Advanced search
- IS5. Advanced knowledge representation and reasoning
- IS6. Agents
- IS7. Natural language processing

AR7. Multiprocessing and alternative architectures (3)
AR8. Performance enhancements
AR9. Architecture for networks and distributed systems

OS. Operating Systems (18 core hours)

OS1. Overview of operating systems (2)
OS2. Operating system principles (2)
OS3. Concurrency (6)
OS4. Scheduling and dispatch (3)
OS5. Memory management (5)
OS6. Device management
OS7. Security and protection
OS8. File systems
OS9. Real-time and embedded systems
OS10. Fault tolerance
OS11. System performance evaluation
OS12. Scripting

NC. Net-Centric Computing (15 core hours)

NC1. Introduction to net-centric computing (2)
NC2. Communication and networking (7)
NC3. Network security (3)
NC4. The web as an example of client-server computing (3)
NC5. Building web applications
NC6. Network management
NC7. Compression and decompression
NC8. Multimedia data technologies
NC9. Wireless and mobile computing

PL. Programming Languages (21 core hours)

PL1. Overview of programming languages (2)
PL2. Virtual machines (1)
PL3. Introduction to language translation (2)
PL4. Declarations and types (3)
PL5. Abstraction mechanisms (3)
PL6. Object-oriented programming (10)
PL7. Functional programming
PL8. Language translation systems
PL9. Type systems
PL10. Programming language semantics
PL11. Programming language design

IS8. Machine learning and neural networks
IS9. AI planning systems
IS10. Robotics

IM. Information Management (10 core hours)

IM1. Information models and systems (3)
IM2. Database systems (3)
IM3. Data modeling (4)
IM4. Relational databases
IM5. Database query languages
IM6. Relational database design
IM7. Transaction processing
IM8. Distributed databases
IM9. Physical database design
IM10. Data mining
IM11. Information storage and retrieval
IM12. Hypertext and hypermedia
IM13. Multimedia information and systems
IM14. Digital libraries

SP. Social and Professional Issues (16 core hours)

SP1. History of computing (1)
SP2. Social context of computing (3)
SP3. Methods and tools of analysis (2)
SP4. Professional and ethical responsibilities (3)
SP5. Risks and liabilities of computer-based systems (2)
SP6. Intellectual property (3)
SP7. Privacy and civil liberties (2)
SP8. Computer crime
SP9. Economic issues in computing
SP10. Philosophical frameworks

SE. Software Engineering (31 core hours)

SE1. Software design (8)
SE2. Using APIs (5)
SE3. Software tools and environments (3)
SE4. Software processes (2)
SE5. Software requirements and specifications (4)
SE6. Software validation (3)
SE7. Software evolution (3)
SE8. Software project management (3)
SE9. Component-based computing
SE10. Formal methods
SE11. Software reliability

	SE12. Specialized systems development **CN. Computational Science and Numerical Methods (no core hours)** CN1. Numerical analysis CN2. Operations research CN3. Modeling and simulation CN4. High-performance computing

Discrete Structures (DS)

Discrete structures is foundational material for computer science. By *foundational* we mean that relatively few computer scientists will be working primarily on discrete structures, but that many other areas of computer science require the ability to work with concepts from discrete structures. Discrete structures includes important material from such areas as set theory, logic, graph theory, and combinatorics.

The material in discrete structures is pervasive in the areas of data structures and algorithms but appears elsewhere in computer science as well. For example, an ability to create and understand a formal proof is essential in formal specification, in verification, and in cryptography. Graph theory concepts are used in networks, operating systems, and compilers. Set theory concepts are used in software engineering and in databases.

As the field of computer science matures, more and more sophisticated analysis techniques are being brought to bear on practical problems. To understand the computational techniques of the future, today's students will need a strong background in discrete structures.

Finally, we note that while areas often have somewhat fuzzy boundaries, this is especially true for discrete structures. We have gathered together here a body of material of a mathematical nature that computer science education must include, and that computer science educators know well enough to specify in great detail. However, the decision about where to draw the line between this area and the Algorithms and Complexity area (AL) on the one hand, and topics left only as supporting mathematics on the other hand, was inevitably somewhat arbitrary. We remind readers that there are vital topics from those two areas that some schools will include in courses with titles like discrete structures.

DS1. Functions, relations, and sets [core]

Minimum core coverage time: 6 hours

Topics:

- Functions (surjections, injections, inverses, composition)
- Relations (reflexivity, symmetry, transitivity, equivalence relations)
- Sets (Venn diagrams, complements, Cartesian products, power sets)
- Pigeonhole principle

- Cardinality and countability

Learning objectives:

1. Explain with examples the basic terminology of functions, relations, and sets.
2. Perform the operations associated with sets, functions, and relations.
3. Relate practical examples to the appropriate set, function, or relation model, and interpret the associated operations and terminology in context.
4. Demonstrate basic counting principles, including uses of diagonalization and the pigeonhole principle.

DS2. Basic logic [core]

Minimum core coverage time: 10 hours

Topics:

- Propositional logic
- Logical connectives
- Truth tables
- Normal forms (conjunctive and disjunctive)
- Validity
- Predicate logic
- Universal and existential quantification
- Modus ponens and modus tollens
- Limitations of predicate logic

Learning objectives:

1. Apply formal methods of symbolic propositional and predicate logic.
2. Describe how formal tools of symbolic logic are used to model algorithms and real-life situations.
3. Use formal logic proofs and logical reasoning to solve problems such as puzzles.
4. Describe the importance and limitations of predicate logic.

DS3. Proof techniques [core]

Minimum core coverage time: 12 hours

Topics:

- Notions of implication, converse, inverse, contrapositive, negation, and contradiction
- The structure of formal proofs
- Direct proofs
- Proof by counterexample

- Proof by contraposition
- Proof by contradiction
- Mathematical induction
- Strong induction
- Recursive mathematical definitions
- Well orderings

Learning objectives:

1. Outline the basic structure of and give examples of each proof technique described in this unit.
2. Discuss which type of proof is best for a given problem.
3. Relate the ideas of mathematical induction to recursion and recursively defined structures.
4. Identify the difference between mathematical and strong induction and give examples of the appropriate use of each.

DS4. Basics of counting [core]

Minimum core coverage time: 5 hours

Topics:

- Counting arguments
    - Sum and product rule
    - Inclusion-exclusion principle
    - Arithmetic and geometric progressions
    - Fibonacci numbers
- The pigeonhole principle
- Permutations and combinations
    - Basic definitions
    - Pascal's identity
    - The binomial theorem
- Solving recurrence relations
    - Common examples
    - The Master theorem

Learning objectives:

1. Compute permutations and combinations of a set, and interpret the meaning in the context of the particular application.
2. State the definition of the Master theorem.
3. Solve a variety of basic recurrence equations.
4. Analyze a problem to create relevant recurrence equations or to identify important counting questions.

DS5. Graphs and trees [core]

Minimum core coverage time: 4 hours

Topics:

- Trees
- Undirected graphs
- Directed graphs
- Spanning trees
- Traversal strategies

Learning objectives:

1. Illustrate by example the basic terminology of graph theory, and some of the properties and special cases of each.
2. Demonstrate different traversal methods for trees and graphs.
3. Model problems in computer science using graphs and trees.
4. Relate graphs and trees to data structures, algorithms, and counting.

DS6. Discrete probability [core]

Minimum core coverage time: 6 hours

Topics:

- Finite probability space, probability measure, events
- Conditional probability, independence, Bayes' theorem
- Integer random variables, expectation

Learning objectives:

1. Calculate probabilities of events and expectations of random variables for elementary problems such as games of chance.
2. Differentiate between dependent and independent events.
3. Apply the binomial theorem to independent events and Bayes theorem to dependent events.
4. Apply the tools of probability to solve problems such as the Monte Carlo method, the average case analysis of algorithms, and hashing.

Programming Fundamentals (PF)

Fluency in a programming language is prerequisite to the study of most of computer science. In the CC1991 report, knowledge of a programming language -- while identified as essential -- was given little emphasis in the curriculum. The "Introduction to a Programming Language" area in CC1991 represents only 12 hours of class time and is identified as optional, under the optimistic assumption that "increasing numbers of students . . . gain such experience in secondary school." We believe that undergraduate computer science programs must teach students how to use at least one programming language well; furthermore, we recommend that computer science programs should teach students to become competent in languages that make use of at least two programming paradigms. Accomplishing this goal requires considerably more than 12 hours.

This knowledge area consists of those skills and concepts that are essential to programming practice independent of the underlying paradigm. As a result, this area includes units on fundamental programming concepts, basic data structures, and algorithmic processes. These units, however, by no means cover the full range of programming knowledge that a computer science undergraduate must know. Many of the other areas -- most notably Programming Languages (PL) and Software Engineering (SE) -- also contain programming-related units that are part of the undergraduate core. In most cases, these units could equally well have been assigned to either Programming Fundamentals or the more advanced area.

PF1. Fundamental programming constructs [core]

Minimum core coverage time: 9 hours

Topics:

- Basic syntax and semantics of a higher-level language
- Variables, types, expressions, and assignment
- Simple I/O
- Conditional and iterative control structures
- Functions and parameter passing
- Structured decomposition

Learning objectives:

1. Analyze and explain the behavior of simple programs involving the fundamental programming constructs covered by this unit.

2. Modify and expand short programs that use standard conditional and iterative control structures and functions.
3. Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, and the definition of functions.
4. Choose appropriate conditional and iteration constructs for a given programming task.
5. Apply the techniques of structured (functional) decomposition to break a program into smaller pieces.
6. Describe the mechanics of parameter passing.

PF2. Algorithms and problem-solving [core]

Minimum core coverage time: 6 hours

Topics:

- Problem-solving strategies
- The role of algorithms in the problem-solving process
- Implementation strategies for algorithms
- Debugging strategies
- The concept and properties of algorithms

Learning objectives:

1. Discuss the importance of algorithms in the problem-solving process.
2. Identify the necessary properties of good algorithms.
3. Create algorithms for solving simple problems.
4. Use pseudocode or a programming language to implement, test, and debug algorithms for solving simple problems.
5. Describe strategies that are useful in debugging.

PF3. Fundamental data structures [core]

Minimum core coverage time: 14 hours

Topics:

- Primitive types
- Arrays
- Records
- Strings and string processing
- Data representation in memory
- Static, stack, and heap allocation
- Runtime storage management
- Pointers and references

- Linked structures
- Implementation strategies for stacks, queues, and hash tables
- Implementation strategies for graphs and trees
- Strategies for choosing the right data structure

Learning objectives:

1. Discuss the representation and use of primitive data types and built-in data structures.
2. Describe how the data structures in the topic list are allocated and used in memory.
3. Describe common applications for each data structure in the topic list.
4. Implement the user-defined data structures in a high-level language.
5. Compare alternative implementations of data structures with respect to performance.
6. Write programs that use each of the following data structures: arrays, records, strings, linked lists, stacks, queues, and hash tables.
7. Compare and contrast the costs and benefits of dynamic and static data structure implementations.
8. Choose the appropriate data structure for modeling a given problem.

PF4. Recursion [core]

Minimum core coverage time: 5 hours

Topics:

- The concept of recursion
- Recursive mathematical functions
- Simple recursive procedures
- Divide-and-conquer strategies
- Recursive backtracking
- Implementation of recursion

Learning objectives:

1. Describe the concept of recursion and give examples of its use.
2. Identify the base case and the general case of a recursively defined problem.
3. Compare iterative and recursive solutions for elementary problems such as factorial.
4. Describe the divide-and-conquer approach.
5. Implement, test, and debug simple recursive functions and procedures.
6. Describe how recursion can be implemented using a stack.
7. Discuss problems for which backtracking is an appropriate solution.
8. Determine when a recursive solution is appropriate for a problem.

PF5. Event-driven programming [core]

Minimum core coverage time: 4 hours

Topics:

- Event-handling methods
- Event propagation
- Exception handling

Learning objectives:

1. Explain the difference between event-driven programming and command-line programming.
2. Design, code, test, and debug simple event-driven programs that respond to user events.
3. Develop code that responds to exception conditions raised during execution.

CC2001 Report
December 15, 2001

Algorithms and Complexity (AL)

Algorithms are fundamental to computer science and software engineering. The real-world performance of any software system depends on only two things: (1) the algorithms chosen and (2) the suitability and efficiency of the various layers of implementation. Good algorithm design is therefore crucial for the performance of all software systems. Moreover, the study of algorithms provides insight into the intrinsic nature of the problem as well as possible solution techniques independent of programming language, programming paradigm, computer hardware, or any other implementation aspect.

An important part of computing is the ability to select algorithms appropriate to particular purposes and to apply them, recognizing the possibility that no suitable algorithm may exist. This facility relies on understanding the range of algorithms that address an important set of well-defined problems, recognizing their strengths and weaknesses, and their suitability in particular contexts. Efficiency is a pervasive theme throughout this area.

AL1. Basic algorithmic analysis [core]

Minimum core coverage time: 4 hours

Topics:

- Asymptotic analysis of upper and average complexity bounds
- Identifying differences among best, average, and worst case behaviors
- Big O, little o, omega, and theta notation
- Standard complexity classes
- Empirical measurements of performance
- Time and space tradeoffs in algorithms
- Using recurrence relations to analyze recursive algorithms

Learning objectives:

1. Explain the use of big O, omega, and theta notation to describe the amount of work done by an algorithm.
2. Use big O, omega, and theta notation to give asymptotic upper, lower, and tight bounds on time and space complexity of algorithms.
3. Determine the time and space complexity of simple algorithms.
4. Deduce recurrence relations that describe the time complexity of recursively defined algorithms.
5. Solve elementary recurrence relations.

AL2. Algorithmic strategies [core]

Minimum core coverage time: 6 hours

Topics:

- Brute-force algorithms
- Greedy algorithms
- Divide-and-conquer
- Backtracking
- Branch-and-bound
- Heuristics
- Pattern matching and string/text algorithms
- Numerical approximation algorithms

Learning objectives:

1. Describe the shortcoming of brute-force algorithms.
2. For each of several kinds of algorithm (brute force, greedy, divide-and-conquer, backtracking, branch-and-bound, and heuristic), identify an example of everyday human behavior that exemplifies the basic concept.
3. Implement a greedy algorithm to solve an appropriate problem.
4. Implement a divide-and-conquer algorithm to solve an appropriate problem.
5. Use backtracking to solve a problem such as navigating a maze.
6. Describe various heuristic problem-solving methods.
7. Use pattern matching to analyze substrings.
8. Use numerical approximation to solve mathematical problems, such as finding the roots of a polynomial.

AL3. Fundamental computing algorithms [core]

Minimum core coverage time: 12 hours

Topics:

- Simple numerical algorithms
- Sequential and binary search algorithms
- Quadratic sorting algorithms (selection, insertion)
- O(N log N) sorting algorithms (Quicksort, heapsort, mergesort)
- Hash tables, including collision-avoidance strategies
- Binary search trees
- Representations of graphs (adjacency list, adjacency matrix)
- Depth- and breadth-first traversals
- Shortest-path algorithms (Dijkstra's and Floyd's algorithms)
- Transitive closure (Floyd's algorithm)
- Minimum spanning tree (Prim's and Kruskal's algorithms)
- Topological sort

Learning objectives:

1. Implement the most common quadratic and O(NlogN) sorting algorithms.
2. Design and implement an appropriate hashing function for an application.
3. Design and implement a collision-resolution algorithm for a hash table.
4. Discuss the computational efficiency of the principal algorithms for sorting, searching, and hashing.
5. Discuss factors other than computational efficiency that influence the choice of algorithms, such as programming time, maintainability, and the use of application-specific patterns in the input data.
6. Solve problems using the fundamental graph algorithms, including depth-first and breadth-first search, single-source and all-pairs shortest paths, transitive closure, topological sort, and at least one minimum spanning tree algorithm.
7. Demonstrate the following capabilities: to evaluate algorithms, to select from a range of possible options, to provide justification for that selection, and to implement the algorithm in programming context.

AL4. Distributed algorithms [core]

Minimum core coverage time: 3 hours

Topics:

- Consensus and election
- Termination detection
- Fault tolerance
- Stabilization

Learning objectives:

1. Explain the distributed paradigm.
2. Explain one simple distributed algorithm.

3. Determine when to use consensus or election algorithms.
4. Distinguish between logical and physical clocks.
5. Describe the relative ordering of events in a distributed algorithm.

AL5. Basic computability [core]

Minimum core coverage time: 6 hours

Topics:

- Finite-state machines
- Context-free grammars
- Tractable and intractable problems
- Uncomputable functions
- The halting problem
- Implications of uncomputability

Learning objectives:

1. Discuss the concept of finite state machines.
2. Explain context-free grammars.
3. Design a deterministic finite-state machine to accept a specified language.
4. Explain how some problems have no algorithmic solution.
5. Provide examples that illustrate the concept of uncomputability.

AL6. The complexity classes P and NP [elective]

Topics:

- Definition of the classes P and NP
- NP-completeness (Cook's theorem)
- Standard NP-complete problems
- Reduction techniques

Learning objectives:

1. Define the classes P and NP.
2. Explain the significance of NP-completeness.
3. Prove that a problem is NP-complete by reducing a classic known NP-complete problem to it.

AL7. Automata theory [elective]

Topics:

- Deterministic finite automata (DFAs)
- Nondeterministic finite automata (NFAs)
- Equivalence of DFAs and NFAs
- Regular expressions
- The pumping lemma for regular expressions
- Push-down automata (PDAs)
- Relationship of PDAs and context-free grammars
- Properties of context-free grammars
- Turing machines
- Nondeterministic Turing machines
- Sets and languages
- Chomsky hierarchy
- The Church-Turing thesis

Learning objectives:

1. Determine a language's location in the Chomsky hierarchy (regular sets, context-free, context-sensitive, and recursively enumerable languages).
2. Prove that a language is in a specified class and that it is not in the next lower class.
3. Convert among equivalently powerful notations for a language, including among DFAs, NFAs, and regular expressions, and between PDAs and CFGs.
4. Explain at least one algorithm for both top-down and bottom-up parsing.
5. Explain the Church-Turing thesis and its significance.

AL8. Advanced algorithmic analysis [elective]

Topics:

- Amortized analysis
- Online and offline algorithms
- Randomized algorithms
- Dynamic programming
- Combinatorial optimization

Learning objectives:

1. Use the potential method to provide an amortized analysis of previously unseen data structure, given the potential function.
2. Explain why competitive analysis is an appropriate measure for online algorithms.
3. Explain the use of randomization in the design of an algorithm for a problem where a deterministic algorithm is unknown or much more difficult.
4. Design and implement a dynamic programming solution to a problem.

AL9. Cryptographic algorithms [elective]

Topics:

- Historical overview of cryptography
- Private-key cryptography and the key-exchange problem
- Public-key cryptography
- Digital signatures
- Security protocols
- Applications (zero-knowledge proofs, authentication, and so on)

Learning objectives:

1. Describe efficient basic number-theoretic algorithms, including greatest common divisor, multiplicative inverse mod n, and raising to powers mod n.
2. Describe at least one public-key cryptosystem, including a necessary complexity-theoretic assumption for its security.
3. Create simple extensions of cryptographic protocols, using known protocols and cryptographic primitives.

AL10. Geometric algorithms [elective]

Topics:

- Line segments: properties, intersections
- Convex hull finding algorithms

Learning objectives:

1. Describe and give time analysis of at least two algorithms for finding a convex hull.
2. Justify the Omega(N log N) lower bound on finding the convex hull.
3. Describe at least one additional efficient computational geometry algorithm, such as finding the closest pair of points, convex layers, or maximal layers.

AL11. Parallel algorithms [elective]

Topics:

- PRAM model
- Exclusive versus concurrent reads and writes
- Pointer jumping
- Brent's theorem and work efficiency

Learning objectives:

1. Describe implementation of linked lists on a PRAM.

2. Use parallel-prefix operation to perform simple computations efficiently in parallel.
3. Explain Brent's theorem and its relevance.

CC2001 Report

December 15, 2001

Architecture and Organization (AR)

The computer lies at the heart of computing. Without it most of the computing disciplines today would be a branch of theoretical mathematics. To be a professional in any field of computing today, one should not regard the computer as just a black box that executes programs by magic. All students of computing should acquire some understanding and appreciation of a computer system's functional components, their characteristics, their performance, and their interactions. There are practical implications as well. Students need to understand computer architecture in order to structure a program so that it runs more efficiently on a real machine. In selecting a system to use, they should to able to understand the tradeoff among various components, such as CPU clock speed vs. memory size.

The learning outcomes specified for these topics correspond primarily to the core and are intended to support programs that elect to require only the minimum 36 hours of computer architecture of their students. For programs that want to teach more than the minimum, the same topics (AR1-AR7) can be treated at a more advanced level by implementing a two-course sequence. For programs that want to cover the elective topics, those topics can be introduced within a two-course sequence and/or be treated in a more comprehesive way in a third course.

AR1. Digital logic and digital systems [core]

Minimum core coverage time: 6 hours

Topics:

- Overview and history of computer architecture
- Fundamental building blocks (logic gates, flip-flops, counters, registers, PLA)
- Logic expressions, minimization, sum of product forms
- Register transfer notation
- Physical considerations (gate delays, fan-in, fan-out)

Learning objectives:

1. Describe the progression of computer architecture from vacuum tubes to VLSI.
2. Demonstrate an understanding of the basic building blocks and their role in the historical

development of computer architecture.
3. Use mathematical expressions to describe the functions of simple combinational and sequential circuits.
4. Design a simple circuit using the fundamental building blocks.

AR2. Machine level representation of data [core]

Minimum core coverage time: 3 hours

Topics:

- Bits, bytes, and words
- Numeric data representation and number bases
- Fixed- and floating-point systems
- Signed and twos-complement representations
- Representation of nonnumeric data (character codes, graphical data)
- Representation of records and arrays

Learning objectives:

1. Explain the reasons for using different formats to represent numerical data.
2. Explain how negative integers are stored in sign-magnitude and twos-complement representation.
3. Convert numerical data from one format to another.
4. Discuss how fixed-length number representations affect accuracy and precision.
5. Describe the internal representation of nonnumeric data.
6. Describe the internal representation of characters, strings, records, and arrays.

AR3. Assembly level machine organization [core]

Minimum core coverage time: 9 hours

Topics:

- Basic organization of the von Neumann machine
- Control unit; instruction fetch, decode, and execution
- Instruction sets and types (data manipulation, control, I/O)
- Assembly/machine language programming
- Instruction formats
- Addressing modes
- Subroutine call and return mechanisms
- I/O and interrupts

Learning objectives:

1. Explain the organization of the classical von Neumann machine and its major functional units.
2. Explain how an instruction is executed in a classical von Neumann machine.
3. Summarize how instructions are represented at both the machine level and in the context of a symbolic assembler.
4. Explain different instruction formats, such as addresses per instruction and variable length vs. fixed length formats.
5. Write simple assembly language program segments.
6. Demonstrate how fundamental high-level programming constructs are implemented at the machine-language level.
7. Explain how subroutine calls are handled at the assembly level.
8. Explain the basic concepts of interrupts and I/O operations.

AR4. Memory system organization and architecture [core]

Minimum core coverage time: 5 hours

Topics:

- Storage systems and their technology
- Coding, data compression, and data integrity
- Memory hierarchy
- Main memory organization and operations
- Latency, cycle time, bandwidth, and interleaving
- Cache memories (address mapping, block size, replacement and store policy)
- Virtual memory (page table, TLB)
- Fault handling and reliability

Learning objectives:

1. Identify the main types of memory technology.
2. Explain the effect of memory latency on running time.
3. Explain the use of memory hierarchy to reduce the effective memory latency.
4. Describe the principles of memory management.
5. Describe the role of cache and virtual memory.
6. Explain the workings of a system with virtual memory management.

AR5. Interfacing and communication [core]

Minimum core coverage time: 3 hours

Topics:

- I/O fundamentals: handshaking, buffering, programmed I/O, interrupt-driven I/O
- Interrupt structures: vectored and prioritized, interrupt acknowledgment
- External storage, physical organization, and drives

- Buses: bus protocols, arbitration, direct-memory access (DMA)
- Introduction to networks
- Multimedia support
- RAID architectures

Learning objectives:

1. Explain how interrupts are used to implement I/O control and data transfers.
2. Identify various types of buses in a computer system.
3. Describe data access from a magnetic disk drive.
4. Compare the common network configurations.
5. Identify interfaces needed for multimedia support.
6. Describe the advantages and limitations of RAID architectures.

AR6. Functional organization [core]

Minimum core coverage time: 7 hours

Topics:

- Implementation of simple datapaths
- Control unit: hardwired realization vs. microprogrammed realization
- Instruction pipelining
- Introduction to instruction-level parallelism (ILP)

Learning objectives:

1. Compare alternative implementation of datapaths.
2. Discuss the concept of control points and the generation of control signals using hardwired or microprogrammed implementations.
3. Explain basic instruction level parallelism using pipelining and the major hazards that may occur.

AR7. Multiprocessing and alternative architectures [core]

Minimum core coverage time: 3 hours

Topics:

- Introduction to SIMD, MIMD, VLIW, EPIC
- Systolic architecture
- Interconnection networks (hypercube, shuffle-exchange, mesh, crossbar)
- Shared memory systems
- Cache coherence

- Memory models and memory consistency

Learning objectives:

1. Discuss the concept of parallel processing beyond the classical von Neumann model.
2. Describe alternative architectures such as SIMD, MIMD, and VLIW.
3. Explain the concept of interconnection networks and characterize different approaches.
4. Discuss the special concerns that multiprocessing systems present with respect to memory management and describe how these are addressed.

AR8. Performance enhancements [elective]

Topics:

- Superscalar architecture
- Branch prediction
- Prefetching
- Speculative execution
- Multithreading
- Scalability

Learning objectives:

1. Describe superscalar architectures and their advantages.
2. Explain the concept of branch prediction and its utility.
3. Characterize the costs and benefits of prefetching.
4. Explain speculative execution and identify the conditions that justify it.
5. Discuss the performance advantages that multithreading can offer in an architecture along with the factors that make it difficult to derive maximum benefits from this approach.
6. Describe the relevance of scalability to performance.

AR9. Architecture for networks and distributed systems [elective]

Topics:

- Introduction to LANs and WANs
- Layered protocol design, ISO/OSI, IEEE 802
- Impact of architectural issues on distributed algorithms
- Network computing
- Distributed multimedia

Learning objectives:

1. Explain the basic components of network systems and distinguish between LANs and WANs.
2. Discuss the architectural issues involved in the design of a layered network protocol.

3. Explain how architectures differ in network and distributed systems.
4. Discuss architectural issues related to network computing and distributed multimedia.

Human-Computer Interaction (HC)

This list of topics is intended as an introduction to human-computer interaction for computer science majors. Emphasis will be placed on understanding human behavior with interactive objects, knowing how to develop and evaluate interactive software using a human-centered approach, and general knowledge of HCI design issues with multiple types of interactive software. Units HC1 (Foundations of Human-Computer Interaction) and HC2 (Building a simple graphical user interface) will be required for all majors, possibly as modules in the introductory courses. The remaining units will most likely be integrated into one or two elective courses at the junior or senior level.

HC1. Foundations of human-computer interaction [core]

Minimum core coverage time: 6 hours

Topics:

- Motivation: Why care about people?
- Contexts for HCI (tools, web hypermedia, communication)
- Human-centered development and evaluation
- Human performance models: perception, movement, and cognition
- Human performance models: culture, communication, and organizations
- Accommodating human diversity
- Principles of good design and good designers; engineering tradeoffs
- Introduction to usability testing

Learning objectives:

1. Discuss the reasons for human-centered software development.
2. Summarize the basic science of psychological and social interaction.
3. Differentiate between the role of hypotheses and experimental results vs. correlations.
4. Develop a conceptual vocabulary for analyzing human interaction with software: affordance, conceptual model, feedback, and so forth.
5. Distinguish between the different interpretations that a given icon, symbol, word, or color can have in (a) two different human cultures and (b) in a culture and one of its subcultures.
6. In what ways might the design of a computer system or application succeed or fail in terms of

respecting human diversity.
7. Create and conduct a simple usability test for an existing software application.

HC2. Building a simple graphical user interface [core]

Minimum core coverage time: 2 hours

Topics:

- Principles of graphical user interfaces (GUIs)
- GUI toolkits

Learning objectives:

1. Identify several fundamental principles for effective GUI design.
2. Use a GUI toolkit to create a simple application that supports a graphical user interface.
3. Illustrate the effect of fundamental design principles on the structure of a graphical user interface.
4. Conduct a simple usability test for each instance and compare the results.

HC3. Human-centered software evaluation [elective]

Topics:

- Setting goals for evaluation
- Evaluation without users: walkthroughs, KLM, guidelines, and standards
- Evaluation with users: usability testing, interview, survey, experiment

Learning objectives:

1. Discuss evaluation criteria: learning, task time and completion, acceptability.
2. Conduct a walkthrough and a Keystroke Level Model (KLM) analysis.
3. Summarize the major guidelines and standards.
4. Conduct a usability test, an interview, and a survey.
5. Compare a usability test to a controlled experiment.
6. Evaluate an existing interactive system with human-centered criteria and a usability test.

HC4. Human-centered software development [elective]

Topics:

- Approaches, characteristics, and overview of process
- Functionality and usability: task analysis, interviews, surveys
- Specifying interaction and presentation

- Prototyping techniques and tools
    - Paper storyboards
    - Inheritance and dynamic dispatch
    - Prototyping languages and GUI builders

Learning objectives:

1. Explain the basic types and features of human-centered development.
2. Compare human-centered development to traditional software engineering methods.
3. State three functional requirements and three usability requirements.
4. Specify an interactive object with transition networks, OO design, or scenario descriptions.
5. Discuss the pros and cons of development with paper and software prototypes.

HC5. Graphical user-interface design [elective]

Topics:

- Choosing interaction styles and interaction techniques
- HCI aspects of common widgets
- HCI aspects of screen design: layout, color, fonts, labeling
- Handling human failure
- Beyond simple screen design: visualization, representation, metaphor
- Multi-modal interaction: graphics, sound, and haptics
- 3D interaction and virtual reality

Learning objectives:

1. Summarize common interaction styles.
2. Explain good design principles of each of the following: common widgets; sequenced screen presentations; simple error-trap dialog; a user manual.
3. Design, prototype, and evaluate a simple 2D GUI illustrating knowledge of the concepts taught in HC3 and HC4.
4. Discuss the challenges that exist in moving from 2D to 3D interaction.

HC6. Graphical user-interface programming [elective]

Topics:

- UIMS, dialogue independence and levels of analysis, Seeheim model
- Widget classes
- Event management and user interaction
- Geometry management
- GUI builders and UI programming environments
- Cross-platform design

Learning objectives:

1. Differentiate between the responsibilities of the UIMS and the application.
2. Differentiate between kernel-based and client-server models for the UI.
3. Compare the event-driven paradigm with more traditional procedural control for the UI.
4. Describe aggregation of widgets and constraint-based geometry management.
5. Explain callbacks and their role in GUI builders.
6. Identify at least three differences common in cross-platform UI design.
7. Identify as many commonalities as you can that are found in UIs across different platforms.

HC7. HCI aspects of multimedia systems [elective]

Topics:

- Categorization and architectures of information: hierarchies, hypermedia
- Information retrieval and human performance
    - Web search
    - Usability of database query languages
    - Graphics
    - Sound
- HCI design of multimedia information systems
- Speech recognition and natural language processing
- Information appliances and mobile computing

Learning objectives:

1. Discuss how information retrieval differs from transaction processing.
2. Explain how the organization of information supports retrieval.
3. Describe the major usability problems with database query languages.
4. Explain the current state of speech recognition technology in particular and natural language processing in general.
5. Design, prototype, and evaluate a simple Multimedia Information System illustrating knowledge of the concepts taught in HC4, HC5, and HC7.

HC8. HCI aspects of collaboration and communication [elective]

Topics:

- Groupware to support specialized tasks: document preparation, multi-player games
- Asynchronous group communication: e-mail, bulletin boards
- Synchronous group communication: chat rooms, conferencing
- Online communities: MUDs/MOOs
- Software characters and intelligent agents

Learning objectives:

1. Compare the HCI issues in individual interaction with group interaction.
2. Discuss several issues of social concern raised by collaborative software.
3. Discuss the HCI issues in software that embodies human intention.
4. Describe the difference between synchronous and asynchronous communication.
5. Design, prototype, and evaluate a simple groupware or group communication application illustrating knowledge of the concepts taught in HC4, HC5, and HC8.
6. Participate in a team project for which some interaction is face-to-face and other interaction occurs via a mediating software environment.
7. Describe the similarities and differences between face-to-face and software-mediated collaboration.

Graphics and Visual Computing (GV)

The area encompassed by Graphics and Visual Computing (GV) is divided into four interrelated fields:

- *Computer graphics.* Computer graphics is the art and science of communicating information using images that are generated and presented through computation. This requires (a) the design and construction of models that represent information in ways that support the creation and viewing of images, (b) the design of devices and techniques through which the person may interact with the model or the view, (c) the creation of techniques for rendering the model, and (d) the design of ways the images may be preserved The goal of computer graphics is to engage the person's visual centers alongside other cognitive centers in understanding.
- *Visualization.* The field of visualization seeks to determine and present underlying correlated structures and relationships in both scientific (computational and medical sciences) and more abstract datasets. The prime objective of the presentation should be to communicate the information in a dataset so as to enhance understanding. Although current techniques of visualization exploit visual abilities of humans, other sensory modalities, including sound and haptics (touch), are also being considered to aid the discovery process of information.
- *Virtual reality.* Virtual reality (VR) enables users to experience a three-dimensional environment generated using computer graphics, and perhaps other sensory modalities, to provide an environment for enhanced interaction between a human user and a computer-created world.
- *Computer vision.* The goal of computer vision (CV) is to deduce the properties and structure of the three-dimensional world from one or more two-dimensional images. The understanding and practice of computer vision depends upon core concepts in computing, but also relates strongly to the disciplines of physics, mathematics, and psychology.

GV1. Fundamental techniques in graphics [core]

Minimum core coverage time: 2 hours

Topics:

- Hierarchy of graphics software
- Using a graphics API
- Simple color models (RGB, HSB, CMYK)
- Homogeneous coordinates
- Affine transformations (scaling, rotation, translation)
- Viewing transformation
- Clipping

Learning objectives:

1. Distinguish the capabilities of different levels of graphics software and describe the appropriateness of each.
2. Create images using a standard graphics API.
3. Use the facilities provided by a standard API to express basic transformations such as scaling, rotation, and translation.
4. Implement simple procedures that perform transformation and clipping operations on a simple 2-dimensional image.
5. Discuss the 3-dimensional coordinate system and the changes required to extend 2D transformation operations to handle transformations in 3D

GV2. Graphic systems [core]

Minimum core coverage time: 1 hour

Topics:

- Raster and vector graphics systems
- Video display devices
- Physical and logical input devices
- Issues facing the developer of graphical systems

Learning objectives:

1. Describe the appropriateness of graphics architectures for given applications.
2. Explain the function of various input devices.
3. Compare and contrast the techniques of raster graphics and vector graphics.
4. Use current hardware and software for creating and displaying graphics.
5. Discuss the expanded capabilities of emerging hardware and software for creating and displaying graphics.

GV3. Graphic communication [elective]

Topics:

- Psychodynamics of color and interactions among colors
- Modifications of color for vision deficiency
- Cultural meaning of different colors
- Use of effective pseudo-color palettes for images for specific audiences
- Structuring a view for effective understanding
- Image modifications for effective video and hardcopy
- Use of legends to key information to color or other visual data
- Use of text in images to present context and background information
- Visual user feedback on graphical operations

Learning objectives:

1. Explain the value of using colors and pseudo-colors.
2. Demonstrate the ability to create effective video and hardcopy images.
3. Identify effective and ineffective examples of communication using graphics.
4. Create effective examples of graphic communication, making appropriate use of color, legends, text, and/or video.
5. Create two effective examples that communicate the same content: one designed for hardcopy presentation and the other designed for online presentation.
6. Discuss the differences in design criteria for hardcopy and online presentations.

GV4. Geometric modeling [elective]

Topics:

- Polygonal representation of 3D objects
- Parametric polynomial curves and surfaces
- Constructive Solid Geometry (CSG) representation
- Implicit representation of curves and surfaces
- Spatial subdivision techniques
- Procedural models
- Deformable models
- Subdivision surfaces
- Multiresolution modeling
- Reconstruction

Learning objectives:

1. Create simple polyhedral models by surface tessellation.
2. Construct CSG models from simple primitives, such as cubes and quadric surfaces.
3. Generate a mesh representation from an implicit surface.
4. Generate a fractal model or terrain using a procedural method.
5. Generate a mesh from data points acquired with a laser scanner.

GV5. Basic rendering [elective]

Topics:

- Line generation algorithms (Bresenham)
- Font generation: outline vs. bitmap
- Light-source and material properties
- Ambient, diffuse, and specular reflections
- Phong reflection model
- Rendering of a polygonal surface; flat, Gouraud, and Phong shading
- Texture mapping, bump texture, environment map
- Introduction to ray tracing
- Image synthesis, sampling techniques, and anti-aliasing

Learning objectives:

1. Explain the operation of the Bresenham algorithm for rendering a line on a pixel-based display.
2. Explain the concept and applications of each of these techniques.
3. Demonstrate each of these techniques by creating an image using a standard API.
4. Describe how a graphic image has been created.

GV6. Advanced rendering [elective]

Topics:

- Transport equations
- Ray tracing algorithms
- Photon tracing
- Radiosity for global illumination computation, form factors
- Efficient approaches to global illumination
- Monte Carlo methods for global illumination
- Image-based rendering, panorama viewing, plenoptic function modeling
- Rendering of complex natural phenomenon
- Non-photorealistic rendering

Learning objectives:

1. Describe several transport equations in detail, noting all comprehensive effects.
2. Describe efficient algorithms to compute radiosity and explain the tradeoffs of accuracy and algorithmic performance.
3. Describe the impact of meshing schemes.
4. Explain image-based rendering techniques, light fields, and associated topics.

GV7. Advanced techniques [elective]

Topics:

- Color quantization
- Scan conversion of 2D primitive, forward differencing
- Tessellation of curved surfaces
- Hidden surface removal methods
- Z-buffer and frame buffer, color channels (a channel for opacity)
- Advanced geometric modeling techniques

Learning objectives:

1. Describe the techniques identified in this section.
2. Explain how to recognize the graphics techniques used to create a particular image.
3. Implement any of the specified graphics techniques using a primitive graphics system at the individual pixel level.
4. Use common animation software to construct simple organic forms using metaball and skeleton.

GV8. Computer animation [elective]

Topics:

- Key-frame animation
- Camera animation
- Scripting system
- Animation of articulated structures: inverse kinematics
- Motion capture
- Procedural animation
- Deformation

Learning objectives:

1. Explain the spline interpolation method for producing in-between positions and orientations.
2. Compare and contrast several technologies for motion capture.
3. Use the particle function in common animation software to generate a simple animation, such as fireworks.
4. Use free-form deformation techniques to create various deformations.

GV9. Visualization [elective]

Topics:

- Basic viewing and interrogation functions for visualization
- Visualization of vector fields, tensors, and flow data
- Visualization of scalar field or height field: isosurface by the marching cube method
- Direct volume data rendering: ray-casting, transfer functions, segmentation, hardware

- Information visualization: projection and parallel-coordinates methods

Learning objectives:

1. Describe the basic algorithms behind scalar and vector visualization.
2. Describe the tradeoffs of the algorithms in terms of accuracy and performance.
3. Employ suitable theory from signal processing and numerical analysis to explain the effects of visualization operations.
4. Describe the impact of presentation and user interaction on exploration.

GV10. Virtual reality [elective]

Topics:

- Stereoscopic display
- Force feedback simulation, haptic devices
- Viewer tracking
- Collision detection
- Visibility computation
- Time-critical rendering, multiple levels of details (LOD)
- Image-base VR system
- Distributed VR, collaboration over computer network
- Interactive modeling
- User interface issues
- Applications in medicine, simulation, and training

Learning objectives:

1. Describe the optical model realized by a computer graphics system to synthesize stereoscopic view.
2. Describe the principles of different viewer tracking technologies.
3. Explain the principles of efficient collision detection algorithms for convex polyhedra.
4. Describe the differences between geometry- and image-based virtual reality.
5. Describe the issues of user action synchronization and data consistency in a networked environment.
6. Determine the basic requirements on interface, hardware, and software configurations of a VR system for a specified application.

GV11. Computer vision [elective]

Topics:

- Image acquisition
- The digital image and its properties
- Image preprocessing

- Segmentation (thresholding, edge- and region-based segmentation)
- Shape representation and object recognition
- Motion analysis
- Case studies (object recognition, object tracking)

Learning objectives:

1. Explain the image formation process.
2. Explain the advantages of two and more cameras, stereo vision.
3. Explain various segmentation approaches, along with their characteristics, differences, strengths, and weaknesses.
4. Describe object recognition based on contour- and region-based shape representations.
5. Explain differential motion analysis methods.
6. Describe the differences in object tracking methods.

Intelligent Systems (IS)

The field of artificial intelligence (AI) is concerned with the design and analysis of autonomous agents. These are software systems and/or physical machines, with sensors and actuators, embodied for example within a robot or an autonomous spacecraft. An intelligent system has to perceive its environment, to act rationally towards its assigned tasks, to interact with other agents and with human beings.

These capabilities are covered by topics such as computer vision, planning and acting, robotics, multiagents systems, speech recognition, and natural language understanding. They rely on a broad set of general and specialized knowledge representations and reasoning mechanisms, on problem solving and search algorithms, and on machine learning techniques.

Furthermore, artificial intelligence provides a set of tools for solving problems that are difficult or impractical to solve with other methods. These include heuristic search and planning algorithms, formalisms for knowledge representation and reasoning, machine learning techniques, and methods applicable to sensing and action problems such as speech and language understanding, computer vision, and robotics, among others. The student needs to be able to determine when an AI approach is appropriate for a given problem, and to be able to select and implement a suitable AI method.

IS1. Fundamental issues in intelligent systems [core]

Minimum core coverage time: 1 hour

Topics:

- History of artificial intelligence
- Philosophical questions
  - The Turing test
  - Searle's "Chinese Room" thought experiment
  - Ethical issues in AI
- Fundamental definitions
  - Optimal vs. human-like reasoning

    - Optimal vs. human-like behavior
- Philosophical questions
- Modeling the world
- The role of heuristics

Learning objectives:

1. Describe the Turing test and the "Chinese Room" thought experiment.
2. Differentiate the concepts of optimal reasoning and human-like reasoning.
3. Differentiate the concepts of optimal behavior and human-like behavior.
4. List examples of intelligent systems that depend on models of the world.
5. Describe the role of heuristics and the need for tradeoffs between optimality and efficiency.

IS2. Search and constraint satisfaction [core]

Minimum core coverage time: 5 hours

Topics:

- Problem spaces
- Brute-force search (breadth-first, depth-first, depth-first with iterative deepening)
- Best-first search (generic best-first, Dijkstra's algorithm, A*, admissibility of A*)
- Two-player games (minimax search, alpha-beta pruning)
- Constraint satisfaction (backtracking and local search methods)

Learning objectives:

1. Formulate an efficient problem space for a problem expressed in English by expressing that problem space in terms of states, operators, an initial state, and a description of a goal state.
2. Describe the problem of combinatorial explosion and its consequences.
3. Select an appropriate brute-force search algorithm for a problem, implement it, and characterize its time and space complexities.
4. Select an appropriate heuristic search algorithm for a problem and implement it by designing the necessary heuristic evaluation function.
5. Describe under what conditions heuristic algorithms guarantee optimal solution.
6. Implement minimax search with alpha-beta pruning for some two-player game.
7. Formulate a problem specified in English as a constraint-satisfaction problem and implement it using a chronological backtracking algorithm.

IS3. Knowledge representation and reasoning [core]

Minimum core coverage time: 4 hours

Topics:

- Review of propositional and predicate logic
- Resolution and theorem proving
- Nonmonotonic inference
- Probabilistic reasoning
- Bayes theorem

Learning objectives:

1. Explain the operation of the resolution technique for theorem proving.
2. Explain the distinction between monotonic and nonmonotonic inference.
3. Discuss the advantages and shortcomings of probabilistic reasoning.
4. Apply Bayes theorem to determine conditional probabilities.

IS4. Advanced search [elective]

Topics:

- Genetic algorithms
- Simulated annealing
- Local search

Learning objectives:

1. Explain what genetic algorithms are and constrast their effectiveness with the classic problem-solving and search techniques.
2. Explain how simulated annealing can be used to reduce search complexity and contrast its operation with classic search techniques.
3. Apply local search techniques to a classic domain.

IS5. Advanced knowledge representation and reasoning [elective]

Topics:

- Structured representation
    - Frames and objects
    - Description logics
    - Inheritance systems
- Nonmonotonic reasoning
    - Nonclassical logics
    - Default reasoning
    - Belief revision
    - Preference logics
    - Integration of knowledge sources
    - Aggregation of conflicting belief
- Reasoning on action and change

    - Situation calculus
    - Event calculus
    - Ramification problems
- Temporal and spatial reasoning
- Uncertainty
    - Probabilistic reasoning
    - Bayesian nets
    - Fuzzy sets and possibility theory
    - Decision theory
- Knowledge representation for diagnosis, qualitative representation

Learning objectives:

1. Compare and contrast the most common models used for structured knowledge representation, highlighting their strengths and weaknesses.
2. Characterize the components of nonmonotonic reasoning and its usefulness as a representational mechanisms for belief systems.
3. Apply situation and event calculus to problems of action and change.
4. Articulate the distinction between temporal and spatial reasoning, explaining how they interrelate.
5. Describe and contrast the basic techniques for representing uncertainty.
6. Describe and contrast the basic techniques for diagnosis and qualitative representation.

IS6. Agents [elective]

Topics:

- Definition of agents
- Successful applications and state-of-the-art agent-based systems
- Agent architectures
    - Simple reactive agents
    - Reactive planners
    - Layered architectures
    - Example architectures and applications
- Agent theory
    - Commitments
    - Intentions
    - Decision-theoretic agents
    - Markov decision processes (MDP)
- Software agents, personal assistants, and information access
    - Collaborative agents
    - Information-gathering agents
- Believable agents (synthetic characters, modeling emotions in agents)
- Learning agents
- Multi-agent systems

    - Economically inspired multi-agent systems
    - Collaborating agents
    - Agent teams
    - Agent modeling
    - Multi-agent learning
- Introduction to robotic agents
- Mobile agents

Learning objectives:

1. Explain how an agent differs from other categories of intelligent systems.
2. Characterize and contrast the standard agent architectures.
3. Describe the applications of agent theory, to domains such as software agents, personal assistants, and believable agents.
4. Describe the distinction between agents that learn and those that don't.
5. Demonstrate using appropriate examples how multi-agent systems support agent interaction.
6. Describe and contrast robotic and mobile agents.

IS7. Natural language processing [elective]

Topics:

- Deterministic and stochastic grammars
- Parsing algorithms
- Corpus-based methods
- Information retrieval
- Language translation
- Speech recognition

Learning objectives:

1. Define and contrast deterministic and stochastic grammars, providing examples to show the adequacy of each.
2. Identify the classic parsing algorithms for parsing natural language.
3. Defend the need for an established corpus.
4. Give examples of catalog and look up procedures in a corpus-based approach.
5. Articulate the distinction between techniques for information retrieval, language translation, and speech recognition.

IS8. Machine learning and neural networks [elective]

Topics:

- Definition and examples of machine learning
- Supervised learning

- Learning decision trees
- Learning neural networks
- Learning belief networks
- The nearest neighbor algorithm
- Learning theory
- The problem of overfitting
- Unsupervised learning
- Reinforcement learning

Learning objectives:

1. Explain the differences among the three main styles of learning: supervised, reinforcement, and unsupervised.
2. Implement simple algorithms for supervised learning, reinforcement learning, and unsupervised learning.
3. Determine which of the three learning styles is appropriate to a particular problem domain.
4. Compare and contrast each of the following techniques, providing examples of when each strategy is superior: decision trees, neural networks, and belief networks..
5. Implement a simple learning system using decision trees, neural networks and/or belief networks, as appropriate.
6. Characterize the state of the art in learning theory, including its achievements and its shortcomings.
7. Explain the nearest neighbor algorithm and its place within learning theory.
8. Explain the problem of overfitting, along with techniques for detecting and managing the problem.

IS9. AI planning systems [elective]

Topics:

- Definition and examples of planning systems
- Planning as search
- Operator-based planning
- Propositional planning
- Extending planning systems (case-based, learning, and probabilistic systems)
- Static world planning systems
- Planning and execution
- Planning and robotics

Learning objectives:

1. Define the concept of a planning system.
2. Explain how planning systems differ from classical search techniques.
3. Articulate the differences between planning as search, operator-based planning, and propositional planning, providing examples of domains where each is most applicable.

4. Define and provide examples for each of the following techniques: case-based, learning, and probablistic planning.
5. Compare and contrast static world planning systems with those need dynamic execution.
6. Explain the impact of dynamic planning on robotics.

IS10. Robotics [elective]

Topics:

- Overview
  - State-of-the-art robot systems
  - Planning vs. reactive control
  - Uncertainty in control
  - Sensing
  - World models
- Configuration space
- Planning
- Sensing
- Robot programming
- Navigation and control

Learning objectives:

1. Outline the potential and limitations of today's state-of-the-art robot systems.
2. Implement configuration space algorithms for a 2D robot and complex polygons.
3. Implement simple motion planning algorithms.
4. Explain the uncertainties associated with sensors and how to deal with those uncertainties.
5. Design a simple control architecture.
6. Describe various strategies for navigation in unknown environments, including the strengths and shortcomings of each.
7. Describe various strategies for navigation with the aid of landmarks, including the strengths and shortcomings of each.

Operating Systems (OS)

An operating system defines an abstraction of hardware behavior with which programmers can control the hardware. It also manages resource sharing among the computer's users. The topics in this area explain the issues that influence the design of contemporary operating systems. Courses that cover this area will typically include a laboratory component to enable students to experiment with operating systems.

Over the years, operating systems and their abstractions have become complex relative to typical application software. It is necessary to ensure that the student understands the extent of the use of an operating system prior to a detailed study of internal implementation algorithms and data structures. Therefore these topics address both the use of operating systems (externals) and their design and implementation (internals). Many of the ideas involved in operating system use have wider applicability across the field of computer science, such as concurrent programming. Studying internal design has relevance in such diverse areas as dependable programming, algorithm design and implementation, modern device development, building virtual environments, caching material across the web, building secure and safe systems, network management, and many others.

OS1. Overview of operating systems [core]

Minimum core coverage time: 2 hours

Topics:

- Role and purpose of the operating system
- History of operating system development
- Functionality of a typical operating system
- Mechanisms to support client-server models, hand-held devices
- Design issues (efficiency, robustness, flexibility, portability, security, compatibility)
- Influences of security, networking, multimedia, windows

Learning objectives:

1. Explain the objectives and functions of modern operating systems.
2. Describe how operating systems have evolved over time from primitive batch systems to sophisticated multiuser systems.
3. Analyze the tradeoffs inherent in operating system design.
4. Describe the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve.
5. Discuss networked, client-server, distributed operating systems and how they differ from single user operating systems.
6. Identify potential threats to operating systems and the security features design to guard against them.
7. Describe how issues such as open source software and the increased use of the Internet are influencing operating system design.

OS2. Operating system principles [core]

Minimum core coverage time: 2 hours

Topics:

- Structuring methods (monolithic, layered, modular, micro-kernel models)
- Abstractions, processes, and resources
- Concepts of application program interfaces (APIs)
- Application needs and the evolution of hardware/software techniques
- Device organization
- Interrupts: methods and implementations
- Concept of user/system state and protection, transition to kernel mode

Learning objectives:

1. Explain the concept of a logical layer.
2. Explain the benefits of building abstract layers in hierarchical fashion.
3. Defend the need for APIs and middleware.
4. Describe how computing resources are used by application software and managed by system software.
5. Contrast kernel and user mode in an operating system.
6. Discuss the advantages and disadvantages of using interrupt processing.
7. Compare and contrast the various ways of structuring an operating system such as object-oriented, modular, micro-kernel, and layered.
8. Explain the use of a device list and driver I/O queue.

OS3. Concurrency [core]

Minimum core coverage time: 6 hours

Topics:

- States and state diagrams
- Structures (ready list, process control blocks, and so forth)
- Dispatching and context switching
- The role of interrupts
- Concurrent execution: advantages and disadvantages
- The "mutual exclusion" problem and some solutions
- Deadlock: causes, conditions, prevention
- Models and mechanisms (semaphores, monitors, condition variables, rendezvous)
- Producer-consumer problems and synchronization
- Multiprocessor issues (spin-locks, reentrancy)

Learning objectives:

1. Describe the need for concurrency within the framework of an operating system.
2. Demonstrate the potential run-time problems arising from the concurrent operation of many separate tasks.
3. Summarize the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each.
4. Explain the different states that a task may pass through and the data structures needed to support the management of many tasks.
5. Summarize the various approaches to solving the problem of mutual exclusion in an operating system.
6. Describe reasons for using interrupts, dispatching, and context switching to support concurrency in an operating system.
7. Create state and transition diagrams for simple problem domains.
8. Discuss the utility of data structures, such as stacks and queues, in managing concurrency.
9. Explain conditions that lead to deadlock.

OS4. Scheduling and dispatch [core]

Minimum core coverage time: 3 hours

Topics:

- Preemptive and nonpreemptive scheduling
- Schedulers and policies
- Processes and threads
- Deadlines and real-time issues

Learning objectives:

1. Compare and contrast the common algorithms used for both preemptive and non-preemptive

scheduling of tasks in operating systems, such as priority, performance comparison, and fair-share schemes.
2. Describe relationships between scheduling algorithms and application domains.
3. Discuss the types of processor scheduling such as short-term, medium-term, long-term, and I/O.
4. Describe the difference between processes and threads.
5. Compare and contrast static and dynamic approaches to real-time scheduling.
6. Discuss the need for preemption and deadline scheduling.
7. Identify ways that the logic embodied in scheduling algorithms are applicable to other domains, such as disk I/O, network scheduling, project scheduling, and other problems unrelated to computing.

OS5. Memory management [core]

Minimum core coverage time: 5 hours

Topics:

- Review of physical memory and memory management hardware
- Overlays, swapping, and partitions
- Paging and segmentation
- Placement and replacement policies
- Working sets and thrashing
- Caching

Learning objectives:

1. Explain memory hierarchy and cost-performance tradeoffs.
2. Explain the concept of virtual memory and how it is realized in hardware and software.
3. Summarize the principles of virtual memory as applied to caching, paging, and segmentation.
4. Evaluate the tradeoffs in terms of memory size (main memory, cache memory, auxiliary memory) and processor speed.
5. Defend the different ways of allocating memory to tasks, citing the relative merits of each.
6. Describe the reason for and use of cache memory.
7. Compare and contrast paging and segmentation techniques.
8. Discuss the concept of thrashing, both in terms of the reasons it occurs and the techniques used to recognize and manage the problem.
9. Analyze the various memory portioning techniques including overlays, swapping, and placement and replacement policies.

OS6. Device management [elective]

Topics:

- Characteristics of serial and parallel devices

- Abstracting device differences
- Buffering strategies
- Direct memory access
- Recovery from failures

Learning objectives:

1. Explain the key difference between serial and parallel devices and identify the conditions in which each is appropriate.
2. Identify the relationship between the physical hardware and the virtual devices maintained by the operating system.
3. Explain buffering and describe strategies for implementing it.
4. Differentiate the mechanisms used in interfacing a range of devices (including hand-held devices, networks, multimedia) to a computer and explain the implications of these for the design of an operating system.
5. Describe the advantages and disadvantages of direct memory access and discuss the circumstances in which its use is warranted.
6. Identify the requirements for failure recovery.
7. Implement a simple device driver for a range of possible devices.

OS7. Security and protection [elective]

Topics:

- Overview of system security
- Policy/mechanism separation
- Security methods and devices
- Protection, access, and authentication
- Models of protection
- Memory protection
- Encryption
- Recovery management

Learning objectives:

1. Defend the need for protection and security, and the role of ethical considerations in computer use.
2. Summarize the features and limitations of an operating system used to provide protection and security.
3. Compare and contrast current methods for implementing security.
4. Compare and contrast the strengths and weaknesses of two or more currently popular operating systems with respect to security.
5. Compare and contrast the security strengths and weaknesses of two or more currently popular operating systems with respect to recovery management.

OS8. File systems [elective]

Topics:

- Files: data, metadata, operations, organization, buffering, sequential, nonsequential
- Directories: contents and structure
- File systems: partitioning, mount/unmount, virtual file systems
- Standard implementation techniques
- Memory-mapped files
- Special-purpose file systems
- Naming, searching, access, backups

Learning objectives:

1. Summarize the full range of considerations that support file systems.
2. Compare and contrast different approaches to file organization, recognizing the strengths and weaknesses of each.
3. Summarize how hardware developments have lead to changes in our priorities for the design and the management of file systems.

OS9. Real-time and embedded systems [elective]

Topics:

- Process and task scheduling
- Memory/disk management requirements in a real-time environment
- Failures, risks, and recovery
- Special concerns in real-time systems

Learning objectives:

1. Describe what makes a system a real-time system.
2. Explain the presence of and describe the characteristics of latency in real-time systems.
3. Summarize special concerns that real-time systems present and how these concerns are addressed.

OS10. Fault tolerance [elective]

Topics:

- Fundamental concepts: reliable and available systems
- Spatial and temporal redundancy
- Methods used to implement fault tolerance
- Examples of reliable systems

Learning objectives:

1. Explain the relevance of the terms *fault tolerance*, *reliability*, and *availability*.
2. Outline the range of methods for implementing fault tolerance in an operating system.
3. Explain how an operating system can continue functioning after a fault occurs.

OS11. System performance evaluation [elective]

Topics:

- Why system performance needs to be evaluated
- What is to be evaluated
- Policies for caching, paging, scheduling, memory management, security, and so forth
- Evaluation models: deterministic, analytic, simulation, or implementation-specific
- How to collect evaluation data (profiling and tracing mechanisms)

Learning objectives:

1. Describe the performance metrics used to determine how a system performs.
2. Explain the main evaluation models used to evaluate a system.

OS12. Scripting [elective]

Topics:

- Scripting and the role of scripting languages
- Basic system commands
- Creating scripts, parameter passing
- Executing a script
- Influences of scripting on programming

Learning objectives:

1. Summarize a typical set of system commands provided by an operating system.
2. Demonstrate the typical functionality of a scripting language, and interpret the implications for programming.
3. Demonstrate the mechanisms for implementing scripts and the role of scripts on system implementation and integration.
4. Implement a simple script that exhibits parameter passing.

Net-Centric Computing (NC)

Recent advances in computer and telecommunications networking, particularly those based on TCP/IP, have increased the importance of networking technologies in the computing discipline. Net-centric computing covers a range of sub-specialties including: computer communication network concepts and protocols, multimedia systems, Web standards and technologies, network security, wireless and mobile computing, and distributed systems.

Mastery of this subject area involves both theory and practice. Learning experiences that involve hands-on experimentation and analysis are strongly recommended as they reinforce student understanding of concepts and their application to real-world problems. Laboratory experiments should involve data collection and synthesis, empirical modeling, protocol analysis at the source code level, network packet monitoring, software construction, and evaluation of alternative design models. All of these are important concepts that can best understood by laboratory experimentation.

NC1. Introduction to net-centric computing [core]

Minimum core coverage time: 2 hours

Topics:

- Background and history of networking and the Internet
- Network architectures
- The range of specializations within net-centric computing
    - Networks and protocols
    - Networked multimedia systems
    - Distributed computing
    - Mobile and wireless computing

Learning objectives:

1. Discuss the evolution of early networks and the Internet.
2. Demonstrate the ability to use effectively a range of common networked applications including e-mail, telnet, FTP, newsgroups, and web browsers, online web courses, and instant

messaging.
3. Explain the hierarchical, layered structure of a typical network architecture.
4. Describe emerging technologies in the net-centric computing area and assess their current capabilities, limitations, and near-term potential.

NC2. Communication and networking [core]

Minimum core coverage time: 7 hours

Topics:

- Network standards and standardization bodies
- The ISO 7-layer reference model in general and its instantiation in TCP/IP
- Circuit switching and packet switching
- Streams and datagrams
- Physical layer networking concepts (theoretical basis, transmission media, standards)
- Data link layer concepts (framing, error control, flow control, protocols)
- Internetworking and routing (routing algorithms, internetworking, congestion control)
- Transport layer services (connection establishment, performance issues)

Learning objectives:

1. Discuss important network standards in their historical context.
2. Describe the responsibilities of the first four layers of the ISO reference model.
3. Discuss the differences between circuit switching and packet switching along with the advantages and disadvantages of each.
4. Explain how a network can detect and correct transmission errors.
5. Illustrate how a packet is routed over the Internet.
6. Install a simple network with two clients and a single server using standard host-configuration software tools such as DHCP.

NC3. Network security [core]

Minimum core coverage time: 3 hours

Topics:

- Fundamentals of cryptography
- Secret-key algorithms
- Public-key algorithms
- Authentication protocols
- Digital signatures
- Examples

Learning objectives:

1. Discuss the fundamental ideas of public-key cryptography.
2. Describe how public-key cryptography works.
3. Distinguish between the use of private- and public-key algorithms.
4. Summarize common authentication protocols.
5. Generate and distribute a PGP key pair and use the PGP package to send an encrypted e-mail message.
6. Summarize the capabilities and limitations of the means of cryptography that are conveniently available to the general public.

NC4. The web as an example of client-server computing [core]

Minimum core coverage time: 3 hours

Topics:

- Web technologies
  - Server-side programs
  - Common gateway interface (CGI) programs
  - Client-side scripts
  - The applet concept
- Characteristics of web servers
  - Handling permissions
  - File management
  - Capabilities of common server architectures
- Role of client computers
- Nature of the client-server relationship
- Web protocols
- Support tools for web site creation and web management
- Developing Internet information servers
- Publishing information and applications

Learning objectives:

1. Explain the different roles and responsibilities of clients and servers for a range of possible applications.
2. Select a range of tools that will ensure an efficient approach to implementing various client-server possibilities.
3. Design and build a simple interactive web-based application (e.g., a simple web form that collects information from the client and stores it in a file on the server).

NC5. Building web applications [elective]

Topics:

- Protocols at the application layer
- Principles of web engineering
- Database-driven web sites
- Remote procedure calls (RPC)
- Lightweight distributed objects
- The role of middleware
- Support tools
- Security issues in distributed object systems
- Enterprise-wide web-based applications

Learning objectives:

1. Illustrate how interactive client-server web applications of medium size can be built using different types of Web technologies.
2. Demonstrate how to implement a database-driven web site, explaining the relevant technologies involved in each tier of the architecture and the accompanying performance tradeoffs.
3. Implement a distributed system using any two distributed object frameworks and compare them with regard to performance and security issues.
4. Discuss security issues and strategies in an enterprise-wide web-based application.

NC6. Network management [elective]

Topics:

- Overview of the issues of network management
- Use of passwords and access control mechanisms
- Domain names and name services
- Issues for Internet service providers (ISPs)
- Security issues and firewalls
- Quality of service issues: performance, failure recovery

Learning objectives:

1. Explain the issues for network management arising from a range of security threats, including viruses, worms, Trojan horses, and denial-of-service attacks
2. Summarize the strengths and weaknesses associated with different approaches to security.
3. Develop a strategy for ensuring appropriate levels of security in a system designed for a particular purpose.
4. Implement a network firewall.

NC7. Compression and decompression [elective]

Topics:

- Analog and digital representations
- Encoding and decoding algorithms
- Lossless and lossy compression
- Data compression: Huffman coding and the Ziv-Lempel algorithm
- Audio compression and decompression
- Image compression and decompression
- Video compression and decompression
- Performance issues: timing, compression factor, suitability for real-time use

Learning objectives:

1. Summarize the basic characteristics of sampling and quantization for digital representation.
2. Select, giving reasons that are sensitive to the specific application and particular circumstances, the most appropriate compression techniques for text, audio, image, and video information.
3. Explain the asymmetric property of compression and decompression algorithms.
4. Illustrate the concept of run-length encoding.
5. Illustrate how a program like the UNIX **`compress`** utility, which uses Huffman coding and the Ziv-Lempel algorithm, would compress a typical text file.

NC8. Multimedia data technologies [elective]

Topics:

- Sound and audio, image and graphics, animation and video
- Multimedia standards (audio, music, graphics, image, telephony, video, TV)
- Capacity planning and performance issues
- Input and output devices (scanners, digital camera, touch-screens, voice-activated)
- MIDI keyboards, synthesizers
- Storage standards (Magneto Optical disk, CD-ROM, DVD)
- Multimedia servers and file systems
- Tools to support multimedia development

Learning objectives:

1. For each of several media or multimedia standards, describe in non-technical language what the standard calls for, and explain how aspects of human perception might be sensitive to the limitations of that standard.
2. Evaluate the potential of a computer system to host one of a range of possible multimedia applications, including an assessment of the requirements of multimedia systems on the underlying networking technology.
3. Describe the characteristics of a computer system (including identification of support tools and appropriate standards) that has to host the implementation of one of a range of possible multimedia applications.
4. Implement a multimedia application of modest size.

NC9. Wireless and mobile computing [elective]

Topics:

- Overview of the history, evolution, and compatibility of wireless standards
- The special problems of wireless and mobile computing
- Wireless local area networks and satellite-based networks
- Wireless local loops
- Mobile Internet protocol
- Mobile aware adaption
- Extending the client-server model to accommodate mobility
- Mobile data access: server data dissemination and client cache management
- Software package support for mobile and wireless computing
- The role of middleware and support tools
- Performance issues
- Emerging technologies

Learning objectives:

1. Describe the main characteristics of mobile IP and explain how differs from IP with regard to mobility management and location management as well as performance.
2. Illustrate (with home agents and foreign agents) how e-mail and other traffic is routed using mobile IP.
3. Implement a simple application that relies on mobile and wireless data communications.
4. Describe areas of current and emerging interest in wireless and mobile computing, and assess the current capabilities, limitations, and near-term potential of each.

Programming Languages (PL)

A programming language is a programmer's principal interface with the computer. More than just knowing how to program in a single language, programmers need to understand the different styles of programming promoted by different languages. In their professional life, they will be working with many different languages and styles at once, and will encounter many different languages over the course of their careers. Understanding the variety of programming languages and the design tradeoffs between the different programming paradigms makes it much easier to master new languages quickly. Understanding the pragmatic aspects of programming languages also requires a basic knowledge of programming language translation and runtime features such as storage allocation.

PL1. Overview of programming languages [core]

Minimum core coverage time: 2 hours

Topics:

- History of programming languages
- Brief survey of programming paradigms
  - Procedural languages
  - Object-oriented languages
  - Functional languages
  - Declarative, non-algorithmic languages
  - Scripting languages
- The effects of scale on programming methodology

Learning objectives:

1. Summarize the evolution of programming languages illustrating how this history has led to the paradigms available today.
2. Identify at least one distinguishing characteristic for each of the programming paradigms

covered in this unit.
3. Evaluate the tradeoffs between the different paradigms, considering such issues as space efficiency, time efficiency (of both the computer and the programmer), safety, and power of expression.
4. Distinguish between programming-in-the-small and programming-in-the-large.

PL2. Virtual machines [core]

Minimum core coverage time: 1 hour

Topics:

- The concept of a virtual machine
- Hierarchy of virtual machines
- Intermediate languages
- Security issues arising from running code on an alien machine

Learning objectives:

1. Describe the importance and power of abstraction in the context of virtual machines.
2. Explain the benefits of intermediate languages in the compilation process.
3. Evaluate the tradeoffs in performance vs. portability.
4. Explain how executable programs can breach computer system security by accessing disk files and memory.

PL3. Introduction to language translation [core]

Minimum core coverage time: 2 hours

Topics:

- Comparison of interpreters and compilers
- Language translation phases (lexical analysis, parsing, code generation, optimization)
- Machine-dependent and machine-independent aspects of translation

Learning objectives:

1. Compare and contrast compiled and interpreted execution models, outlining the relative merits of each..
2. Describe the phases of program translation from source code to executable code and the files produced by these phases.
3. Explain the differences between machine-dependent and machine-independent translation and where these differences are evident in the translation process.

PL4. Declarations and types [core]

Minimum core coverage time: 3 hours

Topics:

- The conception of types as a set of values with together with a set of operations
- Declaration models (binding, visibility, scope, and lifetime)
- Overview of type-checking
- Garbage collection

Learning objectives:

1. Explain the value of declaration models, especially with respect to programming-in-the-large.
2. Identify and describe the properties of a variable such as its associated address, value, scope, persistence, and size.
3. Discuss type incompatibility.
4. Demonstrate different forms of binding, visibility, scoping, and lifetime management.
5. Defend the importance of types and type-checking in providing abstraction and safety.
6. Evaluate tradeoffs in lifetime management (reference counting vs. garbage collection).

PL5. Abstraction mechanisms [core]

Minimum core coverage time: 3 hours

Topics:

- Procedures, functions, and iterators as abstraction mechanisms
- Parameterization mechanisms (reference vs. value)
- Activation records and storage management
- Type parameters and parameterized types
- Modules in programming languages

Learning objectives:

1. Explain how abstraction mechanisms support the creation of reusable software components.
2. Demonstrate the difference between call-by-value and call-by-reference parameter passing.
3. Defend the importance of abstractions, especially with respect to programming-in-the-large.
4. Describe how the computer system uses activation records to manage program modules and their data.

PL6. Object-oriented programming [core]

Minimum core coverage time: 10 hours

Topics:

- Object-oriented design
- Encapsulation and information-hiding
- Separation of behavior and implementation
- Classes and subclasses
- Inheritance (overriding, dynamic dispatch)
- Polymorphism (subtype polymorphism vs. inheritance)
- Class hierarchies
- Collection classes and iteration protocols
- Internal representations of objects and method tables

Learning objectives:

1. Justify the philosophy of object-oriented design and the concepts of encapsulation, abstraction, inheritance, and polymorphism.
2. Design, implement, test, and debug simple programs in an object-oriented programming language.
3. Describe how the class mechanism supports encapsulation and information hiding.
4. Design, implement, and test the implementation of "is-a" relationships among objects using a class hierarchy and inheritance.
5. Compare and contrast the notions of overloading and overriding methods in an object-oriented language.
6. Explain the relationship between the static structure of the class and the dynamic structure of the instances of the class.
7. Describe how iterators access the elements of a container.

PL7. Functional programming [elective]

Topics:

- Overview and motivation of functional languages
- Recursion over lists, natural numbers, trees, and other recursively-defined data
- Pragmatics (debugging by divide and conquer; persistency of data structures)
- Amortized efficiency for functional data structures
- Closures and uses of functions as data (infinite sets, streams)

Learning objectives:

1. Outline the strengths and weaknesses of the functional programming paradigm.
2. Design, code, test, and debug programs using the functional paradigm.
3. Explain the use of functions as data, including the concept of closures.

PL8. Language translation systems [elective]

Topics:

- Application of regular expressions in lexical scanners
- Parsing (concrete and abstract syntax, abstract syntax trees)
- Application of context-free grammars in table-driven and recursive-descent parsing
- Symbol table management
- Code generation by tree walking
- Architecture-specific operations: instruction selection and register allocation
- Optimization techniques
- The use of tools in support of the translation process and the advantages thereof
- Program libraries and separate compilation
- Building syntax-directed tools

Learning objectives:

1. Describe the steps and algorithms used by language translators.
2. Recognize the underlying formal models such as finite state automata, push-down automata and their connection to language definition through regular expressions and grammars.
3. Discuss the effectiveness of optimization.
4. Explain the impact of a separate compilation facility and the existence of program libraries on the compilation process.

PL9. Type systems [elective]

Topics:

- Data type as set of values with set of operations
- Data types
    - Elementary types
    - Product and coproduct types
    - Algebraic types
    - Recursive types
    - Arrow (function) types
    - Parameterized types
- Type-checking models
- Semantic models of user-defined types
    - Type abbreviations
    - Abstract data types
    - Type equality
- Parametric polymorphism
- Subtype polymorphism
- Type-checking algorithms

Learning objectives:

1. Formalize the notion of typing.
2. Describe each of the elementary data types.
3. Explain the concept of an abstract data type.
4. Recognize the importance of typing for abstraction and safety.
5. Differentiate between static and dynamic typing.
6. Differentiate between type declarations and type inference.
7. Evaluate languages with regard to typing.

PL10. Programming language semantics [elective]

Topics:

- Informal semantics
- Overview of formal semantics
- Denotational semantics
- Axiomatic semantics
- Operational semantics

Learning objectives:

1. Explain the importance of formal semantics.
2. Differentiate between formal and informal semantics.
3. Describe the different approaches to formal semantics.
4. Evaluate the different approaches to formal semantics.

PL11. Programming language design [elective]

Topics:

- General principles of language design
- Design goals
- Typing regimes
- Data structure models
- Control structure models
- Abstraction mechanisms

Learning objectives:

1. Evaluate the impact of different typing regimes on language design, language usage, and the translation process.
2. Explain the role of different abstraction mechanisms in the creation of user-defined facilities.

CC2001 Report

December 15, 2001

Information Management (IM)

Information Management (IM) plays a critical role in almost all areas where computers are used. This area includes the capture, digitization, representation, organization, transformation, and presentation of information; algorithms for efficient and effective access and updating of stored information, data modeling and abstraction, and physical file storage techniques. It also encompasses information security, privacy, integrity, and protection in a shared environment. The student needs to be able to develop conceptual and physical data models, determine what IM methods and techniques are appropriate for a given problem, and be able to select and implement an appropriate IM solution that reflects all suitable constraints, including scalability and usability.

IM1. Information models and systems [core]

Minimum core coverage time: 3 hours

Topics:

- History and motivation for information systems
- Information storage and retrieval (IS&R)
- Information management applications
- Information capture and representation
- Analysis and indexing
- Search, retrieval, linking, navigation
- Information privacy, integrity, security, and preservation
- Scalability, efficiency, and effectiveness

Learning objectives:

1. Compare and contrast information with data and knowledge.
2. Summarize the evolution of information systems from early visions up through modern offerings, distinguishing their respective capabilities and future potential.
3. Critique/defend a small- to medium-size information application with regard to its satisfying real user information needs.
4. Describe several technical solutions to the problems related to information privacy, integrity, security, and preservation.
5. Explain measures of efficiency (throughput, response time) and effectiveness (recall, precision).
6. Describe approaches to ensure that information systems can scale from the individual to the global.

IM2. Database systems [core]

Minimum core coverage time: 3 hours

Topics:

- History and motivation for database systems
- Components of database systems
- DBMS functions
- Database architecture and data independence
- Use of a database query language

Learning objectives:

1. Explain the characteristics that distinguish the database approach from the traditional approach of programming with data files.
2. Cite the basic goals, functions, models, components, applications, and social impact of database systems.
3. Describe the components of a database system and give examples of their use.
4. Identify major DBMS functions and describe their role in a database system.
5. Explain the concept of data independence and its importance in a database system.
6. Use a query language to elicit information from a database.

IM3. Data modeling [core]

Minimum core coverage time: 4 hours

Topics:

- Data modeling
- Conceptual models (including entity-relationship and UML)
- Object-oriented model
- Relational data model

Learning objectives:

1. Categorize data models based on the types of concepts that they provide to describe the database structure -- that is, conceptual data model, physical data model, and representational data model.
2. Describe the modeling concepts and notation of the entity-relationship model and UML, including their use in data modeling.
3. Describe the main concepts of the OO model such as object identity, type constructors, encapsulation, inheritance, polymorphism, and versioning.
4. Define the fundamental terminology used in the relational data model .
5. Describe the basic principles of the relational data model.
6. Illustrate the modeling concepts and notation of the relational data model.

IM4. Relational databases [elective]

Topics:

- Mapping conceptual schema to a relational schema
- Entity and referential integrity
- Relational algebra and relational calculus

Learning objectives:

1. Prepare a relational schema from a conceptual model developed using the entity-relationship model
2. Explain and demonstrate the concepts of entity integrity constraint and referential integrity constraint (including definition of the concept of a foreign key).
3. Demonstrate use of the relational algebra operations from mathematical set theory (*union*, *intersection*, *difference*, and *cartesian product*) and the relational algebra operations developed specifically for relational databases (*select*, *product*, *join*, and *division*).
4. Demonstrate queries in the relational algebra.
5. Demonstrate queries in the tuple relational calculus.

IM5. Database query languages [elective]

Topics:

- Overview of database languages
- SQL (data definition, query formulation, update sublanguage, constraints, integrity)
- Query optimization
- QBE and 4th-generation environments
- Embedding non-procedural queries in a procedural language
- Introduction to Object Query Language

Learning objectives:

1. Create a relational database schema in SQL that incorporates key, entity integrity, and referential integrity constraints.
2. Demonstrate data definition in SQL and retrieving information from a database using the SQL **`SELECT`** statement.
3. Evaluate a set of query processing strategies and select the optimal strategy.
4. Create a non-procedural query by filling in templates of relations to construct an example of the desired query result.
5. Embed object-oriented queries into a stand-alone language such as C++ or Java (e.g., **`SELECT Col.Method() FROM Object`**).

IM6. Relational database design [elective]

Topics:

- Database design
- Functional dependency
- Normal forms (1NF, 2NF, 3NF, BCNF)
- Multivalued dependency (4NF)
- Join dependency (PJNF, 5NF)
- Representation theory

Learning objectives:

1. Determine the functional dependency between two or more attributes that are a subset of a relation.
2. Describe what is meant by 1NF, 2NF, 3NF, and BCNF.
3. Identify whether a relation is in 1NF, 2NF, 3NF, or BCNF.
4. Normalize a 1NF relation into a set of 3NF (or BCNF) relations and denormalize a relational schema.
5. Explain the impact of normalization on the efficiency of database operations, especially query optimization.
6. Describe what is a multivalued dependency and what type of constraints it specifies.
7. Explain why 4NF is useful in schema design.

IM7. Transaction processing [elective]

Topics:

- Transactions
- Failure and recovery
- Concurrency control

Learning objectives:

1. Create a transaction by embedding SQL into an application program.
2. Explain the concept of implicit commits.
3. Describe the issues specific to efficient transaction execution.
4. Explain when and why rollback is needed and how logging assures proper rollback.
5. Explain the effect of different isolation levels on the concurrency control mechanisms.
6. Choose the proper isolation level for implementing a specified transaction protocol.

IM8. Distributed databases [elective]

Topics:

- Distributed data storage
- Distributed query processing
- Distributed transaction model
- Concurrency control
- Homogeneous and heterogeneous solutions
- Client-server

Learning objectives:

1. Explain the techniques used for data fragmentation, replication, and allocation during the distributed database design process.
2. Evaluate simple strategies for executing a distributed query to select the strategy that minimizes the amount of data transfer.
3. Explain how the two-phase commit protocol is used to deal with committing a transaction that accesses databases stored on multiple nodes.
4. Describe distributed concurrency control based on the distinguished copy techniques and the voting method.
5. Describe the three levels of software in the client-server model.

IM9. Physical database design [elective]

Topics:

- Storage and file structure
- Indexed files
- Hashed files
- Signature files
- B-trees
- Files with dense index
- Files with variable length records
- Database efficiency and tuning

Learning objectives:

1. Explain the concepts of records, record types, and files, as well as the different techniques for placing file records on disk.
2. Give examples of the application of primary, secondary, and clustering indexes.
3. Distinguish between a nondense index and a dense index.
4. Implement dynamic multilevel indexes using B-trees.
5. Explain the theory and application of internal and external hashing techniques.
6. Use hashing to facilitate dynamic file expansion.
7. Describe the relationships among hashing, compression, and efficient database searches.
8. Evaluate costs and benefits of various hashing schemes.
9. Explain how physical database design affects database transaction efficiency.

IM10. Data mining [elective]

Topics:

- The usefulness of data mining
- Associative and sequential patterns
- Data clustering
- Market basket analysis
- Data cleaning
- Data visualization

Learning objectives:

1. Compare and contrast different conceptions of data mining as evidenced in both research and application.
2. Explain the role of finding associations in commercial market basket data.
3. Characterize the kinds of patterns that can be discovered by association rule mining.
4. Describe how to extend a relational system to find patterns using association rules.
5. Evaluate methodological issues underlying the effective application of data mining.
6. Identify and characterize sources of noise, redundancy, and outliers in presented data.
7. Identify mechanisms (on-line aggregation, anytime behavior, interactive visualization) to close the loop in the data mining process.
8. Describe why the various close-the-loop processes improve the effectiveness of data mining.

IM11. Information storage and retrieval [elective]

Topics:

- Characters, strings, coding, text
- Documents, electronic publishing, markup, and markup languages
- Tries, inverted files, PAT trees, signature files, indexing
- Morphological analysis, stemming, phrases, stop lists
- Term frequency distributions, uncertainty, fuzziness, weighting

- Vector space, probabilistic, logical, and advanced models
- Information needs, relevance, evaluation, effectiveness
- Thesauri, ontologies, classification and categorization, metadata
- Bibliographic information, bibliometrics, citations
- Routing and (community) filtering
- Search and search strategy, information seeking behavior, user modeling, feedback
- Information summarization and visualization
- Integration of citation, keyword, classification scheme, and other terms
- Protocols and systems (including Z39.50, OPACs, WWW engines, research systems)

Learning objectives:

1. Explain basic information storage and retrieval concepts.
2. Describe what issues are specific to efficient information retrieval.
3. Give applications of alternative search strategies and explain why the particular search strategy is appropriate for the application.
4. Perform Internet-based research.
5. Design and implement a small to medium size information storage and retrieval system.

IM12. Hypertext and hypermedia [elective]

Topics:

- Hypertext models (early history, web, Dexter, Amsterdam, HyTime)
- Link services, engines, and (distributed) hypertext architectures
- Nodes, composites, and anchors
- Dimensions, units, locations, spans
- Browsing, navigation, views, zooming
- Automatic link generation
- Presentation, transformations, synchronization
- Authoring, reading, and annotation
- Protocols and systems (including web, HTTP)

Learning objectives:

1. Summarize the evolution of hypertext and hypermedia models from early versions up through current offerings, distinguishing their respective capabilities and limitations.
2. Explain basic hypertext and hypermedia concepts.
3. Demonstrate a fundamental understanding of information presentation, transformation, and synchronization.
4. Compare and contrast hypermedia delivery based on protocols and systems used.
5. Design and implement web-enabled information retrieval applications using appropriate authoring tools.

IM13. Multimedia information and systems [elective]

Topics:

- Devices, device drivers, control signals and protocols, DSPs
- Applications, media editors, authoring systems, and authoring
- Streams/structures, capture/represent/transform, spaces/domains, compression/coding
- Content-based analysis, indexing, and retrieval of audio, images, and video
- Presentation, rendering, synchronization, multi-modal integration/interfaces
- Real-time delivery, quality of service, audio/video conferencing, video-on-demand

Learning objectives:

1. Describe the media and supporting devices commonly associated with multimedia information and systems.
2. Explain basic multimedia presentation concepts.
3. Demonstrate the use of content-based information analysis in a multimedia information system.
4. Critique multimedia presentations in terms of their appropriate use of audio, video, graphics, color, and other information presentation concepts.
5. Implement a multimedia application using a commercial authoring system.

IM14. Digital libraries [elective]

Topics:

- Digitization, storage, and interchange
- Digital objects, composites, and packages
- Metadata, cataloging, author submission
- Naming, repositories, archives
- Spaces (conceptual, geographical, 2/3D, VR)
- Architectures (agents, buses, wrappers/mediators), interoperability
- Services (searching, linking, browsing, and so forth)
- Intellectual property rights management, privacy, protection (watermarking)
- Archiving and preservation, integrity

Learning objectives:

1. Explain the underlying technical concepts in building a digital library.
2. Describe the basic service requirements for searching, linking, and browsing.
3. Critique scenarios involving appropriate and inappropriate use of a digital library, and determine the social, legal, and economic consequences for each scenario.
4. Describe some of the technical solutions to the problems related to archiving and preserving information in a digital library.
5. Design and implement a small digital library.

CC2001 Report

December 15, 2001

Social and Professional Issues (SP)

Although technical issues are obviously central to any computing curriculum, they do not by themselves constitute a complete educational program in the field. Students must also develop an understanding of the social and professional context in which computing is done.

This need to incorporate the study of social issues into the curriculum was recognized in the following excerpt from Computing Curricula 1991 [Tucker91]:

> Undergraduates also need to understand the basic cultural, social, legal, and ethical issues inherent in the discipline of computing. They should understand where the discipline has been, where it is, and where it is heading. They should also understand their individual roles in this process, as well as appreciate the philosophical questions, technical problems, and aesthetic values that play an important part in the development of the discipline.
>
> Students also need to develop the ability to ask serious questions about the social impact of computing and to evaluate proposed answers to those questions. Future practitioners must be able to anticipate the impact of introducing a given product into a given environment. Will that product enhance or degrade the quality of life? What will the impact be upon individuals, groups, and institutions?
>
> Finally, students need to be aware of the basic legal rights of software and hardware vendors and users, and they also need to appreciate the ethical values that are the basis for those rights. Future practitioners must understand the responsibility that they will bear, and the possible consequences of failure. They must understand their own limitations as well as the limitations of their tools. All practitioners must make a long-term commitment to remaining current in their chosen specialties and in the discipline of computing as a whole.

The material in this knowledge area is best covered through a combination of one required course along with short modules in other courses. On the one hand, some units listed as core -- in particular, SP2, SP3, SP4, and SP6 -- do not readily lend themselves to being covered in other traditional

courses. Without a standalone course, it is difficult to cover these topics appropriately. On the other hand, if ethical considerations are covered only in the standalone course and not "in context," it will reinforce the false notion that technical processes are void of ethical issues. Thus it is important that several traditional courses include modules that analyze ethical considerations in the context of the technical subject matter of the course. Courses in areas such as software engineering, databases, computer networks, and introduction to computing provide obvious context for analysis of ethical issues. However, an ethics-related module could be developed for almost any course in the curriculum. It would be explicitly against the spirit of the recommendations to have only a standalone course. Running through all of the issues in this area is the need to speak to the computer practitioner's responsibility to proactively address these issues by both moral and technical actions.

The ethical issues discussed in any class should be directly related to and arise naturally from the subject matter of that class. Examples include a discussion in the database course of data aggregation or data mining, or a discussion in the software engineering course of the potential conflicts between obligations to the customer and obligations to the user and others affected by their work. Programming assignments built around applications such as controlling the movement of a laser during eye surgery can help to address the professional, ethical and social impacts of computing.

There is an unresolved pedagogical conflict between having the core course at the lower (freshman-sophomore) level versus the upper (junior-senior) level. Having the course at the lower level

1. Allows for coverage of methods and tools of analysis (SP3) prior to analyzing ethical issues in the context of different technical areas
2. Assures that students who drop out early to enter the workforce will still be introduced to some professional and ethical issues.

On the other hand, placing the course too early may lead to the following problems:

1. Lower-level students may not have the technical knowledge and intellectual maturity to support in-depth ethical analysis. Without basic understanding of technical alternatives, it is difficult to consider their ethical implications.
2. Students need a certain level of maturity and sophistication to appreciate the background and issues involved. For that reason, students should have completed at least the discrete mathematics course and the second computer science course. Also, if students take a technical writing course, it should be a prerequisite or corequisite for the required course in the SP area.
3. Some programs may wish to use the course as a "capstone" experience for seniors.

Although items SP2 and SP3 are listed with a number of hours associated, they are fundamental to all the other topics. Thus, when covering the other areas, instructors should continually be aware of the social context issues and the ethical analysis skills. In practice, this means that the topics in SP2 and SP3 will be continually reinforced as the material in the other areas is covered.

SP1. History of computing [core]

Minimum core coverage time: 1 hour

Topics:

- Prehistory -- the world before 1946
- History of computer hardware, software, networking
- Pioneers of computing

Learning objectives:

1. List the contributions of several pioneers in the computing field.
2. Compare daily life before and after the advent of personal computers and the Internet.
3. Identify significant continuing trends in the history of the computing field.

SP2. Social context of computing [core]

Minimum core coverage time: 3 hours

Topics:

- Introduction to the social implications of computing
- Social implications of networked communication
- Growth of, control of, and access to the Internet
- Gender-related issues
- International issues

Learning objectives:

1. Interpret the social context of a particular implementation.
2. Identify assumptions and values embedded in a particular design.
3. Evaluate a particular implementation through the use of empirical data.
4. Describe positive and negative ways in which computing alters the modes of interaction between people.
5. Explain why computing/network access is restricted in some countries.

SP3. Methods and tools of analysis [core]

Minimum core coverage time: 2 hours

Topics:

- Making and evaluating ethical arguments
- Identifying and evaluating ethical choices
- Understanding the social context of design
- Identifying assumptions and values

Learning objectives:

1. Analyze an argument to identify premises and conclusion.
2. Illustrate the use of example, analogy, and counter-analogy in ethical argument.
3. Detect use of basic logical fallacies in an argument.
4. Identify stakeholders in an issue and our obligations to them.
5. Articulate the ethical tradeoffs in a technical decision.

SP4. Professional and ethical responsibilities [core]

Minimum core coverage time: 3 hours

Topics:

- Community values and the laws by which we live
- The nature of professionalism
- Various forms of professional credentialing and the advantages and disadvantages
- The role of the professional in public policy
- Maintaining awareness of consequences
- Ethical dissent and whistle-blowing
- Codes of ethics, conduct, and practice (IEEE, ACM, SE, AITP, and so forth)
- Dealing with harassment and discrimination
- "Acceptable use" policies for computing in the workplace

Learning objectives:

1. Identify progressive stages in a whistle-blowing incident.
2. Specify the strengths and weaknesses of relevant professional codes as expressions of professionalism and guides to decision-making.
3. Identify ethical issues that arise in software development and determine how to address them technically and ethically.
4. Develop a computer use policy with enforcement measures.
5. Analyze a global computing issue, observing the role of professionals and government officials in managing the problem.
6. Evaluate the professional codes of ethics from the ACM, the IEEE Computer Society, and other organizations.

SP5. Risks and liabilities of computer-based systems [core]

Minimum core coverage time: 2 hours

Topics:

- Historical examples of software risks (such as the Therac-25 case)

- Implications of software complexity
- Risk assessment and management

Learning objectives:

1. Explain the limitations of testing as a means to ensure correctness.
2. Describe the differences between correctness, reliability, and safety.
3. Discuss the potential for hidden problems in reuse of existing components.
4. Describe current approaches to managing risk, and characterize the strengths and shortcomings of each.

SP6. Intellectual property [core]

Minimum core coverage time: 3 hours

Topics:

- Foundations of intellectual property
- Copyrights, patents, and trade secrets
- Software piracy
- Software patents
- Transnational issues concerning intellectual property

Learning objectives:

1. Distinguish among patent, copyright, and trade secret protection.
2. Discuss the legal background of copyright in national and international law.
3. Explain how patent and copyright laws may vary internationally.
4. Outline the historical development of software patents.
5. Discuss the consequences of software piracy on software developers and the role of relevant enforcement organizations.

SP7. Privacy and civil liberties [core]

Minimum core coverage time: 2 hours

Topics:

- Ethical and legal basis for privacy protection
- Privacy implications of massive database systems
- Technological strategies for privacy protection
- Freedom of expression in cyberspace
- International and intercultural implications

Learning objectives:

1. Summarize the legal bases for the right to privacy and freedom of expression in one's own nation and how those concepts vary from country to country.
2. Describe current computer-based threats to privacy.
3. Explain how the Internet may change the historical balance in protecting freedom of expression.
4. Explain both the disadvantages and advantages of free expression in cyberspace.
5. Describe trends in privacy protection as exemplified in technology.

SP8. Computer crime [elective]

Topics:

- History and examples of computer crime
- "Cracking" ("hacking") and its effects
- Viruses, worms, and Trojan horses
- Crime prevention strategies

Learning objectives:

1. Outline the technical basis of viruses and denial-of-service attacks.
2. Enumerate techniques to combat "cracker" attacks.
3. Discuss several different "cracker" approaches and motivations.
4. Identify the professional's role in security and the tradeoffs involved.

SP9. Economic issues in computing [elective]

Topics:

- Monopolies and their economic implications
- Effect of skilled labor supply and demand on the quality of computing products
- Pricing strategies in the computing domain
- Differences in access to computing resources and the possible effects thereof

Learning objectives:

1. Summarize the rationale for antimonopoly efforts.
2. Describe several ways in which the information technology industry is affected by shortages in the labor supply.
3. Suggest and defend ways to address limitations on access to computing.
4. Outline the evolution of pricing strategies for computing goods and services.

SP10. Philosophical frameworks [elective]

Topics:

- Philosophical frameworks, particularly utilitarianism and deontological theories
- Problems of ethical relativism
- Scientific ethics in historical perspective
- Differences in scientific and philosophical approaches

Learning objectives:

1. Summarize the basic concepts of relativism, utilitarianism, and deontological theories.
2. Recognize the distinction between ethical theory and professional ethics.
3. Identify the weaknesses of the "hired agent" approach, strict legalism, naïve egoism, and naïve relativism as ethical frameworks.

Software Engineering (SE)

Software engineering is the discipline concerned with the application of theory, knowledge, and practice for effectively and efficiently building software systems that satisfy the requirements of users and customers. Software engineering is applicable to small, medium, and large-scale systems. It encompasses all phases of the life cycle of a software system. The life cycle includes requirement analysis and specification, design, construction, testing, and operation and maintenance.

Software engineering employs engineering methods, processes, techniques, and measurement. It benefits from the use of tools for managing software development; analyzing and modeling software artifacts; assessing and controlling quality; and for ensuring a disciplined, controlled approach to software evolution and reuse. Software development, which can involve an individual developer or a team of developers, requires choosing the tools, methods, and approaches that are most applicable for a given development environment.

The elements of software engineering are applicable to the development of software in any computing application domain where professionalism, quality, schedule, and cost are important in producing a software system.

SE1. Software design [core]

Minimum core coverage time: 8 hours

Topics:

- Fundamental design concepts and principles
- Design patterns
- Software architecture
- Structured design
- Object-oriented analysis and design

- Component-level design
- Design for reuse

Learning objectives:

1. Discuss the properties of good software design.
2. Compare and contrast object-oriented analysis and design with structured analysis and design.
3. Evaluate the quality of multiple software designs based on key design principles and concepts.
4. Select and apply appropriate design patterns in the construction of a software application.
5. Create and specify the software design for a medium-size software product using a software requirement specification, an accepted program design methodology (e.g., structured or object-oriented), and appropriate design notation.
6. Conduct a software design review using appropriate guidelines.
7. Evaluate a software design at the component level.
8. Evaluate a software design from the perspective of reuse.

SE2. Using APIs [core]

Minimum core coverage time: 5 hours

Topics:

- API programming
- Class browsers and related tools
- Programming by example
- Debugging in the API environment
- Introduction to component-based computing

Learning objectives:

1. Explain the value of application programming interfaces (APIs) in software development.
2. Use class browsers and related tools during the development of applications using APIs.
3. Design, implement, test, and debug programs that use large-scale API packages.

SE3. Software tools and environments [core]

Minimum core coverage time: 3 hours

Topics:

- Programming environments
- Requirements analysis and design modeling tools
- Testing tools
- Configuration management tools
- Tool integration mechanisms

Learning objectives:

1. Select, with justification, an appropriate set of tools to support the development of a range of software products.
2. Analyze and evaluate a set of tools in a given area of software development (e.g., management, modeling, or testing).
3. Demonstrate the capability to use a range of software tools in support of the development of a software product of medium size.

SE4. Software processes [core]

Minimum core coverage time: 2 hours

Topics:

- Software life-cycle and process models
- Process assessment models
- Software process metrics

Learning objectives:

1. Explain the software life cycle and its phases including the deliverables that are produced.
2. Select, with justification the software development models most appropriate for the development and maintenance of a diverse range of software products.
3. Explain the role of process maturity models.
4. Compare the traditional waterfall model to the incremental model, the object-oriented model, and other apropriate models.
5. For each of various software project scenarios, describe the project's place in the software life cycle, identify the particular tasks that should be performed next, and identify metrics appropriate to those tasks.

SE5. Software requirements and specifications [core]

Minimum core coverage time: 4 hours

Topics:

- Requirements elicitation
- Requirements analysis modeling techniques
- Functional and nonfunctional requirements
- Prototyping
- Basic concepts of formal specification techniques

Learning objectives:

1. Apply key elements and common methods for elicitation and analysis to produce a set of software requirements for a medium-sized software system.
2. Discuss the challenges of maintaining legacy software.
3. Use a common, non-formal method to model and specify (in the form of a requirements specification document) the requirements for a medium-size software system.
4. Conduct a review of a software requirements document using best practices to determine the quality of the document.
5. Translate into natural language a software requirements specification written in a commonly used formal specification language.

SE6. Software validation [core]

Minimum core coverage time: 3 hours

Topics:

- Validation planning
- Testing fundamentals, including test plan creation and test case generation
- Black-box and white-box testing techniques
- Unit, integration, validation, and system testing
- Object-oriented testing
- Inspections

Learning objectives:

1. Distinguish between program validation and verification.
2. Describe the role that tools can play in the validation of software.
3. Distinguish between the different types and levels of testing (unit, integration, systems, and acceptance) for medium-size software products.
4. Create, evaluate, and implement a test plan for a medium-size code segment.
5. Undertake, as part of a team activity, an inspection of a medium-size code segment.
6. Discuss the issues involving the testing of object-oriented software.

SE7. Software evolution [core]

Minimum core coverage time: 3 hours

Topics:

- Software maintenance
- Characteristics of maintainable software
- Reengineering
- Legacy systems

- Software reuse

Learning objectives:

1. Identify the principal issues associated with software evolution and explain their impact on the software life cycle.
2. Discuss the challenges of maintaining legacy systems and the need for reverse engineering.
3. Outline the process of regression testing and its role in release management.
4. Estimate the impact of a change request to an existing product of medium size.
5. Develop a plan for re-engineering a medium-sized product in response to a change request.
6. Discuss the advantages and disadvantages of software reuse.
7. Exploit opportunities for software reuse in a given context.

SE8. Software project management [core]

Minimum core coverage time: 3 hours

Topics:

- Team management
  - Team processes
  - Team organization and decision-making
  - Roles and responsibilities in a software team
  - Role identification and assignment
  - Project tracking
  - Team problem resolution
- Project scheduling
- Software measurement and estimation techniques
- Risk analysis
- Software quality assurance
- Software configuration management
- Project management tools

Learning objectives:

1. Demonstrate through involvement in a team project the central elements of team building and team management.
2. Prepare a project plan for a software project that includes estimates of size and effort, a schedule, resource allocation, configuration control, change management, and project risk identification and management.
3. Compare and contrast the different methods and techniques used to assure the quality of a software product.

SE9. Component-based computing [elective]

Topics:

- Fundamentals
  - The definition and nature of components
  - Components and interfaces
  - Interfaces as contracts
  - The benefits of components
- Basic techniques
  - Component design and assembly
  - Relationship with the client-server model and with patterns
  - Use of objects and object lifecycle services
  - Use of object brokers
  - Marshalling
- Applications (including the use of mobile components)
- Architecture of component-based systems
- Component-oriented design
- Event handling: detection, notification, and response
- Middleware
  - The object-oriented paradigm within middleware
  - Object request brokers
  - Transaction processing monitors
  - Workflow systems
  - State-of-the-art tools

Learning objectives:

1. Explain and apply recognized principles to the building of high-quality software components.
2. Discuss and select an architecture for a component-based system suitable for a given scenario.
3. Identify the kind of event handling implemented in one or more given APIs.
4. Explain the role of objects in middleware systems and the relationship with components.
5. Apply component-oriented approaches to the design of a range of software including those required for concurrency and transactions, reliable communication services, database interaction including services for remote query and database management, secure communication and access.

SE10. Formal methods [elective]

Topics:

- Formal methods concepts
- Formal specification languages
- Executable and non-executable specifications
- Pre and post assertions
- Formal verification

Learning objectives:

1. Apply formal verification techniques to software segments with low complexity.
2. Discuss the role of formal verification techniques in the context of software validation and testing.
3. Explain the potential benefits and drawbacks of using formal specification languages.
4. Create and evaluate pre- and post-assertions for a variety of situations ranging from simple through complex.
5. Using a common formal specification language, formulate the specification of a simple software system and demonstrate the benefits from a quality perspective.

SE11. Software reliability [elective]

Topics:

- Software reliability models
- Redundancy and fault tolerance
- Defect classification
- Probabilistic methods of analysis

Learning objectives:

1. Demonstrate the ability to apply multiple methods to develop reliability estimates for a software system.
2. Identify and apply redundancy and fault tolerance for a medium-sized application.
3. Explain the problems that exist in achieving very high levels of reliability.
4. Identify methods that will lead to the realization of a software architecture that achieves a specified reliability level.

SE12. Specialized systems development [elective]

Topics:

- Real-time systems
- Client-server systems
- Distributed systems
- Parallel systems
- Web-based systems
- High-integrity systems

Learning objectives:

1. Identify and discuss different specialized systems.
2. Discuss life cycle and software process issues in the context of software systems designed for a specialized context.

3. Select, with appropriate justification, approaches that will result in the efficient and effective development and maintenance of specialized software systems.
4. Given a specific context and a set of related professional issues, discuss how a software engineer involved in the development of specialized systems should respond to those issues.
5. Outline the central technical issues associated with the implementation of specialized systems development.

Computational Science and Numerical Methods (CN)

From the earliest days of the discipline, numerical methods and the techniques of scientific computing have constituted a major area of computer science research. As computers increase in their problem-solving power, this area -- like much of the discipline -- has grown in both breadth and importance. At the end of the millennium, scientific computing stands as an intellectual discipline in its own right, closely related to but nonetheless distinct from computer science.

Although courses in numerical methods and scientific computing are extremely valuable components of an undergraduate program in computer science, the CC2001 Task Force believes that none of the topics in this area represent core knowledge. From our surveys of curricula and interaction with the computer science education community, we are convinced no consensus exists that this material is essential for all CS undergraduates. It remains a vital part of the discipline, but need not be a part of every program.

For those who choose to pursue it, this area offers exposure to many valuable ideas and techniques, including precision of numerical representation, error analysis, numerical techniques, parallel architectures and algorithms, modeling and simulation, and scientific visualization. At the same time, students who take courses in this area have an opportunity to apply these techniques in a wide range of application areas, such as the following:

- Molecular dynamics
- Fluid dynamics
- Celestial mechanics
- Economic forecasting
- Optimization problems
- Structural analysis of materials
- Bioinformatics
- Computational biology
- Geologic modeling
- Computerized tomography

Each of the units in this area corresponds to a full-semester course at most institutions. The level of specification of the topic descriptions and the learning objectives is therefore different from that used in other areas in which the individual units typically require smaller blocks of time.

CN1. Numerical analysis [elective]

Topics:

- Floating-point arithmetic
- Error, stability, convergence
- Taylor's series
- Iterative solutions for finding roots (Newton's Method)
- Curve fitting; function approximation
- Numerical differentiation and integration (Simpson's Rule)
- Explicit and implicit methods
- Differential equations (Euler's Method)
- Linear algebra
- Finite differences

Learning objectives:

1. Compare and contrast the numerical analysis techniques presented in this unit.
2. Define error, stability, machine precision concepts. and the inexactness of computational approximations.
3. Identify the sources of inexactness in computational approximations.
4. Design, code, test, and debug programs that implement numerical methods.

CN2. Operations research [elective]

Topics:

- Linear programming
    - Integer programming
    - The Simplex method
- Probablistic modeling
- Queueing theory
    - Petri nets
    - Markov models and chains
- Optimization
- Network analysis and routing algorithms
- Prediction and estimation
    - Decision analysis
    - Forecasting
    - Risk management
    - Econometrics, microeconomics
    - Sensitivity analysis
- Dynamic programming
- Sample applications
- Software tools

Learning objectives:

1. Apply the fundamental techniques of operations research.

2. Describe several established techniques for prediction and estimation.
3. Design, code, test, and debug application programs to solve problems in the domain of operations research.

CN3. Modeling and simulation [elective]

Topics:

- Random numbers
  - Pseudorandom number generation and testing
  - Monte Carlo methods
  - Introduction to distribution functions
- Simulation modeling
  - Discrete-event simulation
  - Continuous simulation
- Verification and validation of simulation models
  - Input analysis
  - Output analysis
- Queueing theory models
- Sample applications

Learning objectives:

1. Discuss the fundamental concepts of computer simulation.
2. Evaluate models for computer simulation.
3. Compare and contrast methods for random number generation.
4. Design, code, test, and debug simulation programs.

CN4. High-performance computing [elective]

Topics:

- Introduction to high-performance computing
  - History and importance of computational science
  - Overview of application areas
  - Review of required skills
- High-performance computing
  - Processor architectures
  - Memory systems for high performance
  - Input/output devices
  - Pipelining
  - Parallel languages and architectures
- Scientific visualization
  - Presentation of results
  - Data formats

    - Visualization tools and packages
- Sample problems
    - Ocean and atmosphere models
    - Seismic wave propagation
    - N-body systems (the Barnes-Hut algorithm)
    - Chemical reactions
    - Phase transitions
    - Fluid flow

Learning objectives:

1. Recognize problem areas where computational modeling enhances current research methods.
2. Compare and contrast architectures for scientific and parallel computing, recognizing the strengths and weaknesses of each.
3. Implement simple performance measurements for high-performance systems.
4. Design, code, test, and debug programs using techniques of numerical analysis, computer simulation, and scientific visualization.

Knowledge Units for Computer Science

Computing Curricula 1991 see Computing Curricula 2001

Contents

Cognates

Advanced Topics

Last update:

Send comments to: webmaster@cs.wwc.edu

AL: Algorithms and Data Structures

There are approximately 47 hours of lectures recommended for this set of knowledge units.

The knowledge units in the common requirements for the subject area of Algorithms and Data Structures emphasize the following topics: basic data structures, abstract data types, recursive algorithms, complexity analysis, sorting and searching, computability and undecidability, problem-solving strategies, and parallel and distributed algorithms.

\input{AL/Handouts/ADTs}

Last update:

Send comments to: webmaster@cs.wwc.edu

AR: Architecture

There are approximately 59 hours of lectures recommended for this set of knowledge units

The knowledge units in the common requirements for the subject area of Architecture emphasize the following topics: digital logic, digital systems, machine level representation of data, assembly level machine organization, memory system organization and architecture, interfacing and communication, and alternative architectures

Sections

Last update:

Send comments to: webmaster@cs.wwc.edu

AI: Artificial Intelligence and Robotics

There are approximately nine hours of lectures recommended for this set of knowledge units.

The knowledge units in the common requirements for the subject area of Artificial Intelligence and Robotics emphasize the following topics history and applications of artificial intelligence; problems, state spaces and search strategies. While this coverage is minimal, it does provide a sufficient introduction to artificial intelligence that will allow students to decide whether or not to pursue further studies in this area.

1. AI1: History and Applications of AI
2. AI2: Problems, State Spaces, and Search Strategies

AI1 Lecture Notes

Last update:

Send comments to: webmaster@cs.wwc.edu

DB: Database and Information Retrieval

There are approximately nine hours of lectures recommended for this set of knowledge units.

The knowledge units in the common requirements for the subject area of Database and Information Retrieval emphasize the following topics overview and applications of database systems, conceptual modeling, and the relational data model.

1. DB1: Overview, Models, and Applications of Database Systems
2. DB2: The Relational Data Model

Lecture notes

Last update:

Send comments to: webmaster@cs.wwc.edu

HU: Human-Computer Communication

There are approximately eight hours of lectures recommended for this set of knowledge units.

The knowledge units in the common requirement for the subject area of Human Computer Communication are directed toward providing the student with knowledge of user interfaces and fundamentals of computer graphics. The following topics are emphasized: input/output devices, use and construction of interfaces, and basic graphics concepts. Since students will gain significant experience with a variety of computing systems and their user interfaces throughout their education, HU knowledge units do not cover this subject area extensively.

1. HU1
2. HU2

Last update:

Send comments to: webmaster@cs.wwc.edu

NU: Numerical and Symbolic Computing

There are approximately seven hours of lectures recommended for this set of knowledge units.

The knowledge units in the common requirements for the subject area of Numerical and Symbolic Computation emphasize the following topics: number representation errors, portability, and iterative approximation methods. While this coverage is surely minimal, many additional topics in the Numerical and Symbolic Computation subject area will naturally occur in other parts of the curriculum, especially in mathematics.

1. NU1
2. NU2

Last update:

Send comments to: webmaster@cs.wwc.edu

OS: Operating Systems

There are approximately 31 hours of lectures recommended for this set of knowledge units.

The knowledge units in the common requirements for the subject area of Operating Systems emphasize the following topics history, evolution, and philosophies; tasking and processes; process coordination and synchronization; scheduling and dispatch; physical and virtual memory organization; device management; file systems and naming; security and protection; communications and networking; distributed operating systems; and real-time concerns.

Last update:

Send comments to: webmaster@cs.wwc.edu

Programming Languages

There are approximately, 46 hours of lectures recommended for this set of knowledge units.

The knowledge units in the common requirements for the subject area of Programming Languages emphasize the following topics: history; virtual machines; representation of data types; sequence control; data control, sharing, and type checking; run-time storage management; finite state automata and regular expressions; context-free grammars and pushdown automata; language translation systems; semantics; programming paradigms; and distributed and parallel programming constructs.

1. PL1: History and Overview
2. PL2: Virtual Machines
3. PL3: Representation of Data Types
4. PL4: Sequence Control
5. PL5: Data Control, Sharing, and Type Checking
6. PL6: Run-time Storage Management
7. PL7: Finite State Automata and Regular Expressions
8. PL8: Context-free Grammars and Pushdown Automata
9. PL9: Language Translation Systems
10. PL10: Programming Language Semantics
11. PL11: Programming Paradigms
12. PL12: Distributed and Parallel Programming Constructs

Last update:

Send comments to: webmaster@cs.wwc.edu

SE: Software Methodology and Engineering

There are approximately 44 hours of lectures recommended for this set of knowledge units.

The knowledge units in the common requirements for the subject area of Software Methodology and Engineering emphasize the following topics: fundamental problem solving concepts, the software development process, Software specifications, software design and implementation, verification, and validation.

1. SE1: Fundamental Problem-Solving Concepts
2. SE2: The Software Development Process
3. SE3: Software Requirements and Specifications
4. SE4: Software Design and Implementation
5. SE5: Verification and Validation

Last update:

Send comments to: webmaster@cs.wwc.edu

SP: Social, Ethical, and Professional Issues

There are approximately 11 hours of lectures recommended for this set of knowledge units. The knowledge units in the common requirements for the subject area of Social and Ethical Issues emphasize the following topics: historical and social context, professional responsibilities, risks and liabilities, and intellectual property. Note: While there are no laboratories listed for knowledge units SP1-SP4, the following kinds of activities should accompany their coverage in a course of instruction:

1. Write a short expository paper, with references, that demonstrates understanding of the historical or social context of some specific aspect of computing (e.g., computers in medicine, computerization and work, information access and privacy, public interest in computer records, the societal role of research).
2. Write a short paper, with references, that discusses an incidence of misuse of computers or information technology
3. Discuss particular aspects of professionalism in a seminar setting. Draw conclusions about ethical and societal dimensions of the profession.
4. Write or discuss a short paper discussing methods of risk assessment and reduction, and their role in the design process.
5. Present a case study in copyright or patent violation as a seminar discussion, with an accompanying writing assignment that demonstrates student understanding of the principles.

Sections

Resources

1. SPA's Competition Principles
2. SP/ICCPE

Last update:

Send comments to: webmaster@cs.wwc.edu

PR: Introduction to a Programming Language

An introduction to the syntactic and execution characteristics of a modern programming language, along with its use in the construction and execution of complete programs that solve simple algorithmic problems.

Recurring Concepts: conceptual and formal models, efficiency, evolution, reuse, trade-offs and consequences.

Lecture Topics: (12 hours minimum)

1. Basic type declarations (e.g., integer, real, boolean, char, string)
2. Arithmetic operators and assignment
3. Conditional statements
4. Loops and recursion
5. Procedures, functions, and parameters
6. Array; and records
7. Overall program structure

Suggested Laboratories: (open or closed) Students should develop and run three or four programs that solve elementary algorithmic problems. Experience with compiling, finding and correcting syntax errors, and executing programs will be gained.

Connections:

- Related to: SE1
- Prerequisites:
- Requisite for:

- Generic Language Description
- Fortran
- Godel
- Haskell
- Pascal
- Prolog
- Scheme
- SML

Last update:

Send comments to: webmaster@cs.wwc.edu

Mathematics

The following documents use MathML Amaya is an appropriate bowser.

Notation

Define:y = 3x+4
Identical:A=B
Congruent: different names
Similar:

Discrete Mathematics

- Functions, Relations and Sets
  - Functions (surjections, injections, inverses, composition)
  - Relations (reflexivity, symmetry, transitivity, equivalence relations)
  - Sets (Venn diagrams, complements, Cartesian products, power sets)
  - Pigeonhole principle
  - Cardinality and countability
- Basic logic
  - Propositional logic
  - Logical connectives
  - Truth tables
  - Normal forms (conjunctive and disjunctive)
  - Validity
  - Predicate logic
  - Universal and existential quantification
  - Modus ponens and modus tollens
  - Limitations of predicate logic
- Proof techniques
  - Notions of implication, converse, inverse, contrapositive, negation, and contradiction
  - The structure of formal proofs
  - Direct proofs
  - Proof by counterexample
  - Proof by contraposition
  - Proof by contradiction
  - Mathematical induction
  - Strong induction
  - Recursive mathematical definitions
  - Well orderings
- Basics of counting

    - Counting arguments
    - The pigeonhole principle
    - Permutations and combinations
    - Solving recurrence relations (common examples, the Master Theorem)
- Graphs and Trees
    - Undirected graphs
    - Directed graphs
    - Trees
    - Spanning trees
    - Traversal strategies
- Discrete Probability
    - Finite probability space, probability measure, events
    - Conditional probability, independence, Bayes' rule
    - Integer random variables, expectation

Additional material

- Mathematical logic -- propositional and predicate logic possibly modal & non-monotonic logics
    - Proof theory (syntax)
        - Operators: not, and, or, if, iff, All x, Exists x.
        - Normal forms: conjunctive, disjunctive, prenex
        - Inference rules
        - Consistency, soundness, completeness
        - Analytic tableau
    - Model theory (semantics)
        - truth tables
        - Herbrand semantics
    - Proof/disproof Techniques
        - Proof by induction
            - basis, induction step, inductive assumption
            - induction on number of elements, length of formulae...
        - Proof by contradiction
        - Counter example
- Algebra (Many sorted) -- used for the specification of ADTs
    - Domains
    - Semantic functions
    - Semantic equations/axioms
- Elementary combinatorics including graph theory and counting arguments
- Elementary discrete mathematics including number theory, discrete probability, recurrence relations

Probability and Statistics

- Discrete Probability

- Statistics

Additional Mathematics

- Calculus
- Linear Algebra
- Number theory
- Symbolic Logic

Computational Science and Numerical Methods (CN)

- CN1. Numerical analysis
  - Floating-point arithmetic
  - Error, stability, convergence
  - Taylor's series
  - Iterative solutions for finding roots (Newton's Method)
  - Curve fitting; function approximation
  - Numerical differentiation and integration (Simpson's Rule)
  - Explicit and implicit methods
  - Differential equations (Euler's Method)
  - Linear algebra
  - Finite differences
  - Note: Old stuff to be integrated
    - Computer arithmetic, including number representations, roundoff, overflow and underflow
    - Classical numerical algorithms
    - Iterative approximation methods
- Operations Research
- Modeling and simulation
- High-performance computing

Maintained by WWC CS Department

Last Modified

Send comments to webmaster@cs.wwc.edu

Science

Physics

The scientific method

Research loop

- observe (data collection and analysis)
- theorize (synthesis)
- test (experiment/validate)

Maintained by WWC CS Department
Last Modified
Send comments to webmaster@cs.wwc.edu

Details of the Advanced Supplemental Material

The following descriptions are intended to give detailed guidance for the design of advanced and supplemental courses in certain topic areas of the discipline. Each one contains a topic summary, an overview of laboratory work (if appropriate), and the prerequisites from the common requirements and other subjects that should precede advanced/supplemental study of the topic. (Recall from section 8 that these descriptions cover only 11 of the 28 advanced and supplemental topics that were listed there. Advanced and supplemental courses for the remaining topic areas in that list can also be offered in undergraduate programs.)

Advanced Operating Systems -- 4

Advanced Software Engineering -- 4

Topic Summary: This topic area is devoted to methods and tools that increase the quality and decrease the cost of developing and maintaining complex software systems. It includes activities that cover the whole spectrum of the software development life cycle.

Subtopics include process and life-cycle models; specification methods notations, and tools; Validation and Verification; debugging and program understanding; quality assurance; testing paradigms (i.e., unit, regression); testing strategies (e.g., white box, functional); metrics; tools (CASE); prototyping; Version control; configuration Management; end-user considerations; standards and international issues; documentation; maintenance; reuse; safety; reliability; portability; and project organization (e.g., cost estimates schedules, economic models.

Suggested Laboratories: A software design and implementation project, preferably done as a group project, is an effective laboratory component of this topic area. Possible projects might include analysis specification and high-level design; modifying and updating an existing piece of software; testing and integrating separate components; or doing a full design and implementation task. Students should also see and use a variety of contemporary software design tools

Prerequisites: AL1, AL2, NU1, PL1-PL6, PL11, SE (all), SP2-SP4, Discrete Mathematics

Analysis of Algorithms -- 4

Artificial Intelligence -- 4

Topic Summary: A selective survey of key concepts and applications of artificial intelligence and an in-depth experience with a language commonly used for building AI systems (e.g. Lisp or Prolog).

Subtopics include knowledge representation, state space/searching, heuristic search, expert systems, expert system shells, natural language processing, propositional logic, learning and cognitive models, and vision.

Suggested Laboratories: Students will implement, modify, or enhance several AI systems using an AI language and associated tool (e.g., expert system shells, knowledge acquisition tools).

Prerequisite: AL1-AL3, AI1, AI2, PL11, SE1, SE2, Discrete Mathematics.

Combinatorial and Graph Algorithms -- 4

Computational Complexity -- 4

Computer Communication Networks -- 4

Topic Summary: This topic gives students a foundation in the study of computer networks. Current methods and practices in the use of computer networks to enable communication are covered. Also covered are the physical and architectural elements and information layers of a communication network, along with diagnostic, design, operational, and performance measurement tools that are used to implement, operate, and tune such a network. Different network architectures are contrasted, and compared with traditional mainframe and time-shared computer models

Important subtopics include network architecture and communication protocols, network elements, data link, switching and routing, end-to-end protocols, LANs, and data security.

Suggested Laboratories: Case studies of existing network architectures and protocols (e.g., Ethernet, Wangnet, and FDDI) provide valuable laboratory work. Hardware communication devices and performance measurement tools should also be used in directed laboratory exercises.

Prerequisite: Significant coverage of the common requirements in the AR area (i.e., AR5-AR7), the OS area (i.e., OS3, OS4, OS7-OS11), and the PL area (i.e., PL5, PL6, PL12).

Computer Graphics -- 4

Topic Summary: An overview of the principles and methodologies of computer graphics, including the representation, manipulation, and display of two- and three-dimensional objects.

Subtopics include characteristics of display devices (e.g., raster, vector); representing primitive objects (lines, curves, surfaces) and composite objects; two- and three-dimensional transformations (translation, rotations, scaling); hidden lines and surfaces; shading and coloring; interactive graphics and the user interface; animation techniques.

Suggested Laboratories: Students Should have access to a suite of graphics software tools and a high quality color display. Exercises will provide experience with the design, implementation, and evaluation of programs that manipulate and display graphic objects.

Prerequisites: HU2, SE1, SE2, Calculus, Linear Algebra, Discrete Mathematics.

Computer Human Interface -- 4

Computer Security -- 4

Topic Summary: This area addresses the problem of how to secure computer systems, networks, and data from unauthorized or accidental access, modification, and denial of service. Courses designed for this area will draw material from the following subtopics: formal definitions of security, privacy, and integrity; risk assessment and management; information theory; information flow and covert channels; coding and cryptography.

Additional subtopics Include: authentication methods; capabilities, access lists, and protection demoralize; standards; malicious software (e.g., viruses, logic bombs); audit and control methods; legal factory; database and inference control; security kernals; and verification methods.

Suggested Laboratories: None specifically recommended; however, this area it especially well-suited to the use of case studies to reinforce various principles covered above.

Prerequisites: AL4, AL5, AL7, DB (all), OS (all), SE5, SP3, Calculus, Linear Algebra.

Database and Information Retrieval -- 4

Topic Summary: This topic in includes the material normally found in a first course in database systems, as well as several advanced subtopics. Basic coverage includes data models (E-R, relational, and object-oriented); query languages (relational algebra, relational calculus); the data dictionary, implementation of a relational database kernel; and case studies of commercial database languages and systems.

Additional subtopics Include: query optimization; theory of normal form and database design; transaction processing, concurrency control, and recovery from failure; security and integrity; distributed database systems; language paradigms and database languages; user interfaces and graphical query languages; advanced study of physical database organization; emerging database technologies (e.g., hypertext and knowledge-based systems); and logic as a data model.

Suggested Laboratories: Open and closed laboratories can be assigned for students to gain experience with relational database kernel implementation using an imperative language (e.g. C++ or Ada), and with implementing parts of a particular database management system. Other similar labs can be designed to cover various other subtopics listed above.

Prerequisites: DB1-DB2, HU1, OS7-OS10, SE1, SE2, Discrete Mathematics.

Digital Design Automation -- 4

Fault-tolerant computing -- 4

Information Theory -- 4

Modeling and Simulation -- 4

Numerical Computation -- 4

Parallel and Distributed Computing -- 4

Topic Summary: This topic involves the design, structure, and use of systems having interacting processors. It includes concepts from most of the nine subject areas of the discipline of computing. Concepts from AL, PL, AR, OS, and SE are important for the basic support of parallel and distributed systems, while concepts from NU, DB, Al, and HU are important in many applications.

Subtopics include concurrency and synchronization; architectural support; programming language constructs for parallel computing; parallel algorithms and computability; messages vs. remote procedure calls vs. shared memory models, structural alternatives (e.g., master-slave, client-server, fully distributed, cooperating objects); coupling (tight vs. loose); naming and winding; verification, validation, and maintenance issues; fault tolerance and reliability; replication and avoidability; security; standards and protocol; temporal concerns (persistence, serializability); data coherence; load balancing and scheduling; appropriate applications.

Suggested Laboratories: Programming assignments should ideally be developed on a multiprocessor or simulated parallel processing architecture.

Prerequisites: AL9, AR6, AR7, OS (all), PL11, PL12, SE3, SE5.

Performance Prediction and Analysis -- 4

Principles of Computer Architecture -- 4

Principles of Programming Languages -- 4

Programming Language Translation -- 4

Topic Summary: This topic is an in depth study of the principles and design aspects of programming language translation. The major components of a compiler are discussed; lexical analysis, syntactic analysis, type checking, code generation, and optimization. Alternative parsing strategies (e.g., top-down, LR, recursive descent) are presented and compared with respect to space and time tradeoffs.

Subtopics include ambiguity, data representation, recovery, symbol table design, binding, compiler generation tools (e.g., LEX and YACC), syntax directed editors, linkers, loaders, incremental compiling, and interpreters.

Suggested Laboratories: Laboratory exercises will assist students in reinforcing concepts by designing and implementing components of a compiler for a small but representative language. Alternative parsing strategies will be implemented and their performance compared. Laboratory work for this course is well suited to team projects.

Prerequisites: AR3, AR4, PL2-PL10, SE2.

\section{Attribute Grammars and Static Semantics} Context-free grammars are not able to completely specify the structure of programming languages. For example, declaration of names before reference, number and type of parameters in procedures and functions, the correspondence between formal and actual parameters, name or structural equivalence, scope rules, and the distinction between identifiers and reserved words are all structural aspects of programming languages which cannot be specified using context-free grammars. These {\em context-sensitive} aspects of the grammar are often called the {\em static semantics} of the language. The term {\em dynamic semantics} is used to refer to semantics proper, that is, the relationship between the syntax and the computational model. Even in a simple language like Simp, context-free grammars are unable to specify that variables appearing in expressions must have an assigned value. Context-free descriptions of syntax are supplemented with natural language descriptions of the static semantics or are extended to become attribute grammars. Attribute grammars are an extension of context-free grammars which permit the specification of context-sensitive properties of programming languages. Attribute grammars are actually much more powerful and are fully capable of specifying the semantics of programming languages as well. For an example, the following partial syntax of an imperative programming language requires the declaration of variables before reference to the variables. \begin{center}\parbox{4.5in}{ \begin{tabbing} P ::= D B \\ D ::= V... \\ B ::= C ... \\ C ::= V := E $|$ ... \end{tabbing}} \end{center} However, this context-free syntax does not indicate this restriction. The declarations define an environment in which the body of the program executes. Attribute grammars permit the explicit description of the environment and its interaction with the body of the program. Since there is no generally accepted notation for attribute grammars, attribute grammars will be represented as context-free grammars which permit the parameterization of non-terminals and the addition of where statements which provide further restrictions on the parameters. Figure~\ref{ag:decl} is an attribute grammar for declarations. \begin{lfig} \label{ag:decl} \begin{tabbing} 123456789012\=123456\=789012345678901234567890\=1234567890\kill P ::= D(Env$\uparrow$) B(Env$\downarrow$)\\ D(Env$\uparrow$) ::= ...V$_i$(Env$_{i-1}\downarrow$,Env$_i\uparrow$)...\\ \>where Env$_0$ = \emptyset$, Env = Env$_n$ and \\ \>\> Env$_i$ = Env$_{i-1} \cup \{{\rm V}_i\}$\\ B(Env$\downarrow$) ::= C(Env$\downarrow$)... \\ C(Env$\downarrow$) ::= V := E(Env$\downarrow$) $|$ ... \\ \> where V $\in$ Env \end{tabbing} \caption{An attribute grammar for declarations} \end{lfig} The parameters marked with $\downarrow$ are called inherited attributes and denote attributes which are passed down the parse tree while the parameters marked with $\uparrow$ are called synthesized attributes and denote attributes which are passed up the parse tree. Attribute grammars have considerable expressive power beyond there use to specify context sensitive portions of the syntax and may be used to specify: \begin{itemize} \item context sensitive rules \item evaluation of expressions \item translation

\end{itemize} \section{Further Reading} The original paper on attribute grammars was by Knuth\cite{Knuth68}. For a more recent source and their use in compiler construction and compiler generators see \cite{DJL88,PittPet92}

Real-time Systems -- 4

Robotics and Machine Intelligence -- 4

Semantics and Verification -- 4

Societal Impact of Computing -- 4

Symbolic Computation -- 4

Topic Summary: This topic provides coverage of the foundations and uses of algebraic systems, as well as insights into current methods for effectively using computers to do symbolic computation. Students should be able to understand basic symbolic computations and their underlying data structures and algorithms. Using a currently available system, students will be able to solve mathematical problems symbolically. The role of symbolic computation in the discipline of computing and related disciplines, as will as its strengths and limitations should also be taught.

Subtopics include computer algebraic systems; data representations; fundamental algorithms (e.g., matrix calculation, Taylor series, differentiation); polynomial simplification; advanced algorithms (e.g. modular methods for GCD, matrix inversion, polynomial factorization); formal integration.

Suggested Laboratories: Exercises should be given so that students can use a contemporary symbol manipulation system (e.g. MACSYMA, REDUCE, Mathematica) to solve problems.

Prerequisites: AL1, AL4, AL8, AR3, AI2, NU1, NU2, PL3, PL4, SE1, Discrete Mathematics, Calculus, Linear Algebra.

Theory of Computation -- 4

Topic Summary: Continuation of the study of formal models of computation, including finite automata, pushdown automata, Linear-bounded automata, and Turing machines (deterministic and nondeterministic). From the formal language perspective, regular, context-free, context-sensitive, and unrestricted grammars will be studied and shown to be equivalent to the corresponding machine models. Church's thesis will be discussed and the equivalence of various models of computation (e.g., Turing machines, random access machines, lambda calculus, and recursive functions) is also included. These models provide a basis for the study of computability, including effectively enunerable and undecidable problems.

Suggested Laboratories: Optionally, students will design and implement simple Turing machines or automata using a simulator (e.g., Turing's world.

Prerequisites: Discrete Mathematics, AL5, AL7, PL7, PL8, SE5.

VLSI System Design -- 4

Topic Summary: This material is intended to provide students with an understanding of the design and implementation of VLSI logic devices. This included issues of casting digital system architecture into silicon devices and the associated design alternatives. Examples of practical design methodologies (e.g., CAD tools) and the attributes of available technologies are compared.

Following an introduction to VLSI design, this topic covers integrated circuit technologies, design methodologies for VLSI, semicustom and custom MOS circuit design, and support technologies. Further subtopics include high-speed VLSI and application specific systems, systolic arrays, and wafer scale integration.

Suggested Laboratories: A current equipped laboratory for the design and simulation of VLSI logic devices should be available. This will enable students to develop a working knowledge of the tools and experiment with different organizations. A significant VLSI design should be the capstone experience for this topic. (Fabrication and testing of the design are not necessary.)

Prerequisites: All of the AR common requirements are a prerequisite for this topic. It is also recommended that the OS and SE common requirements be covered before studying this topic.

Last update:

Send comments to: webmaster@cs.wwc.edu

Laboratory Exercises for Computer Science

1. System
2. Architecture
3. System Administration
4. Compiler Construction
5. Database
6. Functional Programming
7. Logic Programming
8. Concurrent Programmming
9. Preface
10. CS 1
11. Automata
12. Miscellaneous
13. Imperative Programming
14. Cog Sci

Last update:

Send comments to: webmaster@cs.wwc.edu

Compiler Construction

1. Simple recursive descent compiler

Database Management System (DBMS)

Develop a simple DBMS for the relational data model.

Develop and Entity-relationship diagram
: Choose an organization you are most familiar with: college or university, public library, hospital, fast-food restaurant, department store, sports team. Determine the entities of interest and the relationships that exist between these entities. Draw the E-R diagram for the organization. Construct a tabular representation of the entities and relationships.

Database
: create files for the tabular representation.

Data Definition Compiler
: do not implement.

Data Dictionary
: create a special file containing the discription of the structure of the data in the database.

Query Processor
: design and implement a query processor for the relational algebra. The query processor, given a query and the data dictionary, translates the query into a series of requests to the data manager and returns the result of the query.

Data Manager
: use the unix file system facilities
  - File Manager
  - Disk Manager
  - Data Files

Telecommunication System
: not to be implemented.

Relational algebra
: Implement the operations of:

Exercises

The solution of the first exercise is used in the following exercises

1. Define a file format for a the relational database model.
2. Write a sort routine that can be used to sort on any column.
3. Write a file update routine. Assume that you have two files, a master file and a transaction file. A third file is to be produced which is the result of updating the master file from information contained in the transaction file. The update is based on using matching keys is designated columns.
4. Relational Algebra: Using the file format developed in the first exercise,
    a. Implement the operations of union, intersection and difference

b. Implement the operations of product, selection, and projection
c. Implement the natural join

Last update: a.aaby

Send comments to: webmaster@cs.wwc.edu

Functional Programming

The Lambda calculus

This laboratory is an introduction to the theory of functional programming. It may be used as a paper and pencil exercise or in conjunction with the software (provided in Pascal and Prolog). The Pascal program evaluates lambda expressions while the Prolog version provides for evaluation of lambda expressions, transformation of lambda expressions to SKI combinators, and evaluation of SKI expressions. % Note that the Prolog code uses the following syntax for lambda expressions. The lambda calculus is a formalization of the notion computability with fuctions. Its syntax is:

```
Abstract Syntax:

                L in Lambda Expressions
                x in Variables
                c in Constants

                L ::= c | x | (L_1 L_2) | (lambdax.L_3)
where (L_1 L_2) is function application, and (lambda x.L_3) is a lambda
abstraction which defines a function with argument x and body L_3.
```

Lambda expressions are reduced (simplified) using the *Beta*-rule:

```
        ((lambda x.B) y) => B[x:e]
```

which says that the occurrences of x in B can be replaced with e. All bound identifiers in B are renamed so as not to clash with the free identifiers in e. The program `lambda.p` is a program (the code is written in Pascal) which reduces lambda expressions to their normal form and `lambda.pro` is essentially the same program written in Prolog. Lambda expressions are transformed into SKI expressions with the following rules:

```
        C[CV]           ->      CV
        C[(E_1 E_2)]    ->      (C[ E_1] C[ E_2 ])
        C[lambda x.E]   ->      A[(x,C[E] )]
        A[(x,x)]                ->      I
        A[(x,c)]                ->      (K c)
        A[(x,(E_1 E_2))]        ->      ((S A[(x,E_1)]) A[(x,E_2)] )
```

Where CV is a constant or a variable. The reduction rules for the SKI calculus are as follows:

- S f g x -> f x (g x)
- K c x -> c

- I x -> x
- Y e -> e (Y e)
- (A B) -> A B
- (A B C)-> A B C

The reduction rules require that reductions be performed left to right. If no S, K, I, or Y reduction applies, then brackets are removed and reductions continue. The program {\bf ski.pro } is a Prolog program which provides a compiler from the lambda calculus to combinatorial logic, a combinatorial logic reduction machine, and makes provision to compile and then execute programs written in the lambda calculus.

input{FunctionalProgramming/Scheme} input{FunctionalProgramming/Haskell}

Submission of Programs

Programs should be submitted by doing something along the following lines (example uses Scheme).

```
% script
Script started, file is typescript
% cat {\it source code files}
...  {\it the source code listing}
% haskell
T 3.1 (14) SPARC/UNIX  Copyright (C) 1989 Yale University
Haskell Y1.2 (Oct 91) Command Interface. Type :? for help
Main>
... {\it show program execution}
Main> :quit

Do you really want to quit Haskell [no] y
% exit  {\it to exit script}
% Script done, file is typescript
% lpr -p typescript  {\it to print script file}
```

Arithmetic and lists

This laboratory provides practice in producing user defined functions for arithemetic and list processing.

Functionals

Functional programming languages provide ``meta" or ``higher-order" capabilities by permitting functions to be passed as parameters and returned as results. Functional programming languages provide a number of useful built in functionals. Here is a list of functionals:

filter
: applied to a predicate and a list, returns a list containing only those elements that satisfy the predicate. Example

```
filter (>5) [3,7,2,8,1,17] has value [7,8,17]
```

partition
: applied to a predicate and a list returns a pair of lists, those elements of the list that do and do not satisfy the predicate, respectively.

foldl
: folds up a list, using a given binary operator and a given start value, in a left associative way. Example:

```
foldl op r [a,b,c] = (((r op a) op b) op c)
```

But note that in order to run in constant space, foldl forces `op' to evaluate its first parameter.

foldl1
: folds left over non-empty lists.

foldr
: folds up a list, using a given binary operator and a given start value, in a right associative way. Example:

```
foldr op r [a,b,c] = a op (b op (c op r))
```

foldr1
: folds right over non-empty lists.

scanl op r
: applies `foldl op r' to every initial segment of a list. For example `scanl (+) 0 x' computes running sums.

scanl1
: is similar to scanl but without the starting element.

scanr
: is similar to scanl but from the right.

scanr1
: is similar to scanr but without the starting element.

map
: applied to a function and a list returns a copy of the list in which the given function has been applied to every element.

map2
: is similar to `map', but takes a function of two arguments, and maps it along two argument lists. We could also define `map3', `map4' etc., but they are much less often needed.

takewhile
: applied to a predicate and a list, takes elements from the front of the list while the predicate is satisfied. Example:

```
takewhile digit "123gone" has value "123"
```

dropwhile
: applied to a predicate and a list, removes elements from the front of the list while the predicate is satisfied. Example:

```
dropwhile digit "123gone" has value "gone"
```

See also `takewhile'.

until
: applied to a predicate, a function and a value, returns the result of applying the function to the value the smallest number of times necessary to satisfy the predicate. Example

```
until (>1000) (2*) 1 = 1024
```

iterate
: - iterate f x returns the infinite list [x, f x, f(f x), ...] Example, iterate (2*) 1 yields a list of the powers of 2.

Here are two examples to illustrate the usefulness of functionals. The first computes the sum of the elements of a list i.e., \sum_{i=1}^n x_i and the second the sum of the squares of the elements of a list i.e., \sum_{i=1}^n x_i^2

```
sumx = foldr (+) 0
sumsqrs x = foldr (+) 0 (map (^2) x)
```

Lazy evaluation and infinite data structures

This laboratory provides practice in lazy evaluation and infinite data structures.

1. lambda calculus
2. arithmetic and lists
3. functionals
4. lazy evaluation

Last update:

Send comments to: webmaster@cs.wwc.edu

Logic Programming

Labs

1. Database
2. theory
3. Arithmetic and Lists
4. Tailrecursion
5. Pattern matching
6. basic
7. incomplete
8. extra_logical
9. 2ndorder
10. meta_logical
11. dcg
12. kbs
13. oop
14. ski
15. parser

Last update:

Send comments to: webmaster@cs.wwc.edu

Hardware

- RAID

RAID

Older RAID levels

Level	Description	Comments/Advantages	Disadvantages
RAID 0	files *striped* across multiple drives	high read & write performance	no redundancy
RAID 1	files are *mirrored* on second drive	data redundancy faster read performance	double disk space slower write performance
RAID 2	RAID 1 with error-correction code (ECC)	not generally used since SCSI dirives have ECC built in	
RAID 3	files are striped at the byte level across multiple drives; parity value stored on a dedicated drive	hardware based data redundancy faster read and write performance	extra disk required I/O can be a bottleneck expensive
RAID 4	RAID 3 except fiiles are striped at block level	less expensive than RAID 3 data redundancy faster read and write performance	I/O can be a bottleneck
RAID 5	RAID 4 except parity information is distributed across all drives	data redundancy faster reads	writes can be slow

Some vendors provide combinations of RAID levels.

New RAID levels

Term	Description
FRDS (failure-resistant disk system)	system protects against data loss due to failure of a singe part of the system
FRDS plus	FRDS + *hot swapping* & the ability to recover from cache and power failures
FTDS (failure-tolerant disk system)	FRDS + reasonable protection against other failures
FTDS plus	FTDS + protection against bus failures
DTDS (disaster-tolerant disk system)	two or more zones with cooperation to prevent data loss in case of complete failure of one machine or array

DTDS plus	DTDS + recovery in case of al. manner of disasters -- flood, fire, ...

Logic Programming

- Prolog Tutorial
- Program schemata
- Software Engineering
- Examples

Prolog Tutorial

J. A. Robinson: A program is a theory (in some logic) and computation is deduction from the theory.
N. Wirth: Program = data structure + algorithm
R. Kowalski: Algorithm = logic + control

Introduction

Prolog, which stands for PROgramming in LOGic, is the most widely available language in the logic programming paradigm. Logic and therefore Prolog is based the mathematical notions of relations and logical inference. Prolog is a declarative language meaning that rather than describing how to compute a solution, a program consists of a data base of facts and logical relationships (rules) which describe the relationships which hold for the given application. Rather then running a program to obtain a solution, the user asks a question. When asked a question, the run time system searches through the data base of facts and rules to determine (by logical deduction) the answer.

Among the features of Prolog are `logical variables' meaning that they behave like mathematical variables, a powerful pattern-matching facility (unification), a backtracking strategy to search for proofs, uniform data structures, and input and output are interchangeable.

Often there will be more than one way to deduce the answer or there will be more than one solution, in such cases the run time system may be asked find other solutions. backtracking to generate alternative solutions. Prolog is a weakly typed language with dynamic type checking and static scope rules.

Prolog is used in artificial intelligence applications such as natural language interfaces, automated reasoning systems and expert systems. Expert systems usually consist of a data base of facts and rules and an inference engine, the run time system of Prolog provides much of the services of an inference engine.

The Structure of Prolog Programs

- A Prolog program consists of a database of facts and rules, and queries (questions).
  - Fact: ... .
  - Rule: ... :- ... .
  - Query: ?- ... .
  - Variables: must begin with an upper case letter.
  - Constants: numbers, begin with lowercase letter, or enclosed in single quotes.
- Inductive definitions: base and inductive cases
  - Towers of Hanoi: move N disks from pin a to pin b using pin c.

```
hanoi(N)                          :- hanoi(N, a, b, c).
hanoi(0,_,_,_).
hanoi(N,FromPin,ToPin,UsingPin) :- M is N-1,
                                   hanoi(M,FromPin,UsingPin,ToPin),
                                   move(FromPin,ToPin),
                                   hanoi(M,UsingPin,ToPin,FromPin).
move(From,To)                     :- write([move, disk from, pin, From, to, pin, ToPin]),
                                   nl.
```

  - Lists: append, member

```
list([]).
list([X|L])                 :- [list(L).
Abbrev:                       [X1|[...[Xn|[]...] = [X1,...Xn]
append([],L,L).
append([X|L1],L2,[X|L12]) :- append(L1,L2,L12).
member(X,L)                 :- concat(_,[X|_],L).
```

  - Ancestor

```
ancestor(A,D) :- parent(A,B).
ancestor(A,D) :- parent(A,C),ancestor(C,D).
```

but not

```
ancestor(A,D) :- ancestor(A,P), parent(P,D).
```

since infinite recursion may result.

- Depth-first search: Maze/Graph traversal
  A database of arcs (we will assume they are directed arcs) of the form:

```
a(node_i,node_j).
```

  Rules for searching the graph:

```
go(From,To,Trail).
go(From,To,Trail)  :- a(From,In), not visited(In,Trail), go(In,To,[In|Trail]).
```

visited(A,T) :- member(A,T).

- I/O: terms, characters, files, lexical analyzer/scanner
  - read(T), write(T), nl.
  - get0(N), put(N): ascii value of character
  - name(Name,Ascii_list).
  - see(F), seeing(F), seen, tell(F), telling(F), told.
- Natural language processing: Context-free grammars may be represented as Prolog rules. For example, the rule

  sentence ::= noun_clause verb_clause

  can be implemented in Prolog as

  sentence(S) :- append(NC,VC,S), noun_clause(NC), verb_clause(VC).

  or in DCG as:

  sentence -> noun_clause, verb_clause.

  ?- sentence(S,[]).

  Note that two arguments appear in the query. Both are lists and the first is the sentence to be parsed, the second the remaining elements of the list which in this case is empty.

A Prolog program consists of a data base of facts and rules. There is no structure imposed on a Prolog program, there is no main procedure, and there is no nesting of definitions. All facts and rules are global in scope and the scope of a variable is the fact or rule in which it appears. The readability of a Prolog program is left up to the programmer.

A Prolog program is executed by asking a question. The question is called a query. Facts, rules, and queries are called *clauses*.

Syntax

Facts

A **fact** is just what it appears to be --- a fact. A fact in everyday language is often a proposition like ``It is sunny.'' or ``It is summer.'' In Prolog such facts could be represented as follows:

```
'It is sunny'.
'It is summer'.
```

Queries

A **query** in Prolog is the action of asking the program about information contained within its data base. Thus, queries usually occur in the interactive mode. After a program is loaded, you will receive the query prompt,

```
?-
```

at which time you can ask the run time system about information in the data base. Using the simple data base above, you can ask the program a question such as

```
?- 'It is sunny'.
```

and it will respond with the answer

```
Yes
?-
```

A `yes` means that the information in the data base is consistent with the subject of the query. Another way to express this is that the program is capable of proving the query true with the available information in the data base. If a fact is not deducible from the data base the system replys with a no, which indicates that based on the information available (the closed world assumption) the fact is not deducible.

If the data base does not contain sufficient information to answer a query, then it answers the query with a no.

```
?- 'It is cold'.
no
?-
```

Rules

Rules extend the capabilities of a logic program. They are what give Prolog the ability to pursue its decision-making process. The following program contains two rules for temperature. The first rule is read as follows: ``It is hot if it is summer and it is sunny." The second rule is read as follows: ``It is cold if it is winter and it is snowing."

```
'It is sunny'.
'It is summer'.
'It is hot' :- 'It is summer', 'It is sunny'.
'It is cold' :- 'It is winter', 'It is snowing'.
```

The query,

```
?- 'It is hot'.
Yes
?-
```

is answered in the affirmative since both 'It is summer' and 'It is sunny' are in the data base while a query ``?- 'It is cold.' " will produce a negative response.

The previous program is an example of propositional logic. Facts and rules may be parameterized to produce programs in predicate logic. The parameters may be variables, atoms, numbers, or terms. Parameterization permits the definition of more complex relationships. The following program contains a number of predicates that describe a family's genelogical relationships.

```
female(amy).
female(johnette).

male(anthony).
male(bruce).
male(ogden).

parentof(amy,johnette).
parentof(amy,anthony).
parentof(amy,bruce).
parentof(ogden,johnette).
parentof(ogden,anthony).
parentof(ogden,bruce).
```

The above program contains the three simple predicates: `female`; `male`; and `parentof`. They are parameterized with what are called `atoms.' There are other family relationships which could also be written as facts, but this is a tedious process. Assuming traditional marriage and child-bearing practices, we could write a few rules which would relieve the tedium of identifying and listing all the possible family relations. For example, say you wanted to know if `johnette` had any siblings, the first question you must ask is ``what does it mean to be a sibling?" To be someone's sibling you must have the same parent.

This last sentence can be written in Prolog as

```
siblingof(X,Y) :-
            parentof(Z,X),
            parentof(Z,Y).
```

A translation of the above Prolog rule into English would be ``X is the sibling of Y provided that Z is a parent of X, and Z is a parent of Y." `X, Y,` and `Z` are variables. This rule however, also defines a child to be its own sibling. To correct this we must add that `X` and `Y` are not the same. The corrected version is:

```
siblingof(X,Y) :-
            parentof(Z,X),
            parentof(Z,Y),
            X  Y.
```

The relation `brotherof` is similar but adds the condition that `X` must be a male.

```
brotherof(X,Y) :-
            parentof(Z,X),
            male(X),
            parentof(Z,Y),
            X  Y.
```

From these examples we see how to construct facts, rules and queries and that strings are enclosed in single quotes, variables begin with a capital letter, constants are either enclosed in single quotes or begin with a small letter.

Types

Prolog provides for numbers, atoms, lists, tuples, and patterns. The types of objects that can be passed as arguments are defined in this section.

Simple Types

Simple types are implementation dependent in Prolog however, most implementations provide the simple types summarized in the following table.

TYPE	VALUES
boolean	true, fail
integer	integers
real	floating point numbers
variable	variables
atom	character sequences

The boolean constants are not usually passed as parameters but are propositions. The constant `fail` is useful in forcing the generation of all solutions. Variables are character strings beginning with a capital letter. Atoms are either quoted character strings or unquoted strings beginning with a small letter.

Composite Types

In Prolog the distinction between programs and data are blurred. Facts and rules are used as data and data is often passed in the arguments to the predicates. Lists are the most common data structure in Prolog. They are much like the array in that they are a sequential list of elements, and much like the stack in that you can only access the list of elements sequentially, that is, from

one end only and not in random order. In addition to lists Prolog permits arbitrary patterns as data. The patterns can be used to represent tuples. Prolog does not provide an array type. But arrays may be represented as a list and multidimensional arrays as a list(s) of lists. An alternate representation is to represent an array as a set of facts in a the data base.

TYPE

REPRESENTATION	[*comma separated sequence of items*]	*sequence of items*
list	pattern	

A list is designated in Prolog by square brackets ([ ]+). An example of a list is

```
[dog,cat,mouse]
```

This says that the list contains the elements `dog`, {\tt cat, and `mouse`, in that order. Elements in a Prolog list are ordered, even though there are no indexes. Records or tuples are represented as patterns. Here is an example.

```
book(author(aaby,anthony),title(lab_manual),data(1991))
```

The elements of a tuple are accessed by pattern matching.

```
book(Title,Author,Publisher,Date).
author(LastName,FirstName,MI).
publisher(Company,City).

book(T,A,publisher(C,rome),Date)
```

Type Predicates

Since Prolog is a weakly typed language, it is important for the user to be able to determine the type of a parameter. The following built in predicates are used to determine the type of a parameter.

PREDICATE	CHECKS IF
var(V)	V is a variable
nonvar(NV)	NV is not a variable
atom(A)	A is an atom
integer(I)	I is an integer
real(R)	R is a floating point number
number(N)	N is an integer or real
atomic(A)	A is an atom or a number
functor(T,F,A)	T is a term with functor F and arity A
T =..L	T is a term, L is a list (see example below).
clause(H,T)	H :- T is a rule in the program

The last three are useful in program manipulation (metalogical or meta-programming) and require additional explanation. `clause(H,T)` is used to check the contents of the data base. `functor(T,F,A)` and `T=..L` are used to manipulate terms. The predicate, `functor` is used as follows.

```
functor(T,F,A)
```

`T` is a term, `F` is its functor, and `A` is its arity. For example,

```
?- functor(t(a,b,c),F,A).
F = t
A = 3
yes
```

`t` is the functor of the term `t(a,b,c)`, and 3 is the arity (number of arguments) of the term. The predicate `=..` (univ) is used to compose and decompose terms. For example:

```
?- t(a,b,c) =..L.
L = [t,a,b,c]
yes
?- T =..[t,a,b,c].
T = t(a,b,c)
yes
```

Expressions

Arithmetic expressions are evaluated with the built in predicate `is` which is used as an infix operator in the following form.

variable is *expression*

For example,

```
?- X is 3*4.
X = 12
yes
```

Arithmetic Operators

Prolog provides the standard arithmetic operations as summarized in the following table.

SYMBOL	OPERATION
+	addition
-	subtraction
*	multiplication
/	real division
//	integer division
mod	modulus
**	power

Boolean Predicates

Besides the usual boolean predicates, Prolog provides more general comparison operators which compare terms and predicates to test for unifiability and whether terms are identical.

SYMBOL	OPERATION	ACTION
A ?= B	unifiable	A and B are unifiable but does not unify A and B
A = B	unify	unifys A and B if possible
A \+= B	not unifiable	
A == B	identical	does not unify A and B

A \+== B	not identical		
A =:= B	equal (value)	evaluates A and B to	determine if equal
A =\+= B	not equal (value)		
A < B	less than (numeric)		
A =< B	less or equal (numeric)		
A > B	greater than (numeric)		
A >= B	greater or equal (numeric)		
A @< B	less than (terms)		
A @=< B	less or equal (terms)		
A @> B	greater than (terms)		
A @>= B	greater or equal (terms)		

For example, the following are all true.

```
3 @< 4
3 @< a
a @< abc6
abc6 @< t(c,d)
t(c,d) @< t(c,d,X)
```

Logic programming definition of natural number.

```
% natural_number(N) <- N is a natural number.

natural_number(0).
natural_number(s(N)) :- natural_number(N).
```

Prolog definition of natural number.

```
natural_number(N) :- integer(N), N >= 0.
```

Logic programming definition of inequalities

```
% less_than(M,N) <- M is less than M

less_than(0,s(M)) :- natural_number(M).
less_than(s(M),s(N)) :- less_than(M,N).

% less_than_or_equal(M,N) <- M is less than or equal to M

less_than_or_equal(0,N) :- natural_number(N).
less_than_or_equal(s(M),s(N)) :- less_than_or_equal(M,N).
```

Prolog definition of inequality.

```
M =< N.
```

Logic programming definition of addition/substraction

```
% plus(X,Y,Z) <- Z is X + Y

plus(0,N,N) :- natural_number(N).
plus(s(M),N,s(Z)) :- plus(M,N,Z).
```

Prolog definition of addition

```
plus(M,N,Sum) :- Sum is M+N.
```

This does not define substration. Logic programming definition of multiplication/division

```
% times(X,Y,Z) <- Z is X*Y

times(0,N,0) :- natural_number(N).
times(s(M),N,Z) :- times(M,N,W), plus(W,N,Z).
```

Prolog definition of multiplication.

```
times(M,N,Product) :- Product is M*N.
```

This does not define substration. Logic programming definition of Exponentiation

```
% exp(N,X,Z) <- Z is X**N

exp(s(M),0,0) :- natural_number(M).
exp(0,s(M),s(0)) :- natural_number(M).
exp(s(N),X,Z) :- exp(N,X,Y), times(X,Y,Z).
```

Prolog definition of exponentiation is implementation dependent.

Logical Operators

Predicates are functions which return a boolean value. Thus the logical operators are built in to the language. The comma on the right hand side of a rule is logical conjunction. The symbol `:-` is logical implication. In addition Prolog provides negation and disjunction operators. The logical operators are used in the definition of rules. Thus,

```
a :- b. % a if b
a :- b,c. % a if b and c.
a :- b;c. % a if b or c.
a :- \++ b. % a if b is not provable
a :- not b. % a if b fails
a :- b -> c;d. % a if (if b then c else d)
```

This table summarizes the logical operators.

SYMBOL	OPERATION
not	negation
\+	not provable
,	logical conjunction
;	logical disjunction
:-	logical implication
->	if-then-else

Unification and Pattern Matching

The arguments in a query are matched (or unified in Prolog terminology) to select the appropriate rule. Here is an example

which makes extensive use of pattern matching. The rules for computing the derivatives of polynomial expressions can be written as Prolog rules. A given polynomial expression is matched against the first argument of the rule and the corresponding derivative is returned.

```
% deriv(Polynomial, variable, derivative)
% dc/dx = 0
deriv(C,X,0) :- number(C).
% dx/dx} = 1
deriv(X,X,1).
% d(cv)/dx = c(dv/dx)
deriv(C*U,X,C*DU) :- number(C), deriv(U,X,DU).
% d(u v)/dx = u(dv/dx) + v(du/dx)
deriv(U*V,X,U*DV + V*DU) :- deriv(U,X,DU), deriv(V,X,DV).
% d(u ± v)/dx = du/dx ± dv/dx
deriv(U+V,X,DU+DV) :- deriv(U,X,DU), deriv(V,X,DV).
deriv(U-V,X,DU-DV) :- deriv(U,X,DU), deriv(V,X,DV).
% du^n/dx = nu^{n-1}(du/dx)
deriv(U^+N,X,N*U^+N1*DU) :- N1 is N-1, deriv(U,X,DU).
```

Prolog code is often bidirectional. In bidirectional code, the arguments may be use either for input or output. For example, this code may be used for both differentiation and integration with queries of the form:

```
?- deriv(Integral,X,Derivative).
```

where either *Integral* or *Derivative* may be instantiated to a formula.

Functions

Prolog does not provide for a function type therefore, functions must be defined as relations. That is, both the arguments to the function and the result of the function must be parameters to the relation. This means that composition of two functions cannot be constructed. As an example, here is the factorial function defined as relation in Prolog. Note that the definition requires two rules, one for the base case and one for the inductive case.

```
fac(0,1).
fac(N,F) :- N > 0, M is N - 1,
       fac(M,Fm), F is N * Fm.
```

The second rule states that if `N` > 0, `M = N - 1`, `Fm` is `(N-1)!`, and `F = N * Fm`, then `F` is `N!`. Notice how `` `is' `` is used. In this example it resembles an assignment operator however, it may not be used to reassign a variable to a new value. I the logical sense, the order of the clauses in the body of a rule are irrelevant however, the order may matter in a practical sense. `M` must not be a variable in the recursive call otherwise an infinite loop will result. Much of the clumsiness of this definition comes from the fact that `fac` is defined as a relation and thus it cannot be used in an expression. Relations are commonly defined using multiple rules and the order of the rules may determine the result. In this case the rule order is irrelevant since, for each value of `N` only one rule is applicable. Here are the Prolog equivalent of the definitions of the gcd function, Fibonacci function and ackerman's function.

```
gcd(A,B,GCD) :- A = B, GCD = A.
gcd(A,B,GCD) :- A < B, NB is B - A, gcd(A,NB,GCD).
gcd(A,B,GCD) :- A > B, NA is A - B, gcd(NA,B,GCD).

fib(0,1).
fib(1,1).
fib(N,F) :- N > 1, N1 is N - 1, N2 is N - 2,
    fib(N1,F1), fib(N2,F2), F is F1 + F2.
```

```
ack(0,N,A) :- A is N + 1.
ack(M1,0,A) :- M > 0, M is M - 1, ack(M,1,A).
ack(M1,N1,A) :- M1 > 0, N1 > 0, M is M - 1, N is N - 1,
   ack(M1,N,A1), ack(M,A1,A).
```

Notice that the definition of ackerman's function is clumsier than the corresponding functional definition since the functional composition is not available. Logic programming definition of the factorial function.

```
% factorial(N,F) <- F is N!

factorial(0,s(0)).
factorial(s(N),F) :- factorial(N,F1), times(s(N),F1,F).
```

Prolog definition of factorial function.

```
factorial(0,1).
factorial(N,F) :- N1 is N-1, factorial(N1,F1), F is N*F1.
```

Logic programming definition of the minimum.

```
% minimum(M,N,Min) <- Min is the minimum of {M, N}

minimum(M,N,M) :- less_than_or_equal(M,N).
minimum(M,N,N) :- less_than_or_equal(N,M).
```

Prolog programming definition of the minimum.

```
minimum(M,N,M) :- M =< N.
minimum(M,N,N) :- N =< M.
```

Logic programming definition of the modulus.

```
% mod(M,N,Mod) <- Mod is the remainder of the integer division of M by N.

mod(X,Y,Z) :- less_than(Z,Y), times(Y,Q,W), plus(W,Z,X).

% or
mod(X,Y,X) :- less_than(X,Y).
mod(X,Y,X) :- plus(X1,Y,X), mod(X1,Y,Z).
```

Logic programming definition of Ackermann's function.

```
ack(0,N,s(N)).
ack(s(M),0,Val) :- ack(M,s(0),Val).
ack(s(M),s(N),Val) :- ack(s(M),N,Val1), ack(M,Val1,Val).
```

Prolog definition of Ackermann's function.

```
ack(0,N,Val) :- Val is N + 1.
ack(M,0,Val) :- M > 0, M1 is M-1, ack(M1,1,Val).
ack(M,N,Val) :- M > 0, N > 0, M1 is M-1, N1 is N-1,
                ack(M,N1,Val1), ack(M1,Val1,Val).
```

Logic programming definition of the Euclidian algorithm.

```
gcd(X,0,X) :- X > 0.
```

```
gcd(X,Y,Gcd) :- mod(X,Y,Z), gcd(Y,Z,Gcd).
```

Logic programming definition of the Euclidian algorithm.

```
gcd(X,0,X) :- X > 0.
gcd(X,Y,Gcd) :- mod(X,Y,Z), gcd(Y,Z,Gcd).
```

Lists

Objective
Outline

- Lists
- Composition of Recursive Programs
- Iteration

Lists are the basic data structure used in logic (and functional) programming. Lists are a recursive data structure so recursion occurs naturally in the definitions of various list operations. When defining operations on recursive data structures, the definition most often naturally follows the recursive definition of the data structure. In the case of lists, the empty list is the base case. So operations on lists must consider the empty list as a case. The other cases involve a list which is composed of an element and a list.

Here is a recursive definition of the list data structure as found in Prolog.

```
List --> [ ]
List --> [Element|List]
```

Here are some examples of list representation, the first is the empty list.

```
Pair Syntax                Element Syntax
 [ ]             [ ]
 [a|[ ]]         [a]
 [a|b|[ ]]       [a,b]
 [a|X]           [a|X]
 [a|b|X]         [a,b|X]
```

Predicates on lists are often written using multiple rules. One rule for the empty list (the base case) and a second rule for non empty lists. For example, here is the definition of the predicate for the length of a list.

```
% length(List,Number) <- Number is lenght of List

length([],0).
length([H|T],N) :- length(T,M), N is M+1.
```

Element of a list.

```
% member(Element,List) <- Element is an element of the list List

member(X,[X|List).
member(X,[Element|List]) :- member(X,List).
```

Prefix of a list.

```
% prefix(Prefix,List) <- Prefix is a prefix of list List

prefix([],List).
```

```
prefix([X|Prefix],[X|List]) :- prefix(Prefix,List).
```

Suffix of a list.

```
% suffix(Suffix,List) <- Suffix is a suffix of list List

suffix(Suffix,Suffix).
prefix(Suffix,[X|List]) :- suffix(Suffix,List).
```

Append (concatenate) two lists.

```
% append(List1,List2,List1List2) <-
%   List1List2 is the result of concatenating List1 and List2.

append([],List,List).
append([Element|List1],List2,[Element|List1List2]) :-
append(List1,List2,List1List2).
```

Compare this code with the code for plus. sublist -- define using

- Suffix of a prefix
- Prefix of a suffix
- Recursive definition of sublist using prefix
- Suffix of a prefix using append
- Prefix of a suffix using append

member, prefix and suffix -- defined using append reverse, delete, select, sort, permutation, ordered, insert, quicksort.

Iteration

Iterative version of Length

```
% length(List,Number) <- Number is lenght of List
% Iterative version.

length(List,LenghtofList) :- length(List,0,LengthofList).

% length(SufixList,LengthofPrefix,LengthofList) <-
%         LengthofList is LengthofPrefix + length of SufixList

length([],LenghtofPrefix,LengthofPrefix).
length([Element|List],LengthofPrefix,LengthofList) :-
PrefixPlus1 is LengthofPrefix + 1,
length(List,PrefixPlus1,LengthofList).
```

Iterative version of Reverse

```
% reverse(List,ReversedList) <- ReversedList is List reversed.
% Iterative version.

reverse(List,RList) :- reverse(List,[],RList).

% length(SufixList,LengthofPrefix,LengthofList) <-
%         LengthofList is LengthofPrefix + length of SufixList

reverse([],RL,RL).
reverse([Element|List],RevPrefix,RL) :-
```

```
reverse(List,[Element|RevPrefix],RL).
```

Here are some simple examples of common list operations defined by pattern matching. The first sums the elements of a list and the second forms the product of the elements of a list.

```
sum([ ],0).
sum([X|L],Sum) :- sum(L,SL), Sum is X + SL.

product([ ],1).
product([X|L],Prod) :- product(L,PL), Prod is X * PL.
```

Another example common list operation is that of appending or the concatenation of two lists to form a third list. Append may be described as the relation between three lists, L1, L2, L3, where L1 = $[x_1,...,x_m]$, L2 = $[y_1,...,y_n]$ and L3 = $[x_1,...,x_m,y_1,...,y_n]$. In Prolog, an inductive style definition is required.

```
append([ ],L,L).
append([X1|L1],L2, [X1|L3]) :- append(L1,L2,L3).
```

The first rule is the base case. The second rule is the inductive case. In effect the second rule says that

if L1 = $[x_2,...,x_m]$,
L2 = $[y_1,...,y_n]$ and
L3 = $[x_2,...,x_m,y_1,...,y_n]$,
then $[x_1,x_2,...,x_m,y_1,...,y_n]$, is the result of
appending $[x_1,x_2,...,x_m]$ and L2.

The append relation is quite flexible. It can be used to determine if an object is an element of a list, if a list is a prefix of a list and if a list is a suffix of a list.

```
member(X,L) :- append(_,[X|_],L).
prefix(Pre,L) :- append(Prefix,_,L).
suffix(L,Suf) :- append(_,Suf,L).
```

The underscore (_+) in the definitions denotes an anonymous variable (or don`t care) whose value in immaterial to the definition. The member relation can be used to derive other useful relations.

```
vowel(X) :- member(X,[a,e,i,o,u]).
digit(D) :- member(D,['0','1','2','3','4','5','6','7','8','9']).
```

A predicate defining a list and its reversal can be defined using pattern matching and the append relation as follows.

```
reverse([ ],[ ]).
reverse([X|L],Rev) :- reverse(L,RL), append(RL,[X],Rev).
```

Here is a more efficient (iterative/tail recursive) version.

```
reverse([ ],[ ]).
reverse(L,RL) :- reverse(L,[ ],RL).

reverse([ ],RL,RL).
reverse([X|L],PRL,RL) :- reverse(L,[X|PRL],RL).
```

To conclude this section, here is a definition of insertion sort.

```
isort([ ],[ ]).
isort([X|UnSorted],AllSorted) :- isort(UnSorted,Sorted),
                                 insert(X,Sorted,AllSorted).

insert(X,[ ],[X]).
insert(X,[Y|L],[X,Y|L]) :- X =< Y.
insert(X,[Y|L],[Y|IL]) :-  X > Y, insert(X,L,IL).
```

Iteration

Recursion is the only iterative method available in Prolog. However, tail recursion can often be implemented as iteration. The following definition of the factorial function is an `iterative' definition because it is `tail recursive.' It corresponds to an implementation using a while-loop in an imperative programming language.

```
fac(0,1).
fac(N,F) :- N > 0, fac(N,1,F).

fac(1,F,F).
fac(N,PP,F) :- N > 1, NPp is N*PP, M is N-1,
       fac(M,NPp,F).
```

Note that the second argument functions as an *accumulator*. The accumulator is used to store the partial product much as might be done is a procedural language. For example, in Pascal an iterative factorial function might be written as follows.

```
function fac(N:integer) : integer;
var i : integer;
begin
   if N >= 0 then begin
       fac := 1
      for I := 1 to N do
           fac := fac * I
   end
end;
```

In the Pascal solution `fac` acts as an accumulator to store the partial product. The Prolog solution also illustrates the fact that Prolog permits different relations to be defined by the same name provided the number of arguments is different. In this example the relations are fac/2 and fac/3 where fac is the ``functor'' and the number refers to the arity of the predicate. As an additional example of the use of accumulators, here is an iterative (tail recursive version) of the Fibonacci function.

```
fib(0,1).
fib(1,1).
fib(N,F) :- N > 1, fib(N,1,1,F)

fib(2,F1,F2,F) :- F is F1 + F2.
fib(N,F1,F2,F) :- N > 2, N1 is N - 1, NF1 is F1 + F2,
    fib(N1,NF1,F1,F).
```

Iterators, Generators and Backtracking

The following fact and rule can be used to generate the natural numbers. % Natural Numbers

```
nat(0).
nat(N) :- nat(M), N is M + 1.
```

The successive numbers are generated by backtracking. For example, when the following query is executed successive natural

numbers are printed.

```
?- nat(N), write(N), nl, fail.
```

The first natural number is generated and printed, then `fail` forces backtracking to occur and the second rule is used to generate the successive natural numbers. The following code generates successive prefixes of an infinite list beginning with N.

```
natlist(N,[N]).
natlist(N,[N|L]) :- N1 is N+1, natlist(N1,L).
```

As a final example, here is the code for generating successive prefixes of the list of prime numbers.

```
primes(PL) :- natlist(2,L2), sieve(L2,PL).

sieve([ ],[ ]).
sieve([P|L],[P|IDL]) :- sieveP(P,L,PL), sieve(PL,IDL).

sieveP(P,[ ],[ ]).
sieveP(P,[N|L],[N|IDL]) :- N mod P  >  0, sieveP(P,L,IDL).
sieveP(P,[N|L],   IDL)  :- N mod P =:= 0, sieveP(P,L,IDL).
```

Occasionally, backtracking and multiple answers are annoying. Prolog provides the cut symbol (!) to control backtracking. The following code defines a predicate where the third argument is the maximum of the first two.

```
max(A,B,M) :- A <  B, M = B.
max(A,B,M) :- A >= B, M = A.
```

The code may be simplified by dropping the conditions on the second rule.

```
max(A,B,B) :- A <  B.
max(A,B,A).
```

However, in the presence of backtracking, incorrect answers can result as is shown here.

```
?- max(3,4,M).

M = 4;

M = 3
```

To prevent backtracking to the second rule the cut symbol is inserted into the first rule.

```
max(A,B,B) :- A < B.!.
max(A,B,A).
```

Now the erroneous answer will not be generated. A word of caution: cuts are similar to gotos in that they tend to increase the complexity of the code rather than to simplify it. In general the use of cuts should be avoided.

Tuples (or Records)

We illustrate the data type of tuples with the code for the abstract data type of a binary search tree. The binary search tree is represented as either `nil` for the empty tree or as the tuple `btree(Item,L_Tree,R_Tree)`. Here is the Prolog code for the creation of an empty tree, insertion of an element into the tree, and an in-order traversal of the tree.

```
create_tree(niltree).
```

```
inserted_in_is(Item,niltree, btree(Item,niltree,niltree)).

inserted_in_is(Item,btree(ItemI,L_T,R_T),Result_Tree) :-
    Item @< ItemI,
    inserted_in_is(Item,L_Tree,Result_Tree).

inserted_in_is(Item,btree(ItemI,L_T,R_T),Result_Tree) :-
    Item @> ItemI,
    inserted_in_is(Item,R_Tree,Result_Tree).

inorder(niltree,[ ]).
inorder(btree(Item,L_T,R_T),Inorder) :-
          inorder(L_T,Left),
          inorder(R_T,Right),
          append(Left,[Item|Right],Inorder).
```

The membership relation is a trivial modification of the insert relation. Since Prolog access to the elements of a tuple are by pattern matching, a variety of patterns can be employed to represent the tree. Here are some alternatives.

```
[Item,LeftTree,RightTree]
Item/LeftTree/RightTree
(Item,LeftTree,RightTree)
```

Extra-Logical Predicates

Objective
Outline

- Input/Output
- Assert/Retract
- System Access

The class of predicates in Prolog that lie outside the logic programming model are called *extra-logical* predicates. These predicates achieve a side effect in the course of being satisfied as a logical goal. There are three types of extra-logical predicates, predicates for handling I/O, predicates for manipulating the program, and predicates for accessing the underlying operating system.

Input/Output

Most Prolog implementations provide the predicates `read` and `write`. Both take one argument, `read` unifies its argument with the next term (terminated with a period) on the standard input and `write` prints its argument to the standard output. As an illustration of input and output as well as a more extended example, here is the code for a checkbook balancing program. The section beginning with the comment ``Prompts'' handles the I/0.

```
% Check Book Balancing Program.
checkbook :- initialbalance(Balance),
   newbalance(Balance).

% Recursively compute new balances
newbalance(OldBalance) :- transaction(Transaction),
   action(OldBalance,Transaction).

% If transaction amount is 0 then finished.
action(OldBalance,Transaction) :- Transaction = 0,
   finalbalance(OldBalance). %
%
```

```
% If transaction amount is not 0 then compute new balance.
action(OldBalance,Transaction) :- Transaction \+= 0,
   NewBalance is OldBalance + Transaction,
   newbalance(NewBalance).

%

% Prompts
initialbalance(Balance) :- write('Enter initial balance: \'),
   read(Balance).

transaction(Transaction) :-
   write('Enter Transaction, '),
    write('- for withdrawal, 0 to terminate): '),
   read(Transaction).
finalbalance(Balance) :- write('Your final balance is: \'),
   write(Balance), nl.
```

Files

see(File)
: Current input file is now File.

seeing(File)
: File is unified with the name of the current input file.

seen
: Closes the current input file.

tell(File)
: Current output file is now File.

telling(File)
: File is unified with the name of the current output file.

told
: Closes the current output file.

Term I/O

read(Term)
: Reads next full-stop (period) delimited term from the current input stream, if eof then returns the atom 'end_of_file'.

write(Term)
: Writes a term to the current output stream.

print(Term)
: Writes a term to the current output stream. Uses a user defined predicate portray/1 to write the term, otherwise uses write.

writeq(Term)
: Writes a term to the current output stream in a form aceptable as input to read.

Character I/O

get(N)
: N is the ASCII code of the next non-blank printable character on the current input stream. If end of file, then a -1 is returned.

put(N)
: Puts the character corresponding to ASCII code N on the current output stream.

nl
: Causes the next output to be on a new line.

tab(N)
: N spaces are output to the current output stream.

Program Access

consult(SourceFile)
: Loads SourceFile into the interpreter but, if a predicate is defined accross two or more files, consulting them will result in only the clauses in the file last consulted being used.

reconsult(File)
: available in some systems.

Other

name(Atom,ASCII_List)
: the conversion routine between lists of ASCII codes and atoms.

display, prompt

```
% Read a sentence and return a list of words.

read_in([W|Ws]) :- get0(C), read_word(C,W,C1), rest_sent(W,C1,Ws).

% Given a word and the next character, read in the rest of the sentence

rest_sent(W,_,[]) :- lastword(W).
rest_sent(W,C,[W1|Ws]) :- read_word(C,W1,C1), rest_sent(W1,C1,Ws).

read_word(C,W,C1) :- single_character(C),!,name(W,[C]), get0(C1).
read_word(C,W,C2) :- in_word(C,NewC), get0(C1),
                     rest_word(C1,Cs,C2), name(W,[NewC|Cs]).
read_word(C,W,C2) :- get0(C1), read_word(C1,W,C2).

rest_word(C,[NewC|Cs],C2) :- in_word(C,NewC), !, get0(C1),
rest_word(C1,Cs,C2).
rest_word(C,[],C).

% These are single character words.

single_character(33).      % !
single_character(44).      % ,
single_character(46).      % .
single_character(58).      % :
single_character(59).      % ;
single_character(63).      % ?

% These characters can appear within a word.

in_word(C,C) :- C > 96, C < 123.               % a,b,...,z
in_word(C,L) :- C > 64, C < 91, L is C + 32.   % A,B,...,Z
in_word(C,C) :- C > 47, C < 58.                % 0,1,...,9
in_word(39,39).                                % '
in_word(45,45).                                % -

% These words terminate a sentence.

lastword('.').
lastword('!').
lastword('?').
```

Program Access and Manipulation

clause(Head,Body)

assert(Clause)
: adds clause to the end of the database

asserta(Clause)

retract(Clause_Head)

consult(File_Name)

System Access

system(Command)
: Execute Command in the operating system

Style and Layout

Objective
Outline
- Style and Layout
- Debugging

Some conventions for comments.

- Long comments should precede the code they refer to while short comments should be interspersed with the code itself.
- Program comments should describe what the program does, how it is used (goal predicate and expected results), limitations, system dependent features, performance, and examples of using the program.
- Predicate comments explain the purpose of the predicate, the meaning and relationship among the arguments, and any restrictions as to argument type.
- Clause comments add to the description of the case the particular clause deals with and is usefull for documenting cuts.

Some conventions for program layout

- Group clauses belonging to a relation or ADT together.
- Clauses should be short. Their body should contain no more than a few goals.
- Make use of indentation to improve the readability of the body of a clause.
- Mnemonic names for relations and variables should be used. Names should indicate the meaning of relations and the role of data objects.
- Clearly separate the clauses defining different relations.
- The cut operator should be used with care. The use of `red' cuts should be limited to clearly defined mutually exclusive alternatives.

Illustration

```
merge( List1, List2, List3 ) :-
 ( List1 = [], !, List3 = List2 );
 ( List2 = [], !, List3 = List1 );
 ( List1 = [X|L1], List2 = [Y|L2 ),
((X < Y, ! Z = X, merge( L1, List2, L3 ) );
( Z = Y, merge( List1, L2, L3 ) )),
  List3 = [Z|L3].
```

A better version

```
merge( [], List2, List2 ).
merge( List1, [], List1 ).

merge( [X|List1], [Y|List2], [X|List3] ) :-
X < Y, !,  merge( List1, List2, List3 ).  \% Red Cut
merge( List1, [Y|List2], [Y|List3] ) :-
 merge( List1, List2, List3 ).
```

Debugging

trace/notrace, spy/nospy, programmer inserted debugging aids -- write predicates and p :- write, fail.

Negation and Cuts

Objective
Outline

- Negation as failure
- Green Cuts
- Red Cuts

Negation

Cuts

Green cuts: Determinism

Selection among mutually exclusive clauses.

Tail Recursion Optimization

Prevention of backtracking when only one solution exists.

```
A :- B1,...,Bn,Bn1.
A :- B1,...,Bn,!,Bn1. % prevents backtracking
```

Red cuts: omitting explicit conditions

Definite Clause Grammars

Objective:
Outline

- The parsing problem: Context-free grammars; Construct a parse tree for a sentence given the context-free grammar.
- Representing the Parsing Problem in Prolog
- The Grammar Rule Notation] (Definite Clause Grammars -- DCG)
- Adding Extra Arguments
- Adding Extra Tests

Prolog originated from attempts to use logic to express grammar rules and formalize the parsing process. Prolog has special syntax rules which are called *definite clause grammars* (DCG). DCGs are a generalization of context free grammars.

Context Free Grammars

A context free grammar is a set of rules of the form:

```
    ->
```

where `nonterminal` is a nonterminal and `body` is a sequence of one or more items. Each item is either a nonterminal symbol or a sequence of terminal symbols. The meaning of the rule is that the `body` is a possible form for an object of type `nonterminal`.

```
S --> a b
S --> a S b
```

DCG

Nonterminals are written as Prolog atoms, the items in the body are separated with commas and sequences of terminal symbols are written as lists of atoms. For each nonterminal symbol, S, a grammar defines a language which is obtained by repeated nondeterministic application of the grammar rules, starting from S.

```
s --> [a],[b].
s --> [a],s,[b].
```

As an illustration of how DCG are used, the string [a,a,b,b] is given to the grammar to be parsed.

```
?- s([a,a,b,b],[]).
   yes
```

Here is a natural language example.

```
% DCGrammar

sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun.
noun_phrase --> noun.

verb_phrase --> verb.
verb_phrase --> verb, noun_phrase.

% Vocabulary

determiner --> [the].
determiner --> [a].

noun --> [cat].
noun --> [cats].
noun --> [mouse].
noun --> [mice].

verb --> [scare].
verb --> [scares].
verb --> [hate].
verb --> [hates].
```

Context free grammars cannot define the required agreement in number between the noun phrase and the verb phrase. That information is context dependent (sensitive). However, DCG are more general Number agreement

```
% DCGrammar - with number agreement between noun phrase and verb phrase

sentence --> noun_phrase(Number), verb_phrase(Number).

noun_phrase(Number) --> determiner(Number), noun(Number).
noun_phrase(Number) --> noun(Number).

verb_phrase(Number) --> verb(Number).
verb_phrase(Number) --> verb(Number), noun_phrase(Number1).

% Vocabulary

determiner(Number) --> [the].
determiner(singular) --> [a].

noun(singular) --> [cat].
```

```
noun(plural) --> [cats].
noun(singular) --> [mouse].
noun(plural) --> [mice].

verb(plural) --> [scare].
verb(singular) --> [scares].
verb(plural) --> [hate].
verb(singular) --> [hates].
```

Parse Trees

```
% DCGrammar -- with parse tree as a result

sentence(sentence(NP,VP)) --> noun_phrase(NP), verb_phrase(VP).

noun_phrase(noun_phrase(D,NP)) --> determiner(D), noun(NP).
noun_phrase(NP) --> noun(NP).

verb_phrase(verb_phrase(V)) --> verb(V).
verb_phrase(verb_phrase(V,NP)) --> verb(V), noun_phrase(NP).

% Vocabulary

determiner(determiner(the)) --> [the].
determiner(determiner(a)) --> [a].

noun(noun(cat)) --> [cat].
noun(noun(cats)) --> [cats].
noun(noun(mouse)) --> [mouse].
noun(noun(mice)) --> [mice].

verb(verb(scare)) --> [scare].
verb(verb(scares)) --> [scares].
verb(verb(hate)) --> [hate].
verb(verb(hates)) --> [hates].
```

Simple Semantics for Natural Language Sentences

Transitive and intransitive verbs

```
% DCGrammar -- Transitive and intransitive verbs

sentence(VP) --> noun_phrase(Actor), verb_phrase(Actor,VP).

noun_phrase(Actor) --> proper_noun(Actor).

verb_phrase(Actor,VP) --> intrans_verb(Actor,VP).
verb_phrase(Actor,VP) --> transitive_verb(Actor,Something,VP),
                            noun_phrase(Something).

% Vocabulary

proper_noun(john)  --> [john].
proper_noun(annie) --> [annie].

intrans_verb(Actor,paints(Actor))  --> [paints].

transitive_verb(Somebody,Something,likes(Somebody,Something)) --> [likes].
```

Determiners -- `a' and `every'

```
:- op( 100, xfy, and).
:- op( 150, xfy, =>).


% DCGrammar -- Transitive and intransitive verbs

sentence(S) --> noun_phrase(X,Assn,S), verb_phrase(X,Assn).

noun_phrase(X,Assn,S) --> determiner(X,Prop,Assn,S), noun(X,Prop).

verb_phrase(X,Assn) --> intrans_verb(X,Assn).

% Vocabulary

determiner(X,Prop,Assn,exists(X,Prop and Assn)) --> [a].
determiner(X,Prop,Assn,    all(X,Prop => Assn)) --> [every].

noun(X,man(X))  --> [man].
noun(X,woman(X))  --> [woman].

intrans_verb(X,paints(X))  --> [paints].
intrans_verb(X,dances(X))  --> [dances].
```

Relative Clauses

Interleaving syntax and semantics in DCG

```
% Word level
sentence --> word(W), rest_sent(W).

rest_sent(W) --> {last_word(W)}.
rest_sent(_) --> word(W), rest_sent(W).

% Character level
word(W) --> {single_char_word(W)},   [W].
word(W) --> {multiple_char_word(W)}, [W].

% Read a sentence and return a list of words.

sentence --> {get0(C)}, word(C,W,C1), rest_sent(C1,W).

% Given the next character and the previous word,
% read the rest of the sentence

rest_sent(C,W) --> {lastword(W)}.                  % empty
rest_sent(C,_) --> word(C,W,C1), rest_sent(C1,W).

word(C,W,C1) --> {single_character(C),!,name(W,[C]), get0(C1)}, [W].  % !,.:;?
word(C,W,C2) --> {in_word(C,Cp), get0(C1), rest_word(C1,Cs,C2),
                     name(W,[Cp|Cs])},[W].
word(C,W,C2) --> {get0(C1)}, word(C1,W,C2).                  % consume blanks

% These words terminate a sentence.

lastword('.').
lastword('!').
```

```
lastword('?').

% This reads the rest of the word plus the next character.

rest_word(C,[Cp|Cs],C2) :- in_word(C,Cp), get0(C1), rest_word(C1,Cs,C2).
rest_word(C,[],C).

% These are single character words.

single_character(33).      % !
single_character(44).      % ,
single_character(46).      % .
single_character(58).      % :
single_character(59).      % ;
single_character(63).      % ?

% These characters can appear within a word.

in_word(C,C) :- C > 96, C < 123.               % a,b,...,z
in_word(C,L) :- C > 64, C < 91, L is C + 32.   % A,B,...,Z
in_word(C,C) :- C > 47, C < 58.                % 0,1,...,9
in_word(39,39).                                % '
in_word(45,45).                                % -
```

a calculator!!

Incomplete Data Structures

Objective
Outline

- Difference Lists
- Dictionaries
- Queue
- QuickSort

An incomplete data structure is a data structure containing a variable. Such a data structure is said to be `partially instantiated' or `incomplete.' We illustrate the programming with incomplete data structures by modifying the code for a binary search tree. The resulting code permits the relation `inserted_in_is` to define both the insertion and membership relations. The empty tree is represented as a variable while a partially instantiated tree is represented as a tuple.

```
create_tree(Niltree) :- var(Niltree). % Note: Nil is a variable

inserted_in_is(Item,btree(Item,L_T,R_T)).

inserted_in_is(Item,btree(ItemI,L_T,R_T)) :-
    Item @< ItemI,
    inserted_in_is(Item,L_T).

inserted_in_is(Item, btree(ItemI,L_T,R_T)) :-
    Item @> ItemI,
    inserted_in_is(Item,R_T).

inorder(Niltree,[ ]) :- var(Niltree).
inorder(btree(Item,L_T,R_T),Inorder) :-
          inorder(L_T,Left),
          inorder(R_T,Right),
          append(Left,[Item|Right],Inorder).
```

Meta Level Programming

Meta-programs treat other programs as data. They analyze, transform, and simulate other programs. Prolog clauses may be passed as arguments, added and deleted from the Prolog data base, and may be constructed and then executed by a Prolog program. Implementations may require that the functor and arity of the clause be previously declared to be a dynamic type.

Objective
Outline

- Meta-logical Type Predicates
- Assert/Retract
- System Access

Meta-Logical Type Predicates

var(V)
: Tests whether V is a variable.

nonvar(NV)
: Tests whether NV is a non-variable term.

atom(A)
: Tests whether A is an atom (non-variable term of arity 0 other than a number).

integer(I)
: Tests whether I is an integer.

number(N)
: Tests whether N is a number.

Term Comparison

X = Y
X == Y
X =:= Y

The Meta-Variable Facility

call(X)
: this

Assert/Retract

Here is an example illustrating how clauses may be added and deleted from the Prolog data base. The example shows how to simulate an assignment statement by using `assert` and `retract` to modify the association between a variable and a value.

```
:- dynamic x/1 .% this may be required in some Prologs

x(0).  % An initial value is required in this example

assign(X,V) :- Old =..[X,_], retract(Old),
                  New =..[X,V], assert(New).
```

Here is an example using the assign predicate.

```
?- x(N).

N = 0
yes
```

```
?- assign(x,5).
yes
?- x(N).

N = 5
```

Here are three programs illustrating Prolog's meta programming capability. This first program is a simple interpreter for pure Prolog programs.

```
% Meta Interpreter for pure Prolog

prove(true).
prove((A,B)) :- prove(A), prove(B).
prove(A) :- clause(A,B), prove(B).
```

Here is an execution of an append using the interpreter.

```
 ?- prove(append([a,b,c],[d,e],F)).

F = [a,b,c,d,e]
```

It is no different from what we get from using the usual run time system. The second program is a modification of the interpreter, in addition to interpreting pure Prolog programs it returns the sequence of deductions required to satisfy the query.

```
% Proofs for pure Prolog programs

proof(true,true).
proof((A,B),(ProofA,ProofB)) :- proof(A,ProofA), proof(B,ProofB).
proof(A,(A:-Proof)) :- clause(A,B), proof(B,Proof).
```

Here is a proof an append.

```
?- proof(append([a,b,c],[d,e],F),Proof).

F = [a,b,c,d,e]
Proof = (append([a,b,c],[d,e],[a,b,c,d,e]) :-
           (append([b,c],[d,e],[b,c,d,e]) :-
           (append([c],[d,e],[c,d,e]) :-
           (append([ ],[d,e],[d,e]) :- true))))
```

The third program is also a modification of the interpreter. In addition to interpreting pure Prolog programs, is a trace facility for pure Prolog programs. It prints each goal twice, before and after satisfying the goal so that the programmer can see the parameters before and after the satisfaction of the goal.

```
% Trace facility for pure Prolog

trace(true).
trace((A,B)) :- trace(A), trace(B).
trace(A) :- clause(A,B), downprint(A), trace(B), upprint(A).

downprint(G) :- write('>'), write(G), nl.
upprint(G)   :- write('<'), write(G), nl.
```

Here is a trace of an append.

```
?- trace(append([a,b,c],[d,e],F)).
>append([a,b,c],[d,e],[a|_1427104])
```

```
>append([b,c],[d,e],[b|_1429384])
>append([c],[d,e],[c|_1431664])
>append([ ],[d,e],[d,e])
<append([ ],[d,e],[d,e])
<append([c],[d,e],[c,d,e])
<append([b,c],[d,e],[b,c,d,e])
<append([a,b,c],[d,e],[a,b,c,d,e])

F = [a,b,c,d,e]
```

Predictates for program manipulation

- consult(*file name*)
- var(*term*), nonvar(*term*), atom(*term*), integer(*term*), atomic(*term*)
- functor(*Term*,*Functor*,*arity*), arg(*N*,*term*,*N-th arg*), *Term =..List*
- call(*Term*)
- clause(*Head*,*Body*), assertz(*Clause*), retract(*Clause*)

Second-Order Programming

Objective:

Second-Order Programming

Outline:

- Setof, Bagof, Findall
- Other second-order predicates
- Applications

Setof, Bagof and Findall

Other second-order predicates

has_property, map_list, filter, foldr etc

- Variable predicate names

```
p(P,X,Y) :- P(X,Y).

p(P,X,Y) :- R =..[P,X,Y], call(R).
```

For the following functions let S be the list [S_1,...,S_n].

1. The function `map` where `map(f,S)` is [f(S_1),...,f(S_n)].
2. The function `filter` where `filter(P,S)` is the list of elements of S that satisfy the predicate P.
3. The function `foldl` where `foldl(Op,In,S)` which folds up S, using the given binary operator Op and start value In, in a left associative way, ie, foldl(op, r,[a,b,c]) = (((r op a) op b) op c).
4. The function `foldr` where `foldr(Op,In,S)` which folds up S, using the given binary operator Op and start value In, in a right associative way, ie, foldr(op,r,[a,b,c]) = a op (b op (c op r)).
5. The function `map2` is similar to `map`, but takes a function of two arguments, and maps it along two argument lists.
6. The function `scan` where `scan(op, r, S)` applies `foldl op r)` to every initial segment of a list. For example `scan (+) 0 x)` computes running sums.
7. The function `dropwhile` where `dropwhile(P,S)` which returns the suffix of S where each element of the prefex satisfies the predicate P.
8. The function `takewhile` where `takewhile(P,S)` returns the list of initial element of S which satisfy P.
9. The function `until` where `until(P,F,V)` returns the result of applying the function F to the value the smallest

number of times necessary to satisfy the predicate. Example until (>1000) (2*) 1 = 1024

10. The function `iterate` where `iterate(f,x)` returns the infinite list [x, f x, f(f x), ...]
11. Use the function `foldr` to define the functions, sum, product and reverse.
12. Write a generic sort program, it should take a comparison function as a parameter.
13. Write a generic transitive closer program, it should take a binary relation as a parameter.

Applications

Generalized sort, transitive closure ...

```
transitive_closure(Relation,Item1,Item2) :- Predicate =..[Relation,Item1,Item2],
                                            call(Predicate).
transitive_closure(Relation,Item1,Item2) :- Predicate =..[Relation,Item1,Link],
                                            call(Predicate),
transitive_closure(Relation,Link,Item2).
```

Database Programming

Objective:

Logic Programming as Database Programming

Outline

- Simple Family Database
- Recursive Rules
- Logic Programming and the Relational Database Model (relational algebra)

Simple Databases

Basic predicates: `father/2,mother/2, male/1, female/1`.

```
father(Father,Child).
mother(Mother,Child).
male(Person).
female(Person).
son(Son,Parent).
daughter(Daughter,Parent).
parent(Parent,Child).
grandparent(Grandparent,Grandchild).
```

Question: Which should be facts and which should be rules? Example: if parent, male and female are facts then father and mother could be rules.

```
father(Parent,Child) :- parent(Parent,Child), male(Parent).
mother(Parent,Child) :- parent(Parent,Child), female(Parent).
```

Some other relations that could be defined are.

```
mother(Woman) :- mother(Woman,Child).
parents(Father,Mother) :- father(Father,Child), mother(Mother,Child).
brother(Brother,Sibling) :- parent(P,Brother), parent(P,Sibling),
    male(Brother), Brother  Sibling.
uncle(Uncle,Person) :- brother(Uncle,Parent), parent(Parent,Person).
sibling(Sib1,Sib2) :- parent(P,Sib1), parent(P,Sib2), Sib1 =\= Sib2.
cousin(Cousin1,Cousin2) :- parent(P1,Cousin1), parent(P2,Cousin2),
```

```
   sibling(P1,P2).
```

What about: sister, niece, full_ sibling, mother_in_law, etc.

Recursive Rules

```
ancestor(Ancestor,Descendent) :- parent(Ancestor,Descendent).
ancestor(Ancestor,Descendent) :- parent(Ancestor,Person),
   ancestor(Persion,Descendent).
```

The `ancestor` relation is an example of the more general relation of transitive closure. Here is an example of the transitive closure for graphs. Transitive closure: connected

```
edge(Node1,Node2).
...
connected(Node1,Node2) :- edge(Node1,Node2).
connected(Node1,Node2) :- edge(Node1,Link), connected(Link,Node2).
```

Logic programs and the relational database model

The mathematical concept underlying the relational database model is the set-theoretic *relation*, which is a subset of the Cartesian product of a list of domains. A domain is a set of values. A *relation* is any subset of the Cartesian product of one or more domains. The members of a relation are called *tuples*. In relational databases, a relation is viewed as a table. The Prolog view of a relation is that of a set of named tuples. For example, in Prolog form, here are some unexpected entries in a city-state-population relation.

```
city_state_population('San Diego','Texas',4490).
city_state_population('Miami','Oklahoma',13880).
city_state_population('Pittsburg','Iowa',509).
```

In addition to defining relations as a set of tuples, a relational database management system (DBMS) permits new relations to be defined via a query language. In Prolog form this means defining a rule. For example, the sub-relation consisting of those entries where the population is less than 1000 can be defined as follows:

```
smalltown(Town,State,Pop) :- city_state_pop(Town,State,Pop), Pop < 1000.
```

One of the query languages for relational databases is the Relational Algebra. Its operations are union, set difference, Cartesian product, projection, and selection. They may be defined for two relations r and s as follows.

```
% Union of relations r/n and s/n
r_union_s(X1,...,Xn) :- r(X1,...,Xn).
r_union_s(X1,...,Xn) :- s(X1,...,Xn).

% Set Difference  r/n $\setminus$ s/n
r_diff_s(X1,...,Xn) :- r(X1,...,Xn), not s(X1,...,Xn).
r_diff_s(X1,...,Xn) :- s(X1,...,Xn), not r(X1,...,Xn).

% Cartesian product r/m, s/n
r_x_s(X1,...,Xm,Y1,...,Yn) :- r(X1,...,Xm), s(Y1,...,Yn).

% Projection
r_p_i_j(Xi,Xj) :- r(X1,...,Xn).

% Selection
r_c(X1,...,Xn) :- r(X1,...,Xn), c(X1,...,Xn).
```

```
% Meet
r_m_s(X1,...,Xn) :- r(X1,...,Xn), s(X1,...,Xn).

% Join
r_j_s(X'1,...,X'j,Y'1,...,Y'k) :- r(X1,...,Xn), s(Y1,...,Yn).
```

The difference between Prolog and a Relational DBMS is that the in Prolog the relations are stored in main memory along with the program whereas in a Relational DBMS the relations are stored in files and the program extracts the information from the files.

Expert systems

Expert systems may be programmed in one of two ways in Prolog. One is to construct a knowledge base using Prolog facts and rules and use the built-in inference engine to answer queries. The other is to build a more powerful inference engine in Prolog and use it to implement an expert system.

Pattern matching: Symbolic differentiation

```
d(X,X,1)                :- !.
d(C,X,0)                :- atomic(C).
d(-U,X,-A)              :- d(U,X,A).
d(U+V,X,A+B)            :- d(U,X,A), d(V,X,B).
d(U-V,X,A-B)            :- d(U,X,A), d(V,X,B).
d(C*U,X,C*A)            :- atomic(C), CX, d(U,X,A),!.
d(U*V,X,B*U+A*V)        :- d(U,X,A), D(V,X,B).
d(U/V,X,A)              :- d(U*V^-1,X,A)
d(U^C,X,C*U^(C-1)*W)    :- atomic(C), CX, d(U,X,W).
d(log(U),X,A*U^(-1))    :- d(U,X,A).
```

Object-Oriented Programming

```
object( Object, Methods )

/*************************************************************************
                                    OOP
*************************************************************************/

/*=======================================================================
                        Interpreter for OOP
=======================================================================*/

send( Object, Message ) :- get_methods( Object, Methods ),
                           process( Message, Methods ).

get_methods( Object, Methods ) :- object( Object, Methods ).
get_methods( Object, Methods ) :- isa( Object, SuperObject ),
                                  get_methods( SuperObject, Methods ).

process( Message, [Message|_] ).
process( Message, [(Message :- Body)|_] ) :- call( Body ).
process( Message, [_|Methods] ) :- process( Message, Methods ).

/*=======================================================================
```

```
                          Geometric Shapes
=============================================================================*/

object( polygon( Sides ), [ (perimeter( P ) :- sum( Sides, P )) ] ).

object( reg_polygon( Side, N ), [ ((perimeter( P ) :- P is N*Side)),
                                  (describe :- write('Regular polygon')) ] ).


object( rectangle( Length, Width ),
                [ (area( A ) :- A is Length * Width ),
                  (describe  :- write('Rectangle of size ' ),
                                write( Length*Width)) ] ).

object( square( Side ), [ (describe :- write( 'Square with side ' ),
                                       write( Side )) ] ).

object( pentagon( Side ), [ (describe :- write('Pentagon')) ] ).

isa( square( Side ), rectangle( Side, Side ) ).
isa( square( Side ), reg_polygon( Side, 4 ) ).
isa( rectange( Length, Width ), polygon([Length, Width, Length, Width]) ).
isa( pentagon( Side ), reg_polygon( Side, 5 ) ).

isa( reg_polygon( Side, N ), polygon( L ) ) :- makelist( Side, N, L ).
```

Appendix

The entries in this appendix have the form: `pred/n definition` where `pred` is the name of the built in predicate, `n` is its arity (the number of arguments it takes), and `definition` is a short explanation of the function of the predicate.

ARITHMETIC EXPRESSIONS
+, -, *, /, sin, cos, tan, atan, sqrt, pow, exp, log
I/O
see/1
: the current input stream becomes arg1
seeing/1
: arg1 unifies with the name of the current input stream.
seen/0
: close the current input stream
tell/1
: the current output stream becomes arg1
telling/1
: arg1 unifies with the name of the current output stream.
told/0
: close current output stream
read/1
: arg1 is unified with the next term delimited with a period from the current input stream.
get/1
: arg1 is unified with the ASCII code of the next printable character in the current input stream.
write/1
: arg1 is written to the current output stream.
writeq/1
: arg1 is written to the current output stream so that it can be read with `read`.
nl/0
: an end-of-line character is written to the current output stream.
spaces/1

arg1 number of spaces is written to the current output stream.

PROGRAM STATE

listing/0

all the clauses in the Prolog data base are written to the current output stream

listing/1

all the clauses in the Prolog data base whose functor name is equal to arg1 are written to the current output stream

clause(H,B)

succeeds if H is a fact or the head of some rule in the data base and B is its body (true in case H is a fact).

PROGRAM MANIPULATION

consult/1

the file with name arg1 is consulted (loaded into the Prolog data base)

reconsult/1

the file with name arg1 is reconsulted

assert/1

arg1 is interpreted as a clause and is added to the Prolog data base (functor must be dynamic)

retract/1

the first clause which is unifiable with arg1 is retracted from the Prolog data base (functor must be dynamic)

META-LOGICAL

ground/1

succeeds if arg1 is completely instantiated (BIM)

functor/3

succeeds if arg1 is a term, arg2 is the functor, and arg3 is the arity of the term.

T =..L

succeeds if T is a term and L is a list whose head is the principle functor of T and whose tail is the list of the arguments of T.

name/2

succeeds if arg1 is an atom and arg2 is a list of the ASCII codes of the characters comprising the name of arg1.

call/1

succeeds if arg1 is a term in the program.

setof/3

arg3 is a set (list) of all instances of arg1 for which arg2 holds. Arg3 must be of the form X^T where X is an unbound variables in T other than arg1.

bagof/3

arg3 is a list of all instances of arg1 for which arg2 holds. See setof.

\+/1

succeeds if arg1 is not provable (Required instead of **not** in some Prologs if arg1 contains variables.

not/1

same as \+ but may requires arg1 to be completely instantiated

SYSTEM CONTROL

halt/0, C-d

exit from Prolog

DIRECTIVES

:- dynamic pred/n .

the predicate pred of order n is dynamic

References

Clocksin & Mellish, *Programming in Prolog* 4th ed. Springer-Verlag 1994.
Hill, P. & Lloyd, J. W., *The Gödel Programming Language* MIT Press 1994.
Hogger, C. J., *Introduction to Logic Programming* Academic Press 1984.
Lloyd, J. W., *Foundations of Logic Programming* 2nd ed. Springer-Verlag 1987.
Nerode, A. & Shore, R. A., *Logic for Applications* Springer-Verlag 1993.
Robinson, J. A., *Logic: Form and Function* North-Holland 1979.
Sterling and Shapiro, *The Art of Prolog*. MIT Press, Cambridge, Mass. 1986.

List Processing Schemata

List Schemata (processing at top level only)

List Processing Schemata	Description/Examples
Map([], []). *Map*([H_1 | T_1],[H_2 | T_2]) :- [*Element-Test*(H_1),] *Transform*(H_1, H_2), *Map*(T_1, T_2). *Map*([H | T_1], [H | T_2]) :- [*Not-Element-Test*(H),] *Map*(T_1, T_2).]	transform selected list elements
Func-Reduce([], *Neutral-Value*). *Func-Reduce*([H | T], R) :- [*Element-Test*(H),] *Func-Reduce*(T, RL), *Element-Reduce*(H, RL, R). [*Func-Reduce*([H | T], R) :- [*Not-Element-Test*(H),] *Func-Reduce*(T,R).]	
Count-if([], 0). *Count-if*([H | T], C) :- *Element-Test(H*), *Count-if*(T, CL), C is CL + 1. *Count-if*([H | T], C) :- [*Not-Element-Test*(H),] *Count-if*(T, C).	count the number of elements satisfying the test
Count-if(L,C) :- *Count-if*(L,0,C). *Count-if*([],C,C). *Count-if*([H | T], PC, C) :- *Element-Test(H*), NPC is PC + 1, *Count-if*(T, NPC,C). *Count-if*([H | T], PC, C) :- [*Not-Element-Test*(H),] *Count-if*(T, PC, C).	Tail recursive version using an ***accumulator***
Count-if-not([], 0). *Count-if-not*(H | T], C) :- *Element-Test*(H), Count-if-not(T, C). *Count-if-not*([H | T], C) :- [*Not-Element-Test*(H),] *Count-if-not*(T, CL), C is CL + 1.	count the number of elements failing the test

Find-if(E, [H | _]) :-*Element*(E, H). *Find-if*(E, [_ | T]) :- *Find-if*(E, T). *Every*([]). *Every*([H | T]):- *Element-Test*(H), *Every*(T). *Some*([H | _]) :- *Element-Test*(H). *Some*([H | T]) :- [*Not-Element-Test*(H)] *Some* (T). *None*([]). *None*([H | T]) :- *Element-Test*(H), !, fail. *None*([H | T]) :- [*Not-Element-Test*(H),] *None*(T). *Some-not*([H | T]) :- *Element-Test*(H), *Some-not*(T). *Some-not*([H | _]) [:- *Not-Element-Test*(H)].	member(X,L)
Remove-if([], []). *Remove-if*([H | T1], T2) :- *Element-Test*(H), *Remove-if*(T1, T2). *Remove-if*([H | T1], [H | T2]) :- [*Not-Element-Test*(H),] *Remove-if*(T1, T2). *Remove-if-not*([], []). *Remove-if-not*([H |T1], [H | T2] :- *Element-Test*(H), *Remove-if_not*(T1, T2). *Remove-if_not*(H | T1, T2) :- [*Not-Element-Test*(H),] *Remove-if_not*(T1, T2).	remove_duplicates(L, ND)

List of Lists Schemata (processing at all levels)

	Examples

DMap([H_1 | T_1], [H_2 | T_2]) :-
  DMap(H_1, H_2),
  DMap(T_1, T_2).
DMap(E_1, E_2) :-
  Test(E_1[, E_2]), *Transform*(E_1, E_2).
DMap(E, E &1) [:- *Not-Test*(E &2)].	

```
DFunc-Reduce(H | T], R) :-
   DFunc-Reduce(H, RH),
   DFunc-Reduce(T, RT),
   Reduction(RH, RT, R).
DFunc-Reduce(E, E) :- Test(E).
DFunc-Reduce(E, Neutral-Value) [:- Not-Test(E )].
```

```
DCount-if([H | T], C) :-
   DCount-if(H, CH),
   DCount-if(T, CT),
   C is CH + CT.
DCount-if(E, 1) :- Test(E).
DCount-if(E, 0) [:- Not-Test(E)].
```

```
DCount-if-not([H | T], C) :-
   DCount-if-not(H, CH),
   DCount-if-not(T, CH),
   C is CH + CT.
DCount-if-not(E, 0) :- Test(E).
DCount-if-not(E, 1) [:- Not-Test(E)].
```

```
DFind-if(E, [H | T]) :-DFind-if(E, H) ; DFind-if(E, T).
DFind-if(E, H) :- Test(E, H).

DEvery([]).
DEvery([H | T]) :- DEvery(H), DEvery(T).
DEvery(E) :- Test(E).

DSome([H | T]) :- DSome(H) ; DSome(T).
DSome(E) :- Test(E).

DNone([H | T]) :- DNone(H), DNone(T).
DNone(E) :- Test(E), !, fail.
DNone(E).

DSome-not([H | T]) :- DSome-not(H) ; DSome-not(T).
[DSome-not([]) :- !, fail.]
DSome-not(E) :- Test(E & 2), !, fail.
DSome-not(E).
```

```
DRemove-if([H1 | T1], R) :-
   DRemove-if(H1, H2), DRemove-if(T1, T2),
   combine(H2, T2, R).
DRemove-if(E, remove-flag) :- Test(E).
DRemove-if(E, E) [:- Not-Test(E)].

[DRemove-if-not([],[].]
DRemove-if-not([H1 | T1], R) :-
   DRemove-if-not(H1, H2), DRemove-if-not(T1, T2),
   combine(H2, T2, R).
DRemove-if-not(E, E) :- Test(E).
DRemove-if-not(E, remove-flag) [:- Not-Test(E)].

combine(remove-flag, T, T).
combine(H, T, [H | T]).
```

References

T.S. Gegg-Harrison. Representing logic program schemata in Prolog. In *Proc. of the 12th Int. Conf. On Logic Programming*, L. Sterling (Ed.), 467-481. MIT Press, 1995.

T.S Gegg-Harrison. Extensible logic program schemata. In *Proc. of the 6th Int. Conf. on Logic Program Synthesis and Transformation*, I. Gaallagher (Ed.). Springer-Verlag, 1996.

L.S. Sterling, M. Kirshenbaum. Applying techniques to skeletons. In *Constructing Logic Programs*, J.M.Jacquet (Ed.), 127-140. John Wiley, 1993.

W.W. Vasconcelos, M. Vaargas-Vera, D.S. Robertson. Building Large-Scale Prolog Programs using a Techniques Editing System. In *Int. Logic Programming Symposium*. The MIT Press. 1993

W. W. Vasconclos, N.E. Fuchs. Prolog Program Development via Enhanced SchemaBased Transformations. In *Proc. of 7th Workshop on Logic Programming Environments*, Portland, 1995.

Adapted from: **ACM SIGPLAN, volume 33, number 2 pp. 46,47**

Logic Programming

from Deville

Software Engineering

Logic programs are theories and computation is deduction from the theory. Thus the process of software engineering becomes:

- obtain a problem description
- define the intended model of interpretation (domains, symbols etc.)
- devise a suitable theory (the logic component) suitably restricted so as to have an efficient proof procedure.
- describe the control component of the program
- use declarative debugging to isolate errors in definitions

Objective

To present a methodology for logic program development based on the following steps:

- Problem
- Specification in `Natural language'
- Logic description in First-order logic
- Logic program in Prolog

The methodology presented here is motivated by the following: ``Given a specification, how do we construct, from the given specification, a program meeting it?" The program development methodology is suitable for ``programming in the small." The development of logic programs can be based on the following three steps:

- elaboration of a specification,
- constructions of a logic description,
- derivation of a logic program in a logic programming language.

The development of a program is decomposed into several steps. The first one is the elaboration of the problem description into a specification of the solution. It should be noted that we are not concerned with specifying a problem (requirements analysis) level), but rather with specifying a procedure that solves the problem (design level). The second is the construction of a logic description in pure logic from the specification independent of any programming language or procedural semantics. The third step and final step is the derivation of a logic program (in Prolog) from the logic description. Our subject is the *rigorous* construction of logic programs which are both *correct* and *efficient*. %

Program Development

%

%

- Recursion %
- Generalization %
- Pictures/relationships %
- Correctness %
- Transparency, readability. %
- Modifiability %
- Robustness %
- Documentation %
- Efficiency %

An Example

Problem

The example problem is:

> **Problem:** Write a procedure efface(X, L, LEff) which removes the first occurrence of X from the list L, giving the list LEff. If there is no such X in the list L, it should fail.

Specification

No particular specification language is imposed. The specification is basically an informal description of a relation.

> The specification must provide to the *intended user* **all** the information that he will need to use the program correctly, and **nothing more.**
>
> The specification must provide to the *implemented* **all** the information about the intended use that he needs to complete the program and **no additional information.** Parnas

Type information is added, as well as the directionalities for which the program has to be correct. If some side-effects need to be specified, they will be described in an extra part of the specification.

```
Specification:
procedure: efface( X, L, LEff )
Type: X : Term
              L, LEff : lists
Relation: X is an element of L and LEff is the list L without
the
    first occurrence of X in L.
Application conditions:
   in( any, ground, any ): out( ground, ground, ground )
   in( ground, any, ground ): out( ground, ground, ground )
```

\begin{figure} \rule{4.5in}{.01in}

```
Specification:
procedure: p( T1, ..., Tn )

Type: T1 : type1
              ...
              Tn : typen

Restriction on Parameters:

Relation: Description of a relation between p and the
parameters T1...Tn.

Application-conditions:
   directionality: in( m1,...,mn ) : out( M1,...,Mn )
   environment precondition

Side-effects:
    side-effects description
```

\caption{General form of a specification} \rule{4.5in}{.01in} \end{figure}

Logic Description

The form of the constructed logic description will be as follows:

```
relation( parameters ) <-->  \/ C1 &
F1
    \/ C2 & F2
      .
      .
    \/ Cn & Fn
```

where C_i and F_i are formulas. The logic description is constructed in several steps:

1. *Choice of an induction parameter:* L.
2. *Choice of a well-founded relation:* $l_1 < l_2$ iff l_1 is a proper suffix of l_2.
3. *Structural forms of the induction parameter* C_1 : L empty : L = [ ] C_2 : L non-empty : L = [H|T]
4. *Construction of the structural cases* F_i must be a necessary and sufficient condition to have `efface( X, L, LEff )` when C_i is true:
   - for L = [ ], it is impossible to have list LEff which is the list L without the first occurrence of X. We obtain F_1 false.
   - For L = [H|T], there are two possibilities depending on whether H = X or not:
     1. For H = X, the necessary and sufficient condition is LEff = T, because T is the list L without the first occurrence of X.
     2. For H != X, the necessary and sufficient condition is that LEff must be of the form [H|TEff], where TEff is the list T without the first occurrence of X.

     We obtain F_2:

```
( H = X & LEff = T
 \/ H != X & efface( X, T, TEff ) & LEff = [H|TEff] ).
```

Note that T < L according to the well-founded relation when L is any ground list since T is a proper suffix of L. The constructed logic description is thus:

```
Logic Description:
efface( X, L, LEff ) <-->
     L = [ ] & false
         \/ L= [H|T] & ( H = X & LEff = T
           \/ H != X & efface( X, T, TEff )
            & LEff = [H|TEff] )
```

where the variables H, T and TEff are assumed to be existentially quantified on the right hand side of this logic description.

Logic Program

A logic program is derived from the logic description by translation to program clauses. In order to obtain a correct Prolog procedure, a permutation of the literals must be found such that Prolog's computation rule is safe, and the directionality of the recursive call `efface( X, T, TEff )` respects the specifications.

```
Logic Program:
efface( X, L, LEff ) :- L = [H|T], H = X, LEff = T.
efface( X, L, LEff ) :-  L = [H|T], LEff = [H|TEff], efface( X, T, TEff
)
    not ( H = X ).
```

A more efficient version:

```
efface( X, [X|T], T ).
efface( X, [H|T], [H|TEff] ) :- efface( X, T, TEff), not ( X = H ).
```

Documentation

Specification:
procedure: efface(X, L, LEff)
Type: X : Term
L, LEff : lists
Relation: X is an element of L and LEff is the list L without the first occurrence of X in L.
Application conditions:
in(ground, ground, any): out(ground, ground, ground)

Logic Description:
Induction Parameter: L
Well-founded relation: Proper suffix

```
efface(    X, L, LEff ) <-->
     L = [ ] & false
          \/ L= [H|T] &   ( H = X & LEff = T & list(T)
            \/ H != X   & efface( X, T, TEff )
            &  LEff = [H|TEff] )
```

Logic Procedure:

```
efface( X, L, LEff ) :- L = [H|T], H = X, LEff = T.

efface( X, L, LEff ) :-   L = [H|T], LEff = [H|TEff], efface( X, T, TEff
)

    not ( H = X ).
```

Prolog Code:

```
efface( X, [X|T], LEff ) :- !, LEff = T.

efface( X, [H|T], [H|TEff] ) :- efface( X, T, TEff).
```

Efficiency

Objective:

- Nondeterminism
- Structure Sharing
- Improved Algorithm/Data Structures
- Replace Recursion with iteration
- Use of side effects

Efficiency in programs is often determined by the choice of algorithm and data structures. Inefficiency in Prolog programs occurs in backtracking and in the copying of data structures. With respect to backtracking, the principle is to avoid unnecessary backtracking and stop execution of useless alternatives as soon as possible. With respect to copying of data structures, take advantage of structure sharing. The efficiency of Prolog programs may be improved by

- Replacing non-determinism with determinism
- Taking advantage of structure sharing
- Replacing recursion with iteration
- Use of improved algorithm/data structure
- Use of side effects.

The best source for the improvement is the choice of algorithm. Here are some additional hints:

- Use good goal ordering, `` Fail as early as you can."
- Eliminate nondeterminism by using explicit conditions and cuts.
- Use a more suitable data structure to represent data objects so that operations on objects can be implemented more efficiently.
- Minimize the number of data structures generated.

Programming Tricks

To determine if a goal is true with out affecting the current state of the variable bindings.

```
... not not A, ...
```

Transformation Rules

Many of the transformation presented in this section involve the use of the cut (!). Cuts are classified as either *green* or {\em red depending on whether their addition or removeal does not affect correctness (green) or affects correcteness (red).

> **Definition:** A literal $p(t_1,...,t_n)$ is **deterministic** iff ithe sequence of answer substitutions for this literal has *at most one* computed answer substitution.
>
> A literal $p(t_1,...,t_n)$ is **fully deterministic** iff ithe sequence of answer substitutions for this literal has *one and only one* answer substitution.
>
> A literal $p(t_1,...,t_n)$ is **infinite** iff ithe sequence of answer substitutions for this literal is *infinite*.
>
> A literal $p(t_1,...,t_n)$ is **incompatible** with the literal $q(s_1,...,s_m)$ iff the sequence of answer substitutions for $q(s_1,...,s_m)$ is empty when the sequence of answer substitutions for $p(t_1,...,t_n)$ is not empty.

Transformations based on equivalent SLDNF-trees

Transformation 1: (Factor common code)

```
p( x ) :- T, S1.
p( x ) :- T, S2.
\rule{1.5in}{.01in}
p( x ) :- T, p1( y ).
p1( y ) :- S1.
p1( y ) :- S2.
```

where

- y is the n-tuple of all the variables occuring in S_1 and S_2,
- p has no side effects,
- T, S_1, S_2 contain no cuts, and

- p is not infinite

Transformation 2a: (if-then-else)

```
p( x ) :- C, S1.
p( x ) :- not C, S2.
\rule{1.5in}{.01in}
p( x ) :- C, !, S1.
p( x ) :- S2.
```

where C has no side effects. The cut is red since its presence is essential to maintain correctness.

Transformation 2b: (if-then-else)

```
p( x ) :- C, S1.
p( x ) :- not C, S2.
\rule{1.5in}{.01in}
p( x ) :- C -> S1; S2.
```

where C has no side effects.

Transformation 3

```
p( x ) :- not C, S1.
p( x ) :- C, S2.
\rule{1.5in}{.01in}
p( x ) :- not C, !, S1.
p( x ) :- S2.
```

where C has no side effects.

Transformation 4

```
p( x ) :- T, C, S1.
p( x ) :- T, not C, S2.
\rule{1.5in}{.01in}
p( x ) :- T, C, !, S1.
p( x ) :- T, S2.
```

where

- T and C have no side effects, and
- T is deterministic.

Transformation 5 (Simple Case)

```
p( x ) :- C1, S1.
p( x ) :- C2, S2.
\rule{1.5in}{.01in}
p( x ) :- C1, !, S1.
p( x ) :- C2, S2.
```

where

- C_1 is deterministic,
- C_1 and C_2 have no side effects,
- C_1 and C_2 are compatible

Transformation 6: (Case)

```
p( x ) :- C1, S1.
p( x ) :- C2, S2.
...
p( x ) :- Cn-1, Sn-1.
p( x ) :- Cn, Sn.
\rule{1.5in}{.01in}
p( x ) :- C1,!, S1.
p( x ) :- C2,!, S2.
...
p( x ) :- Cn-1,!, Sn-1.
p( x ) :- Cn, Sn.
```

where

- the C_i are deterministic,
- the C_i have no side effects,
- each C_i is incompatible with C_j (i!=j).

These cuts are green since they do not affect correctness.

Transformation 7 (Case with an else)

```
p( x ) :- C1, S1.
p( x ) :- C2, S2.
...
p( x ) :- Cn-1, Sn-1.
p( x ) :- not C1,...,not Cn-1, Sn.
\rule{1.5in}{.01in}
p( x ) :- C1,!, S1.
p( x ) :- C2,!, S2.
...
p( x ) :- Cn-1,!, Sn-1.
p( x ) :-  Sn.
```

where

- the C_i have no side effects,
- each C_i is incompatible with C_j (i!=j).

Transformations based on computation and search rules

Transformation 8

```
p( s1 ) :- S1.
```

```
...
p( s_n ) :- S_n1, S_n2.
\rule{1.5in}{.01in}
p( s_1 ) :- S_1.
...
p( s_n ) :- S_n1, !, S_n2.
```

where

- S_n1 has no side effects,
- S_n1 is deterministic.

Transformation 9

```
p( s ) :- S_1, !, S_2, S_3.
\rule{1.5in}{.01in}
p( s ) :- S_1, !, S_2,!, S_3.
```

where

- S_2 has no side effects, and
- S_2 is deterministic.

Transformations based on partial evaluation

Transformation 10: (unfolding)

```
p( s_1 ) :- S_1.
...
p( s_i ) :- S_i1, q(t_i), S_i2.
...
p( s_n ) :- S_n.
q( t ) :- T.
\rule{1.5in}{.01in}
p( s_1 ) :- S_1.
...
p( s_i ) :- S_i1, t = t_i, T, S_i2.
...
p( s_n ) :- S_n.
q( t ) :- T.
```

where

- there is no common variable between, t, T and s_i, t_i, S_i1, S_i2, and
- T contains no cuts.

Transformations based on equality substitutions

Transformation 11

```
p( s ) :- S_1, Y = t, S_2.
\rule{1.5in}{.01in}
p( s ) :- S_1, Y = t, S_2\{ Y/t \}.
```

Transformation 12

```
p( s ) :-  Y = t, S.
\rule{1.5in}{.01in}
p( s\{Y/t\} ) :- Y = t, S2\{ Y/t \}.
```

Transformation 13

```
p( s ) :- S1, Y = t, S2.
\rule{1.5in}{.01in}
p( s ) :- S1,  S2\{ Y/t \}.
```

where Y does not occur in s, S_1, S_2 and t. **Transformation 14**

```
p( s ) :- S1, !, q(t), S2.
\rule{1.5in}{.01in}
p( s ) :- S1, q(t), !, S2.
```

where

- q has no side effects
- q(t) is fully deterministic

Transformations based on tail recursion

This transformation occurs at the logic description level rather than at the Prolog level.

> **Definition:** A logic procedure p is **tail recursive** iff it has one and only one recursive subgoal and its last program clause has the form
>
> ```
> p( s ) :- S, p(t).
> ```
>
> where S is deterministic. When the last program clause of a prodecure has this form but the procedure has more than one recuresive subgoal, the procedure is said to be **semi-tail recursive**.

Replace

```
fac( 0, 1  ).
fac( N, NF ) :- N > 0, N1 is N-1, fac( N1, N1F ), NF is N*N1F.
```

with

```
fac( N, NF ) :- fac( N, 1, NF ).

fac( 0, NF, NF ).
fac( N, F,  NF ) :- N > 0, N1 is N-1, TF is N*F, fac( N1, TF, NF ).
```

Replace

```
reverse( [], [] ).
reverse( [H|L], Rev ) :- reverse( L, RL ), concat( RL, [H], Rev ).
```

with

```
reverse( List, Reverse ) :- reverse( List, [], Reverse ).

reverse( [], Reverse, Reverse ).
reverse( [H|List], Rev, Reverse ) :- reverse( List, [H|Rev], Reverse ).
```

Transformations based on Prolog implementation techniques

The binding for a variable which occurs just once in a program clause is not important during the computation process. Such variables may be replaced with an **annonymous** variable, usually indicated by the underscore `_'.

Transformation 15: (Annonymous variables)

```
p( s ) :- S.
\rule{1.5in}{.01in}
p( s\{Y/\_\} ) :- S\{ Y/\_ \}.
```

where Y occurs only once in s or in S (but not in both).

Transformation 9: (Indexing) Indexing will be explained with the following example.

```
p(a,X) :- S1.
p(b,X) :- S2.
p(c,X) :- S3.
p(d,X) :- S4.
```

With indexing, given the goal p(c,1), the Prolog interpreter immediately selects the proper rule. The list of potentially unifiable program clauses is found by a hash-code technique.

Indexing is carried out on the principle functor of the parameter so that neither ground nor variable terms can be indexed parameters.

Therefore, to utilize indexing, some permutation of the parameters may be required. Such a transformation requires a modification of the specification.

Transformations based on global parameters

There are two ways of representing data structures, terms and relations. In the relational representation the data structure is a global object. The problem with the relational representation is that modifications to the data structure can affect correctness. Transformations of this sort should be applied at the end of the optimization process (after a deterministic program has been constructed). The transformation from the term representation to the global object should be considered when the object is relative large and the manipulation and updates of the object is expensive.

Transformations based on derived facts

Replace

```
fac( 0, 1  ).
fac( N, NF ) :- N > 0, N1 is N-1, fac( N1, N1F ), NF is N*N1F.
```

with

```
fac( 0, 1  ).
fac( N, NF ) :- N > 0, N1 is N-1, fac( N1, N1F ), NF is N*N1F,
                  asserta( fac( N, NF ).
```

Note the use of asserta, the fact is add before the second defining clause.

Transformations based on structure sharing

This transformation should occur at the logic description level rather than at the Prolog level. Replace

```
sublist( Xs, AXsB ) :- prefix( AXs, AXsB ), suffix( Xs, AXs ).
```

with

```
sublist( Xs, AXsB ) :- suffix( XsB, AXsB ), prefix( Xs, XsB ).
```

since the suffix call does not create a new list (avoids consing in Lisp terminology)

Alternative Data Structures

This transformation should occur at the logic description level rather than at the Prolog level. Although, at the logic description level it may be appropriate to treat some predicates as system predicates so as to take advantage of alternative representations. We present two examples.

Difference Lists When list operations require access to the end of the list, the difference list may be a more suitable data structure.

Replace

```
quicksort( [], [] ).
quicksort( [X|List], Sorted ) :- partition( List, X, Less, More ),
                                 quicksort( Less, SLess ),
                                 quicksort( More, SMore ),
                                 concat( SLess, [X|SMore], Sorted ).
```

with

```
quicksort( List, Sorted ) :- quicksort\_dl( List, Sorted/[] ).

quicksort\_dl( [],       Z     /Z ).
quicksort\_dl( [X|List], Sorted/Z ) :- partition( List, X, Less, More ),
                                       quicksort\_dl( Less, Sorted/[X|MSort] ),
                                       quicksort\_dl( More, MSort/Z ).
```

List ordering Map coloring example from Bratko

```
neighbors( Country, Neighbors ).
...
coloring( [] ).
coloring( [Country/Color|CountryColorList] ) :-
   coloring( Country/ColorList ),
   member( Color, [yellow, blue, red, green] ),
   not ( member( Country1/Color, CountryColorList ),
            neighbor( Country, Country1 ) ).

neighbor( Country, Country1 ) :- neighbors( Country, Neighbors ),
    member( Country1, Neighbors )

country( Country ) :- neighbors( Country, \_ ).
```

The Query:

```
?-  setof( Country/Color, country(Country), CountryColorList ),
    coloring( CountryColorList ).
```

Improvement:

```
makelist( List ) :- collect( [westgermany], [], List ).

collect( [], Closed, Closed ).
collect( [X|Open], Closed, List ) :-
    member( X, Closed ), !, \% RED CUT
    collect( Open, Closed, List ).
collect( [X|Open], Closed, List) :-
    neighbors( X, NBS ),
   concat( NBS, Open, Open1 ),
   collect( Open1, [X|Closed], List ).
```

The Query:

```
?-  setof( Country/Color, country(Country), CountryColorList ),
    coloring( CountryColorList ).
```

Transformation Strategy

1. transformation 1
2. transformation 2-7
3. transformation 8
4. transformation 8,9
5. transformation 10
6. transformation 11-14
7. transformation 15

Transformations based on program clause indexing and global parameters should be performed separately; those based on global parameters should be done before any other transformations; while those based on clause indexing should be done at the very end of the transformation process.

Logic Programming: Examples

All Code

Lists and Sets

Towers of Hanoi Puzzle

Problem: There are n disks each of different size that are stacked from largest to smallest on one of three pins. The problem is to move the disks one at-a-time from the first pin to the third pin. Larger disks must not be placed on smaller disks. Only one disk may be in transit at a time.

State Machines

CFG Parser

Map Coloring

Problem: Color a map so that no two adjacent regions have the same color.

General Logic Puzzle Solver

Problem: Three friends came first, second, and third in a programming contest. Each had a different first name, liked a different sport, and had a different nationality.

Michael likes basketball and did better than the American. Simon, the Israeli, did better than the tennis player. The cricket player came first.

State Space Search Techniques

Problem: A farmer has a wolf, goat, and cabbage on the left side of a river. The farmer has a boat that can carry at most one of the three and he must transport all three to the right side of the river. He may not leave the goat alone with the cabbages or the wolf alone with the goat.

How does the farmer transport the trio across the river?

Knowledge based systems/expert systems

Natural Language Processing

Definite Clause Grammars

Compiler

Database Management System

Problem: design a query processor for a relational database management system.

AI Classics

Eliza, Prolog Consultant

Logic Programming Theory

Design and Implementation of a Prolog Compiler for Tiger

Introduction

Tiger Syntax

Lexical Issues

The language is case sensitive.

The ASCII character set is assumed.

Identifiers: An identifier is a sequence of letters, digits, and underscores, starting with a letter.
Integer Literal: An integer literal is sequence of one or more digits designating an integer value.
String Literal: A string literal begins and ends with double quotes. Embedded double quotes must be escaped, \".
Comments: A comment may appear between any two tokens. Comments start with /* and end with */ and may be nested.
Reserved words: The following are reserved words: **nil**, **of**, **if**, **then**, **else**, **while**, **do**, **for**, **to**, **break**, **let**, **in**, **end**, **type**, **array**, **of, var**, **function**.
Key words: int, **string** (predefined types that may be redefined)
Punctuation and operators: The following are punctuation symbols and operators: (,), -, `,', {, }, [, ], `.', `;', +, -, *, /, =, :, &, |, :=, <>, >, <, >=, <=

Grammar

Grammar rules take the following form, *non-terminal -> regular expression*, where the regular expression is

- **e** for the empty string
- *a b* for concatination
- (*a*) for grouping
- *a* | *b* for alternatives
- [*a*] for optional elements
- { *a* } for zero or more

- **bold** for reserved words, keywords, and symbols of the language.

Program

A program is a parameterless expression.

program -> exp

Expressions

exp -> **nil** | *integer-literal* | *string-literal* | *lvalue* | **(** *expSeq* **)** | **-** *exp* | *id***(** *expList* **)** | *exp op exp* | *type-id* **{** { *id* **=** *exp***,** } *id* =*exp* } **}** | *type-id* **[** *exp* **]** **of** *exp* | *lvalue* **:=** *exp* | **if** exp_1 **then** exp_2 | **if** exp_1 **then** exp_2 [**else** exp_3] | **while** *exp* **do** *exp* | **for** *id* **:=** *exp* **to** *exp***do** *exp* | **break** | **let** *decs* **in** *expSeq* **end**	literals function call record creation: id, type, and order must match declaration array creation: number of elements and initial value produces no value produces no value local variable *id* scope limited to for use only within while or for terminates nearest enclosing **for** or **while** expression
lvalue -> id | *lvalue* {(**.** *id* |**[** *exp* **]**)}	location whose value may be read or assigned
expSeq -> [{ *exp* **;** } *exp*]	- sequence of commands
expList -> [{ *exp* , } *exp*]	- function arguments

Operators

op -> arith-op | *rel-op* | *bool-op*	listed in order of precedence (high to low)
arith-op -> ***** | / | + | -	left associative with the usual ariithmetic precedence rules and unary minus having the highest precedence
rel-op -> = | <> | < | <= | > | >=	equal precedence; non-associative, a = b = c is not legal
bool-op -> **&** | |	The short circuit boolean conjunctions and disjunctions; integer zero is false and any nonzero integer value is true.

Declarations

decs -> { *dec* } *dec* -> *tydec* | *vardec* | *fundec*	Mutually recursive definitions must be consecutive
Data Types	
tydec -> **type** *type-id* = *ty*	
ty -> **{** *tyfields* **}** | **array of** *type-id* | *type-id*	product type; name equivalance is used array size is not part of the declaration -- problem with reduction of *id* to *type-id*
tyfields -> [{ *id* **:** *type-id* **,** }*id* **:** *type-id*]	field names are local to the product type
type-id -> *id* | **int** | **string**	built-in types may be redefined so **int** and **string**should be entered in the symbol table
Variables	
v*ardec* -> **var** *id* [**:** *type-id*] **:=** *exp*	require initialization; type required if exp is **nil**
Functions and Procedures	
fundec -> **function** *id* **(** *tyfields***)** [**:** *type-id*] = *exp*	parameters are passed by value; array and record references are passed by reference.

Problems

When the grammar is used in a bottom up parser generator, a Reduce-Reduce error occurs with *id* reductions. The solution is to replace *type-id* with *id* and let the semantic phase perform the syntax checking.

SCOPE RULES

let ...x... in exp end -- scope starts with x and ends at the end
function f(...x....) = exp -- scope starts with x and lasts throughout exp
nesting -- permitted
name spaces -- type name space, function and variable name space
redeclarations -- a let declaration hides global name declarations

SEMANTICS

Array and record variables are references, i.e., arrays and records are passed by reference, comparison tests for same instance, assignment results in aliasing.

STANDARD LIBRARY

```
function print( s : string )
function flush( )
function getchar( ) : string
function ord( s : string ) : int
function chr( i : int ) : string
function size( s : string ) : int
function substring( s : string, first : int, n : int ) : string
function concat( s1 : string, s2 : string ) : string
function not( i : int ) : int
function exit( i : int )
```

The Scanner

The Parser

Since the parser is implemented using recursive descent, the grammar was rewritten in a form suitable for top-down parsing, an LL(1) grammar.

Program
program -> exp

Expressions	
exp -> disj disjS	expressions are a sequence of disjunctions
disj -> conj conjS	disjuctions are a sequence of conjunctions
conj -> arithExp relExp	conjuctions are a sequence of arithmetic expressions
arithExp -> term termS	arithmetic expressions are a sequence of terms
relExp -> relOp arithExp | **e**	
term -> factor factorS	terms are a sequence of factors
disjS -> | *exp* | **e**	
conjS -> & *disj* | **e**	
termS -> (+ | -) *term termS* | **e**	
factorS -> (* | /) *factor factorS* | **e**	
factor -> **nil**	factors
| *integer-literal*	literals
| *string-literal*	
| **(** *expSeq* **)**	
| - *exp*	
| **if** exp_1 **then** exp_2 [**else** exp_3]	produces no value
| **while** *exp* **do** *exp*	produces no value
| **for** *id* **:=** *exp* **to** *exp* **do** *exp*	
| **break**	local variable *id* scope limited to for use only within while or for
| **let** *decs* **in** *expSeq* **end**	

	idExp	
idExp -> id (*fra* | *av*) *fra ->* **(** *expList* **)** | **{** *id* = *exp* { **,** *id* =*exp* } **}** | **[** *exp* **]** (**of** *exp* | *av*) *av ->* {(**.** *id* | **[** *exp* **]**)} [**:=** *exp*]	location whose value may be read or assigned function call record creation array creation the ambiguity is resolved by prefering *fra*
expSeq -> [*exp* { **;** *exp* }] *expList ->* [*exp* { , *exp* }]	

Operators

op -> arith-op | *rel-op* | *bool-op*	listed in order of precedence (high to low)
arith-op -> ***** | / | + | -	left associative with the usual ariithmetic precedence rules and unary minus having the highest precedence
rel-op -> = | <> | > | < | >= | <=	equal precedence; non-associative - a = b = c is not legal
bool-op -> **&** | |	The short circuit boolean conjunctions and disjunctions; integer zero is false and any nonzero integer value is true.

Declarations

The keywords, **int** and **string**, should be inserted into the symbol table as part of the initialization process.

decs -> { *dec* } *dec -> tydec* | *vardec* | *fundec*
Type declaration
tydec -> **type** *type-id* = *ty*
ty -> **{** *tyfields* **}** | **array of** *type-id* | *type-id*
tyfields -> [*id* **:** *type-id* { **,** *id* : *type-id* }]
type-id -> id | **int** | **string**
Variable declaration
v*ardec ->* **var** *id* [**:** *type-id*] **:=** *exp*
Function declaration
fundec -> **function** *id* **(** *tyfields***)** [**:** *type-id*] = *exp*

Abstract Syntax

Expressions

expressions
null nilExp intExp(Int) stringExp(String) varExp(Var) callExp(Symbol, expList) opExp(Exp, Op, Exp) arrayExp(Symbol, Size, Exp) recordExp(Symbol, FieldExpList) seqExp(expList) assignExp(Var, Exp) ifExp(Test, ThenExp, null) ifExp(Test, ThenExp, ElseExp) whileExp(Test, Exp) forExp(VarDec, High, Body) breakExp letExp(DecList, Exp)
variables
simpleVar(Symbol) fieldVar(Var, Symbol) subscriptVar(Var, Exp)

Declarations

declarations
functionDec(Symbol, fieldList, nameType, exp, functionDecNext) varDec(Symbol, nameType, Exp) typeDec(Symbol, Type, typeDec Next)
type
nameType(Symbol) recordType(FieldList) arrayType(Symbol)
miscellaneous

decList(Dec, DecList) expList(Exp, ExpList) fieldExpList(Symbol, Exp, FieldExpList) fieldList(Symbol, SymbolType, FieldList)
operators

plus	**eq**	**lt**	**gt**
minus	**ne**	**le**	**ge**
mul			
div			

Translation of concrete syntax to abstract syntax; parse(CST, AST)

Concrete Syntax	***translation to abstract syntax***
expressions	
lvalue	varExp(*lvalue*)
nil	nilExp
integer-literal	intExp(*integer-literal*)
string-literal	stringExp(*string-literal*)
id(null)	callExp(*id*, expList(null))
id(*expList*)	callExp(*id*, expList(H,T))
Exp Op Exp	opExp(*Exp*, *Op*, *Exp*)
id **{** *fieldExpList* **}**	recordExp(*id*, *fieldExpList*)
(*ExpSeq*)	seqExp(expList(*Exp*, *ExpList*))
( )	seqExp(null)
Var **:=** *Exp*	assignExp(*Var*, *Exp*)
if *Exp* **then** *Exp*	ifExp(*Exp*, *Exp*, null)
if *Exp* **then** *Exp* **else** *Exp*	ifExp(*Exp*, *Exp*, *Exp*)
while *Exp* **do** *Exp*	whileExp(*Exp*, *Exp*)
for *id* **:=** *Exp* **to** *Exp* **do** *Exp*	forExp(varDec(*id*, *type*, *Exp*), *Exp*, *Exp*)
break	breakExp
let *Decs* **in** *ExpSeq* **end**	letExp(decList(*Dec*, *Decs*), seqExp(expList(*Exp*, *ExpList*)))
id **[** *exp* **]** **of** *exp*	arrayExp(*id*, *exp* , *exp*)
lvalues	
id	simpleVar(*id*)
var **.** *id*	fieldVar (*var*, *id*)
var **[** *exp* **]**	subscriptVar (*var*, *exp*)
declarations	

function *id* (*tyfields*) [**:** *typeId*] = *exp* **var** *id* [**:** *typeId*] **:=** *exp* **type** *id* = *type*	functionDec(*id*, *tyfields*, *typeId*, *exp*) varDec(*id*, *typeId*, *exp*) typeDec(*id*, *typeId*)
type	
int **string** *type-id* **{** *tyfields* **}** **array of** *type-id*	nameType(*type-id*) recordType( ) arrayType(*type-id*)
miscellaneous	
dec, *decs* *exp*, *exps* **{** *id* = *exp* {**,** ...}**}** *id* **:** *type-id* { **,** ... }	decList(*dec*, *decs*) expList(*exp*, *exps*) fieldExpList(*id*, *exp*, *FieldExpList*) fieldList(*id*, *type-id*, *FieldExpList*)
operators	**operators**
+ - * / = <> < <= > >=	**plus** **minus** **mul** **div** **eq** **ne** **lt** **le** **gt** **ge**

Symbol Table & Semantic Analysis

Activation record and machine dependencies

Intermediate Representation

Intermediate Representation
Expressions

const(Value) name(Label) temp(Temp) binOp(BinOp,Exp,Exp) mem(Exp) call(Function,Args) eSeq(Stmt, Exp)
Statements
move(Dst,Src) exp(Exp) jump(Exp, Labels) cJump(RelOp, Exp, Exp, Label, Label) seq(Stmt, Stmt) label(Label)
other classes
expList(Head, Tail) stmList(Head, Tail)
Constants

Translation of abstract syntax to intermediate representation WARNING: the following is incomplete

Abstract Syntax	***Intermediate Representation***
variables	
simpleVar(*id*) fieldVar(Var, Symbol) subscriptVar(Var, Exp)	mem(*id*) --*var* **.** *id* --*id* **[** *exp* **]**
expressions	
varExp(*Var*) nilExp intExp(*Int*) stringExp(String) callExp(*Symbol*, *expList*) opExp(Exp, Op, Exp) recordExp(Symbol, FieldExpList) seqExp(null) seqExp(expList(*head*, *tail*)) assignExp(Var, Exp) ifExp(Test, ThenExp, null) ifExp(Test, ThenExp, ElseExp) whileExp(Test, Exp)	mem(*Var*) **--nil** const(*Int*) --*string-literal* call(mem(*Symbol*), Args) binOp(*Op*, *Exp*, *Exp*) --*id* **{** *fieldExpList* **}** seq(null) seq(*head*, *Exp*) move(*Var*, *Exp*) cJump(RelOp, Exp, Exp, Label, Label) cJump(RelOp, Exp, Exp, Label, Label) **--while** *Exp* **do** *Exp*

forExp(VarDec, High, Body) breakExp letExp(DecList, Exp) arrayExp(Symbol, Size, Exp)	letExp(*VarDec*, whileExp(*High*, *Body*)) **--break** **--let** *Decs* **in** *ExpSeq* **end** *--id* **[** *exp* **]** **of** *exp*
declarations	
functionDec(Symbol, fieldList, nameType, exp, functionDecNext) varDec(Symbol, nameType, Exp) typeDec(Symbol, Type, typeDec Next)	**--function** *id* (*tyfields*) [**:** *typeId*] = *exp* **--var** *id* [**:** *typeId*] **:=** *exp* **--type** *id* = *type*
type	
nameType(Symbol) recordType(FieldList) arrayType(Symbol)	**--int** **--string** *--type-id* **--{** *tyfields* **}** **--array of** *type-id*
miscellaneous	
decList(Dec, DecList) expList(Exp, ExpList) fieldExpList(Symbol, Exp, FieldExpList) fieldList(Symbol, SymbolType, FieldList)	*--dec, decs* *--exp, exps* **--{** *id = exp* {**,** ...}**}** *--id* **:** *type-id* { **,** ... }
operators	
plus **minus** **mul** **div** **eq** **ne** **lt** **le** **gt** **ge**	**plus** **minus** **mul** **div** **eq** **ne** **lt** **le** **gt** **ge**

Code Generation

Stack machine code
Expressions

push(A) BinOp call(F) store(V) load(V) jmp(L) jmpF(L)
Statements
other classes
expList(Head, Tail) stmList(Head, Tail)
Constants

Intermediate Representation	***Stack machine Assembly Code***
Expressions	
const(Value) name(Label) temp(Temp) binOp(BinOp,Exp,Exp) mem(Exp) call(Function,Args) eSeq(Stmt, Exp)	push(C) Exp, Exp, BinOp Exp, load Args, call Function
Statements	
move(Dst,Src) exp(Exp) jump(Exp, Labels) cJump(RelOp, Exp, Exp, Label, Label) seq(Stmt, Stmt) label(Label)	Src, store(Dst) Exp Exp, jmp Exp, Exp, RelOp, jmpfalse Label stmt, stmt
other classes	
expList(Head, Tail) stmList(Head, Tail)	head, ... head, ...
operators	

plus	**plus**
minus	**minus**
mul	**mul**
div	**div**
eq	**eq**
ne	**ne**
lt	**lt**
le	**le**
gt	**gt**
ge	**ge**

```
:-
module(scan,[initScan/1,scanToken/2,currentToken/2,currentToken/3,ctType/2,ctSpelling/2,ctValue/2,relOp/1,period/1,
lbracket/1]).

% DEBUGGING
rc :- consult(scan).
es :- edit(scan).
ts :- see('Test/scan'), getChar(C), loop(C), seen.
loop(C) :- eof(C), !.
loop(C) :- scanToken(C,Tok,Cn),!, write_ln(Tok), loop(Cn).
% End DEBUGGING

/*
INPUT
   CurrentToken-CurrentCharacter
TOKENS
   ct(ReservedWord         ) -- Reserved word
   ct(id,          Spelling) -- identifier
   ct(intLit,         Value) -- integer literal
   ct(stringLit,       Value) -- string literal
   Type                      -- for single and double charactor tokens
*/

%%% initScan/1

initScan(Tok-NC):- % initialize the scanner
     getChar(C), scanToken(_-C,Tok-NC).

scanToken(T-CC,Token-NC):- % gets next token
     scanToken(CC,Token,NC),
     write_ln(Token-NC).

currentToken(T-C,Type,NT-NC):-  % If current token       - Input
    currentToken(T-C,Type),!,   % is of appropriate type - Type
    scanToken(T-C,NT-NC).       % gets the next token    - NT

%currentToken/2 -- Extracts type of current Token
currentToken(ct(Type,Spelling)-NC,Type).
currentToken(ct(Type)-NC,         Type).
currentToken(Type-NC,             Type).

% Access functions

ctType(ct(RW),     RW  ):- !.
ctType(ct(Type,SV),Type):- !.
ctType(Type,       Type).
ctValue(ct(Type,Value),Value).
%ctValue(ct(intLit,Value),Value).

ctSpelling(ct(id,Spelling),Spelling).


%%%%%%%%%%%%%%%%%%%%%%%% scanToken/3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

scanToken(CC,eof,CC) :- eof(CC),!.
scanToken(CC,Token,NC) :- separators(CC,Char),!, token(Char,Token,NC).

separators(CC,NC) :- blank(CC),!, getChar(NC).
separators(CC,CC).


%%%%%%%%%%%%%%%%%%%%%%%% token/3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

token(CC,eof,CC) :- eof(CC).

% Identifiers and Reserved Words
token(CC,Token,NC):- first(id,CC),!, getChar0(IC), id(IC,ASCII,NC),
                     name(Spelling,[CC|ASCII]), screen(Spelling,Token).

% Integer Literals
token(CC,Token,NC):- first(intLit,CC),!, getChar0(IC), intLit(IC,ASCII,NC),
                     name(Value,[CC|ASCII]), Token = ct(intLit,Value).

% String Literals WARNING: DOES NOT HANDLE EMBEDDED QUOTES

token(CC,Token,NC):- first(stringLit,CC),!, getChar0(IC),
                     stringLit(IC,ASCII,NC),
                     name(Value,[CC|ASCII]), Token = ct(stringLit,Value).


% Single & Double Character Tokens
```

```
token(CC, TType, NC) :- ascii(CC,Type),
                        singleOrDouble(CC,Type,TType,NC).

singleOrDouble(CC,Type, Type,NC) :- singleChar(SC), member(Type,SC),!,
                                    getChar0(NC).

singleOrDouble(CC,Type,TType,NC) :- doubleChar(DC), member(Type,DC),!,
                                    getChar0(IC), screen(Type,IC,IType,I2C),
                                    comment(IType,I2C,TType,NC).

% Unrecognized character/token
singleOrDouble(CC,Type,Type,NC) :- getChar0(NC).

first(id    ,CC):- ascii(CC,letter).
first(intLit,CC):- ascii(CC,digit).
first(stringLit,CC):- quote(CC).

%%%%%%%%%%%%%%%%%%%%%%%%%%%% comment/4 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

comment(bComment,CC,Type,NC) :- !, scanToken(CC,TT,IC),
                                comments(TT,1,IC,Type,NC).
comment(Type,    NC,Type,NC). % not a comment

comments(eof,     N,CC,eof, CC):- !,
        write('Expected to find '), write(N),
        write_ln(' closing comment(s), found end of file instead.').
comments(bComment,N,CC,Type,NC):- !, M is N+1,
                                  scanToken(CC,TT,IC),
                                  comments(TT,M,IC,Type,NC).
comments(eComment,1,CC,Type,NC):- !, scanToken(CC,Type,NC).
comments(eComment,N,CC,Type,NC):- !, M is N-1,
                                  scanToken(CC,TT,IC),
                                  comments(TT,M,IC,Type,NC).
comments(_,N,CC,Type,NC):- scanToken(CC,TT,IC), comments(TT,N,IC,Type,NC).

%%%%%%%%%%%%%%%%%%%%%%% screen/2 identifiers and reserved words %%%%%%%%%%%%%%%

screen(Spelling,ct(Spelling)   ):- rw(RW), member(Spelling,RW),!.
screen(Spelling,ct(id,Spelling)). % user defined name

%finish(Token).
%start.

%%%%%%%%%%%%%%%%%%%%%%% screen/4 two character operators %%%%%%%%%%%%%%%%%%%%%%

screen(divide,  IC, bComment, NC):- ascii(IC,asterisk),!, getChar0(NC).
screen(asterisk,IC, eComment, NC):- ascii(IC,divide),  !, getChar0(NC).
screen(colon,   IC, assignOp, NC):- ascii(IC,equal   ),!, getChar0(NC).
screen(less,    IC, ltEq,     NC):- ascii(IC,equal   ),!, getChar0(NC).
screen(less,    IC, notEq,    NC):- ascii(IC,greater ),!, getChar0(NC).
screen(greater, IC, gtEq,     NC):- ascii(IC,equal   ),!, getChar0(NC).
screen(Type,    IC, Type,     IC).

%%%%%%%%%%%%%%%%%%%%%%%%%%%% getChar/1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%% getChar0/1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getChar( Char):- get( Char). %Char is next non-blank char, eof = -1.
getChar0(Char):- get0(Char). %Char is next char, eof = -1.

/* Tokens: Defined using RE.

   Reserved words are their own type.
   identifier = l(l+d)*
   number = dd*
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%% id/3 Identifier/reserved word %%%%%%%%%%%%%%%%%%

id(CurrentChar,[CurrentChar|Rest],NextChar):-
        letterOrDigit(CurrentChar),!, getChar0(C), id(C,Rest,NextChar).

id(CurrentChar,[],CurrentChar).

%%%%%%%%%%%%%%%%%%%%%%%%%%%% intLit/3 integer literal   %%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%% stringLit/3 integer literal %%%%%%%%%%%%%%%%%%%%%

intLit(CurrentChar,[CurrentChar|Rest],NextChar):-
       digit(CurrentChar),!, getChar0(C), intLit(C,Rest,NextChar).

intLit(CurrentChar,[],CurrentChar).
```

```
stringLit(CurrentChar,[CurrentChar|Rest],NextChar):-
       quote(CurrentChar) ->
             (getChar0(NextChar),Rest=[]);
             (getChar0(C), stringLit(C,Rest,NextChar)).


%%%%%%%%%%%%%%%%%%%%%%%%%%%% singlechar/1 list of single character words %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%% doublechar/1 list of double character words %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%% rw/1 reserved words                         %%%%

singleChar([lparen,rparen,lbracket,rbracket,lbrace,rbrace,
            plus,minus,
            vbar,ampersand,equal,semicolon,comma,period]).

doubleChar([asterisk,divide,less,greater,colon]).

rw([array, break, do, else, end, for, function, if, in, int, let, nil, of, string, then, to,
type, var, while]).

relOp([equal,notEq,less,ltEq,greater,gtEq]).
%%%%%%%%%%%%%%%% eof/1, digit/1, blank/1, letterOrDigit/1 %%%%%%%%%%%%%%%%%%%%%%

:- consult('/home/aabyan/lib/prolog/ascii').

period(C) :- ascii(C,period).
lbracket(C) :- ascii(C,lbracket).
quote(C) :- ascii(C,dblquote).
eof(C) :- endfile(C).
digit(C) :- ascii(C,digit).
blank(C) :- ascii(C,blank),!.
blank(C) :- \+ printable(C).
letterOrDigit(C) :- ascii(C,letter),!.
letterOrDigit(C) :- ascii(C,digit).
```

```
:- module(ast,[program/2,spy/1]).

% Parser for Tiger, returns an AST

% DEBUGGING
% PROGRAM
rc :- consult(ast).
ep :- edit(ast).
spy :- spy([exp/3,lvalue/3]).
% End DEBUGGING

program(Token-CC,AST) :- exp(Token-CC,AST,T-NC),!, currentToken(T-NC,eof).
program(Token-CC,null).

%%%%%%%%% Expressions %%% exp/3 CTok-CC, AST, NTok-NC %%%%%%%%%%%%%%%%%%%%%%%%%
%------------------------ Disjunction ----------------------------------------
%------------------------ Conjunction ----------------------------------------
%------------------------ Relational Expresssion -----------------------------
%------------------------ Term -----------------------------------------------
%------------------------ Factor ---------------------------------------------

exp(Input,AST,Rest):- disj(Input,Lhs,In1),
                      disjS(In1,Lhs,AST,Rest).

disj(Input,AST,Rest):- conj(Input,Lhs,In1),
                       conjS(In1,Lhs,AST,Rest).

conj(Input,AST,Rest):- term(Input,Lhs,In1), termS(In1,Lhs,TST,In2),
                       relExpP(In2,TST,AST,Rest).

% = <> < <= > >=
relExpP(Op-C,Lhs,AST,Rest):- currentToken(Op-C,RelOp),
                             relOp(RO), member(RelOp,RO),
                             op(Op,OP),
                             scanToken(Op-C,NT),
                             term(NT,Rhs,Rest),
                             AST=opExp(Lhs,OP,Rhs).

relExpP(In,AST,AST,In).


term(Input,AST,Rest):- factor(Input,LHS,In1), factorS(In1,LHS,AST,Rest).

% OR
disjS(Input,Lhs,AST,Rest):- currentToken(Input,vbar,In1),
                            exp(In1,Rhs,Rest),
                            AST = ifExp(opExp(Lhs,gt,intExp(0)),intExp(1),Rhs).
disjS(In,AST,AST,In).

% AND
conjS(Input,Lhs,AST,Rest):- currentToken(Input,ampersand,In1), % AND
                            disj(In1,Rhs,Rest),
                            AST = ifExp(opExp(Lhs,gt,intExp(0)),Rhs,intExp(0)).
conjS(In,AST,AST,In).

% + -
termS(Op-C,Lhs,AST,Rest):- (currentToken(Op-C,plus,In1);
                            currentToken(Op-C,minus,  In1)),!,
                           op(Op,OP),
        term(In1,Rhs,In2), termS(In2,opExp(Lhs,OP,Rhs),AST,Rest).

termS(In,AST,AST,In).

factorS(Op-C,Lhs,AST,Rest):- (currentToken(Op-C,asterisk,In1);
                              currentToken(Op-C,divide,  In1)),!,
                             op(Op,OP),
        factor(In1,Rhs,In2), factorS(In2,opExp(Lhs,OP,Rhs),AST,Rest).
```

```
factorS(In,AST,AST,In).

%----------------------- FACTOR -------------------------------------------
% NIL: nil
factor(Input,nilExp,Rest) :- currentToken(Input, nil,Rest).

% INTEGER-LITERAL: integer-literal
factor(T-NC,intExp(Value),Rest) :- currentToken(T-NC, intLit,Rest),
                                   ctValue(T,Value).
% STRING-LITERAL: string-literal.
factor(T-NC,stringExp(Value),Rest) :- currentToken(T-NC,stringLit,Rest),
                                     ctValue(T,Value).

% L-VALUE: lvalue
% L-VALUE | L-VALUE := EXP
factor(T-C,AST,Rest) :- currentToken(T-C,id,NT), !,
                        idExp(T,NT,LV,Next),
                        (currentToken(Next,assignOp,NT2) -> % V := Exp
                          (exp(NT2,EXP,Rest), AST=assignExp(LV,EXP));
                        (AST=LV,Rest=Next)).

% ( expSeq )
factor(Input,ExpSeq,Rest) :- currentToken(Input, lparen, Next),
                   expSeq(Next,ExpSeq,Last),
                   currentToken(Last,  rparen, Rest).

% IF: if exp then exp | if exp then exp [ else exp ]
factor(Input,AST, Rest) :- currentToken(Input, if, In1),
                   exp(In1,Test,In2),
                   currentToken(In2,  then, In3),
                   exp(In3, Then, In4),
                   else(In4,Else, Rest),
                   (Else = empty -> AST = ifExp(Test,Then) ;
                                    AST = ifExp(Test,Then,Else)).

% WHILE: while exp do exp
factor(Input,whileExp(Test,Body), Rest) :-
                   currentToken(Input, while, In1),
                   exp(In1,Test,In2),
                   currentToken(In2,  do, In3),
                   exp(In3,Body, Rest).

% FOR: for id := exp to exp do exp WARNING: DEFINITION NOT COMPLETE
factor(Input,forExp(VarDec,High,Body), Rest) :-
           currentToken(Input, for, Id-C),
                currentToken(Id-C,id,In1), idExp(Id,In1,LV,In2),
                currentToken(In2,assignOp,In3),
                exp(In3,EXP,In4), VarDec=assignExp(LV,EXP),
           currentToken(In4,    to, In5), exp(In5,High, In6),
           currentToken(In6,    do, In7), exp(In7,Body,Rest).


% LET: let decs in expseq end
factor(Input,AST, Rest) :- currentToken(Input, let, In1),
                   decs(In1,Decs,In2),
                   currentToken(In2,  in, In3),
                   expSeq(In3,ExpSeq,In4),
                   currentToken(In4, end, Rest), AST=letExp(Decs,ExpSeq).

% break:
factor(Input,breakExp,Rest) :- currentToken(Input, break, Rest).

% - EXP
factor(Input,AST,Rest):- currentToken(Input,minus,Next), exp(Next,East,Rest),
                         AST = opExp(intExp(0),minus,East).
```

```
/*
exp:
op :- + | - | * | / | = | < | > | <= | >= | & | '|'
*/

% ELSE

else(Input,AST,Rest ):- currentToken(Input, else, Next),
                        exp(Next,AST,Rest).
else(Input,empty,Input).

% LVALUE: lvalue :- id lv
% LV: lv :- ( . id | [ exp ] ) lv.
%     lv.
% exp -> | lvalue
%        | lvalue :- ex
%        | id ( [ exp { , exp }] )
%        | type-id { id = exp { , id = exp } }
%        | type-id [ exp ] of exp

% WARNING: THIS IS INCOMPLETE
idExp(Id,Next,AST, Rest):- ctSpelling(Id,Spelling),
                           fraOrAv(Id,Next,AST,Rest).
% WARNING: THIS IS UNDER DEVELOPMENT
fraOrAv(Id,Input,AST,Rest):- ctSpelling(Id,Spelling),write_ln(id-Spelling),
                      (currentToken(Input,lbracket,In1) -> % id[exp]
                               (exp(In1,Exp,In2),
                                 currentToken(In2,rbracket,In3),
                                 ofOrAv(Spelling,Exp,In3,Var,In4),
                                 assignOp(Var,In4,AST,Rest)
                               );
/*                     (currentToken(Input,lparen,NxT) ->   % id(Args)
                               (expList(NxT,Args,NT1),
                                currentToken(NT1,rparen,Rest),
                                AST=callExp(Spelling,Args)
                               ); % WARNING: FIELDS IS INCOMPLETE
                      (currentToken(Input,lbrace,NxT) -> % id{Fields}
                               (expList(NxT,Args,NT1),
                                currentToken(NT1,rbrace,Rest),
                                AST=recordExp(Spelling,Args)
                               );
*/
                      (Rest=Input, AST = simpleVar(Spelling))
%                      )))
).

ofOrAv(Spelling,Size,Input,AST,Rest):- currentToken(Input,of,In1),!,
                            exp(In1,Value,Rest),
                            AST=arrayExp(Spelling,Size,Value).

ofOrAv(Spelling,Index,Input,AST,Rest):- av(Input,AV,Rest),
                           AST=subsriptVar(var(Spellling,AV),Index).

av(Input,AST,Rest):- (currentToken(Input,period,Id-C) -> % .id av
                               (currentToken(Id-C,id,In1),
                                ctSpelling(Id,Sp),
                                av(In1,AV,Rest),
                                AST=fieldVar(SV,Sp)
                                );
                      (currentToken(Next,assignOp,NxT) ->   % := exp
                               (expList(NxT,Args,NT1),
                                currentToken(NT1,rparen,Rest),
                                AST=callExp(SV,Args)
                               );
                      (currentToken(Next,lbracket,NxT) -> % exp ] av
                               (expList(NxT,Args,NT1),
```

```
                                currentToken(NT1,rbracket,Rest),
                                AST=callExp(SV,Args)
                               );
                      (Rest=Next, AST = null)
                      ))).

assignOp(Var,Input,AST,Rest):-currentToken(Input,assignOp,NT),!,
                          exp(NT,Exp,Rest),
                          AST=assignExp(Var,Exp).
assignOp(Var,Input,Var,Input).

% EXPSEQ: expSeq :- '(' exp, moreExp ')'.
% expList: expList :- exp ','  moreExp | empty
% expSeq.
% moreExp :- ';' exp, moreExp
% moreExp.
% moreExpL :- ',' exp, moreExpL
% moreExpL.

expList(Input,expList(Head,Tail), Rest):-
        exp(Input,Head,Next),
        mExpList(Next,Tail,Rest).
expList(Input, null, Input). % Empty List

mExpList(Input, expList(Head,Tail), Rest):-
          currentToken(Input,comma,Next),
          exp(Next,Head,Last), mExpList(Last,Tail,Rest).
mExpList( Input, null, Input).% End of expresssion sequence

expSeq(Input,seqExp(expList(Head,Tail)), Rest):- exp(Input,Head,Next),
                                    mExpSeq(Next, Tail,Rest).
expSeq(Input, null, Input). % Empty

mExpSeq(Input, expList(Head,Tail), Rest):-
          currentToken(Input,semicolon,Next),
          exp(Next,Head,Last), mExpSeq(Last,Tail,Rest).
mExpSeq( Input, null, Input).% End of expresssion sequence


%------------ DECLARATIONS ---------------------------------------------
% decs :- { dec }.
decs(Input,null,Input):- !. % WARNING: NOT IMPLEMENTED
decs(Input,AST,Rest) :- dec(Input,AST,IM), mDec(IM,AST,Rest).
decs(Input,AST,Input).

mDec(Input,AST,Rest) :- dec(Input,AST,IM), mDec(IM,AST,Rest).
mDec(Input,AST,Input).

dec(I,R) :- tydec(I,R).
dec(I,R) :- vardec(I,R).
dec(I,R) :- fundec(I,R).

tydec(I,I).
vardec(I,AST,R) :- currentToken(I,var,NT).
fundec(I,I).


% Data types

tydec :- true.    %type id = ty.
ty :- true.       %id | { tyfields } | array of id
tyfields :- true. %[ id : type-id { , id : type-id } ]
type-id :-  true. %int | string | id

% Variables

vardec :- true.   %var id [ : type-id ] := exp.
```

```
% Functions

fundec :- true.   %function id(tyfields) [ : type-id ] = exp.

% Operations op/2 token, ast
%  token,    ast
op(plus,     plus).
op(minus,    minus).
op(asterisk, mul).
op(divide,   div).
op(equal,    eq).
op(notEq,    ne).
op(less,     lt).
op(ltEq,     le).
op(greater,  gt).
op(gtEq,     ge).
```

```
:- module(translate,[translate/3]).
rc :- consult(translate).

%%%%%%%%%%%%%%%%%%%% translate/3 - AST,Env,IR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%---------- Variables ------------------------------------------------------

translate(simpleVar(Symbol),     Env, mem(Symbol)).
translate(fieldVar(Var,Symbol),  Env, fieldVar(Var,Symbol)).
translate(subscriptVar(Var,Exp), Env, subscriptVar(Var,Exp)).

%---------- Expressions ----------------------------------------------------

translate(varExp(Var), Env, mem(Var)).
translate(nilExp, Env, nil).
translate(intExp(Int), Env, const(Int)).
translate(stringExp(String), Env, stringExp(String)).
translate(callExp(FN,Args), Env, Code):-
      translate(Args,Env,AC), Code=call(FN,AC).
translate(opExp(Left,OP,Right), Env, binOp(OP,LE,RE)) :-
       translate(Left,Env,LE), translate(Right,Env,RE).

translate(recordExp(Name,FieldExpList), Env, const(V)).

translate(seqExp(null), Env, null).
translate(seqExp(expList(Head,Tail)), Env, TL):-
              translate(Head,       Env,TH),
              translate(seqExp(Tail),Env,TT), TL = seq(TH,TT).

% assignExp
translate(assignExp(V,E), Env, move(TV,TE)):- translate(V,Env,TV),
                                              translate(E,Env,TE).

% ifExp
translate(ifExp(C,Then), Env, cJump(C,V)).
translate(ifExp(C,Then,Else), Env, IC):-
         translate(C,Env,binOp(Op,L,R)),
         translate(Then,Env,TThen),
         translate(Else,Env,TElse),
         IC = seq(cjump(Op,L,R,name(true),name(false)),
              seq(label(true), seq(TThen,
              seq(jump(1,name(end)),
              seq(label(false),
              seq(TElse,label(end)))))))).

% whileExp
translate(whileExp(E,E), Env, const(V)).

% forExp WARNING: THIS IS INCORRECT
translate(forExp(assignExp(Var,Start),Limit,Body), Env, TE):-
                 translate(letExp(VDec,whileExp(Limit,Body)),Env,TE).

% break
translate(breakExp, Env, jump(name(n),n)).

% letExp - WARNING: does not handle declarations
translate(letExp(Decs,E), Env, IR):- translate(E,Env,IR).

% arrayExp
translate(arrayExp(S,E,E), Env, const(V)).

%---------- Declarations ---------------------------------------------------

translate(functionDec(N,Params,Result,Body,Next), Env, const(V)).
translate(varDec(N,Type,Initial), Env, const(V)).

%---------- Types ----------------------------------------------------------
```

```
translate(nameTy(Name),Env,nameTy(Name)).
translate(recordTy(Fields),Env,recordTy(Fields)).
translate(arrayTy(Type),Env,arrayTy(Type)).

%---------- Miscellaneous -------------------------------------------------

translate(decList(Head,Tail),Env,decList(Head,Tail)).
translate(null,Env,null).
translate(expList(Head,Tail),Env,TEL):- translate(Head,Env,TH),
                                        translate(Tail,Env,TT),
                                        TEL = seq(TH,TT).

translate(fieldExpList(Symbol,Exp,FieldExpList), Env,
          fieldExpList(Symbol,Exp,FieldExpList)).
translate(fieldList(Symbol,Symbol,FielidList), Env,
          fieldList(Symbol,Symbol,FielidList)).

% DEFAULT
translate(A,E,A):- write_ln(default-A).

/*****************************************************************************/
% ast,ir
op(eq,eq).
op(ne,ne).
op(lt,lt).
op(le,le).
op(gt,gt).
op(ge,ge).
```

```
:- module(codegen,[codegen/2]).
rc :- consult(codegen).

%%%%%%%%%%%%%%%%%%%% codegen/2 - IR,Code %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

codegen(const(Value),[pushC(Value)]).

codegen(name(Label),[name(Label)]).

codegen(temp(Temp),[temp(Temp)]).

codegen(binOp(Op,L,R),Code):- codegen(L,LC), codegen(R,RC),
                              opcode(Op,OP),
                              append(RC,[OP],RHS), append(LC,RHS,Code).

codegen(mem(Addr),[load(Addr)]).

codegen(call(Function,Args),[call(Function,Args)]).

codegen(eSeq(Stmt,Exp),[eSeq(Stmt,Exp)]).

codegen(assignExp(V,E),Code):- codegen(E,EC), append(EC,[store(V)],Code).

codegen(move(mem(Addr),Src),Code):- codegen(Src,Scode),
                                    append(Scode,[store(Addr)],Code).

codegen(move(Dst,Src),Code):- codegen(Src,Scode),
                              append(Scode,[store(Dst)],Code).

codegen(exp(Exp),[exp(Exp)]).

codegen(jump(Exp,Labels),[jump(Exp,Labels)]).

codegen(cjump(RelOp,Lhs,Rhs,True,False),Code):-
           codegen(Lhs,LC), codegen(Rhs,RC),
           append(RC,[RelOp,jmpfalse(False)],RJ), append(LC,RJ,Code).

codegen(seq(Stmt1,Stmt2),Code):- codegen(Stmt1,C1),
                                 codegen(Stmt2,C2),
                                 append(C1,C2,Code).

codegen(label(Label),[label(Label)]).

codegen(expList(Head,Tail),Code):-codegen(Head,HC),
                                   codegen(Tail,TC),
                                   append(HC,TC,Code).

codegen(stmtList(Head,Tail),Code):-codegen(Head,HC),
                                    codegen(Tail,TC),
                                    append(HC,TC,Code).

codegen(null,[]).
codegen(A,A).

%----- OP      OpCode

opcode(plus,   add).
opcode(minus,  sub).
opcode(mul,    mul).
opcode(div,    div).
opcode(eq,     eq).
opcode(ne,     ne).
opcode(lt,     lt).
opcode(le,     le).
opcode(gt,     gt).
opcode(ge,     ge).
opcode(or,     or).
opcode(and,    and).
```

```
opcode(lshift, lshift).
opcode(rshift, rshift).
opcode(arshift,arshift).
opcode(xor,    xor).
opcode(ult,    ult).
opcode(ule,    ule).
opcode(ugt,    ugt).
opcode(uge,    uge).

/*****************************************************************************/
```

Overview of Logic

Logic is a science of truths - Quine 1950
Logic is a science of deduction - Hacking 1979

Connections:

- Related to:
- Prerequisites:
- Requisite for: Classical, Multivalued, Modal, Constructive

Logic is based on deduction, a method of exact inference with the advantage that its conclusions are exact -there is no possibility of mistake if the rules are followed exactly. Deduction requires that information be complete, precise, and consistent.

The usual use of a logic consists in translating information from some domain into the language of the logic where reasoning can be conducted without thinking about the meaning of each formula. The resulting inferences are then translated back into the domain of interest.

> A *logic* is a language for describing and reasoning about some *domain of discourse (structure)*. The language is a *syntactic* representation of a domain and its properties of interest (the domain is called a *structure, interpretation,* or *model* of the formal logical system).

We begin with a definition of formal systems of which logic is an example.

Formal Systems

First, an example:

- The natural numbers: 0, 1, 2, ..., are the domain of discourse.
- The Peano axioms for the natural numbers are a language for describing and reasoning about the natural numbers.
    - A syntactical representation of the natural numbers using the BNF: N ::= 0 | s(N)
    - A definition of addition
        - 0 + s(n) = s(n)
        - s(m) + s(n) = s(m + s(n))
    - etc.

Figure :

		Source
Syntactic methods:	Axiomatic method	Aristotle, Hilbert
	Sequent method	Gehard Gentzen
Semantic methods:	Analytic tableaux	Beth, Hintikka, Kripke

A formal system defined syntactically in this manner has no notion of meaning or semantics. It is can be viewed as a meaningless game with symbols.

Examples of formal systems include mathematics and programming languages.

The language defined in a formal system is designed for reasoning about some domain of discourse. A *structure* is a domain of discourse for which a formal system provides the language of discourse. An *interpretation*, or *model* assigns meaning (semantics) to the formulas of the language by associating each formula with some aspect of the structure.

Figure 0: **Syntax and Semantics**

Language	**Semantic Function**	**Model**
L	$->_s$	***M***
l in ***L***	s(*l*) = *m*	*m* in ***M***

A given formal system may be used to reason about more than one structure. For example, geometry without the parallel postulate may be used to reason about both Euclidean and non-Euclidean geometry's -theorems proved about one system are true in the other. The separation of the language of discourse from the object under discussion is an important aspect of modern mathematics/axiom systems (axioms describe the properties of multiple worlds). Since the language and its axioms often become the subject of study rather than the structure (or domain of discourse), the structure is called an interpretation or model of the axioms.

Figure 1: **Syntax**

Symbols	- the alphabet of the logic
Formulas	P - a set of atomic formulas F - the formulas of the logic (atomic and compound)
Axioms	- a set of formulas said to be true

Inference rules - the way formulas are derived from other formulas.
- they have the form:

$$\frac{H_0,...,H_{n-1}}{C}$$

- where the H_i are called *hypotheses* or *premises* and C the *conclusion.*

Figure 2: **Semantics**

Structure & Interpretation
M = <S, V> called a model where S is a setV in F --> S,
The set of valuation functions ***M*** |= p iff V(p) in S, for p an atomic formula
M |= F iff by some rule F is valid

An *interpretation* is a mapping between the formulas of a logic and a structure.

I: Formulas --> Structure

Exercises

1. Construct a formal system for arithmetic.
2. Define an interpretation for arithmetic
3. Construct a formal system for simple algebraic equations.
4. Define an interpretation for algebraic equations

The study of formal systems naturally separates into two areas, proof theory which is concerned with the language and model theory which is concerned with the relation between the language and the domain of discourse.

In proof theory the concerned is with the following terms and concepts:

Axiom
An axiom is a formula (ultimately a statement that is true for some structure). The selection of axioms should be those that facilitate proofs.
Proof
A *proof* is a sequence of formulas each of which is an *axiom*, or may be inferred by an inference rule, from formulas appearing earlier in the sequence.
Theorem
A *theorem* is the last formula in a proof. If F is such a formula, we write |-F to say that it is a theorem.
Decision method
A *decision method* is a method for deciding whether or not a formula is a theorem.
Consistent and **Inconsistent**
A language is called *consistent* if a formula can not be proved to be both a

theorem and a not a theorem. A language is called *inconsistent* if a formula can be proved to be both a theorem and a not a theorem.

In model theory the concerned is with the following terms and concepts:

Satisfiable
A formula F that is true for some structure M, M |= F is said to be *satisfiable*.
Valid
A formula F that is true in all structures is said to be *valid*, |= F (i.e. if ***M*** |= F for all ***M***, then F is valid).
Tautology
A formula that is valid is called a *tautology*. The formula *A or not A* is a tautology in classical logic.
Contradiction
A formula that is not satisfiable in any structure is called a *contradiction* (i.e., if not ***M*** |= F for all ***M***, then F is a contradiction). The formula *A and not A* is a contradiction in classical logic.
Sound
The inference rules in a language are said to be *sound* iff every theorem derived by an inference rule is valid (i.e. for each formula F, if |-F then |=F).
Complete
A language is *complete* iff every valid formula is a theorem (i.e., for each formula F, if |=F then |-F).
Consistent
A theory (a language) is *consistent* if it has a model.

Examples

	Language	**Structure**
Linear languages	Regular expressions	Sets of strings
Arithmetic	Arithmetic expressions	Numbers & functions
Logic	Logical expressions	Truth values

Logic as a formal system

Logic is concerned with the relationship between language (syntax), reasoning (deduction and computation), and meaning (semantics).

Syntax (Language *L*)

The syntax of the language ***L*** is described using the symbols of the language ***L***, standard mathematical notation, and meta symbols from natural languages.

Figure 3: **Language - *L* = (C, V, P, F)**

C = { c_i | i = 0, 1, ... } A set of constants k_i in C
V = { x_i | i = 0, 1, ... } A set of variables; x in V
P = { p_0^0, p_0^1, ... , p_1^0, p_1^1, ... , p_2^0, p_2^1, ... , ...} A set of predicate symbols.
At = { $p_i^j k_0...k_{j-1}$ | p_i^j in P and $k_0,...,k_{j-1}$ in C } A set of **atomic formulas**; f in At.
F ::= f | ¬F | /\FF | \/FF | ->FF | <->FF -- *propositional formulas*
| **[]**F | **<>**F | ... -- ***modal*** *formulas*
| /\x.[F]^k_x | \/x.[F]^k_x-- *first-order formulas*
where
Textual substitution, **[F]$^{\mathbf{k}}_{\mathbf{x}}$** , is part of the meta language.
For every formula, A and B, predicate symbol, p_i^{j-1}, symbols x and y, and c,
the **substitution** of c for x in A, **[A]$^{\mathbf{x}}_{\mathbf{c}}$** is defined as follows:

- $[p_i^{j-1}v_0,...,v_j]^x_c = p_i^{j-1}[v_0]^x_c, ... , [v_j]^x_c$ where for distinct variables x and y
  - $v_i = x$; $[x]^x_c = c$
  - $v_i = y$; $[y]^x_c = y$
  - $v_i = c_k$; $[y]^x_c = c_k$
- $[¬A]^x_c = ¬[A]^x_c$
- $[/\AB]^x_c = /\[A]^x_c[B]^x_c$
- ...
- $[/\xA]^x_c = /\x.A$
- $[/\y.A]^x_c = /\y.[A]^x_c$
- ...

There are several alternate notations for textual substitution.
There are several alternatives expressions of syntax.

The p_i^0 are propositions, the p_i^1 are properties, the p_i^2 are binary relations, and the p_i^n (n>1) are n-ary relations. A logic is 0-order if it does not contain quantifiers, 1st-order if quantification is permitted over variables, and 2nd order if quantification is permitted over predicates.

Simple sentences in natural languages are easily represented in symbolic form as shown in the following figure.

Figure 4: **Examples**

	Natural language form	Symbolic form
Proposition	The sun is shining	P

Property	Clarence is tall.	tall(Clarence)
Binary Relation	John loves Mary.	loves(John, Mary)
Complex example	There is a man that likes pizza.	\/x.[man(x) /\ likes(x,pizza)]

Definitions: A **sentence** (or a **closed** formula) is a formula A such that for every variable x and every constant c, $[A]^x_c$ = A. This means that in a closed formula A, all of the variables in A are quantified. In the formula /\x.A, A is the **scope** of the quantifier and x is **not free** in /\x.A. There are alternative approaches to defining free and bound variables.

A **theory** is a set of sentences and rules of deduction.

Semantics

In classical logic there are precisely two **truth values**: **true** and **false**. A **valuation function** *v* maps atomic formulas to truth values. Fuzzy and probabilistic logics are examples of continuous valued logics. The following figure suggests some valuation functions for various types of logics.

Figure 5: **Valuation functions for classical and multivalued logics**

v : At -> {0, 1}	*v* is boolean valued as in classical logic
v : At -> {_|_, 0, 1}	*v* identifies undefined formulas such as - y > 1/x.
v : At -> {0, ... , n}	*v* is a multivalued logic with a range of values.
v : At -> [0, 1]	*v* is infinite valued and is suitable for use in a probabilistic, fuzzy, and other continuous logics.

Additional information on multivalued logics is available.

Valuation functions may be required to be total predicates and often are extended to all formulas as suggested in the following figure.

Figure 6: ***v*|= A iff *v*(A) = 1**

v |= p	*v*(p) where p is in At
v |= ¬A	*v*(¬A) = 1 - *v*(A)
v |= ->AB	*v*(->AB) = max(*v*(¬A), *v*(B))
v |= /\x.A	iff *v* |= $[A]^x_c$ for all c in C
v |= ¬/\x.A	iff *v* |= $[¬A]^x_c$ for some c in C

A formula is said to be **satisfiable** if there is a validation function which make it true. A formula is said to be a **contradiction** if there is a validation function which makes it false. A formula A is said to be **valid** (a **tautology**), |=A, if it is true for all valuations *v*. For propositional formulas, truth tables are an accepted method to determine whether a formula is a tautology (valid), satisfiable, or a contradiction. Boolean semantics are based on the coherence theory of truth.

Exercises

1. Show that *v*(A/\B) = min(*v*(A), *v*(B)).
2. Show that
    1. all the propositional operators can be defined in terms of ¬, and /\,
    2. all the propositional operators can be defined in terms of ¬, and \/,
    3. all the propositional operators can be defined in terms of ¬, and ->.

Structure *S* (Alfred Tarski)

Figure 7: **Relational Structure** ***w***= <C,R>

A set of constants	C = { c_i | i = 0, 1, ... }
A set of relations	R = { R_0^0, R_0^1, ... , R_1^0, p_1^1, ... , R_2^0, R_2^1, ... , ...} where each R_i^j a subset of A^{j+1}

The constants in a language can be treated as names of constants in a structure and predicate symbols as names of relations. A valuation function says whether or not an atomic formula maps to an appropriate tuple in the structure. As with valuation functions, a structure can be extended to formulas and is said to **model** (or be an **interpretation** of) a formula if the formula is true in the structure.

Figure 8: **Model -** ***M*** |=F
where ***M***= (***w***, *v*)

M|= p	iff *v*(p) in ***w***
M|= ¬p	iff *v*(p) *not* in ***w***
M|= ->AB	iff ***M***|= ¬A or ***M*** |= B
M|= ¬->AB	iff ***M***|= A and ***M*** |= ¬B
M|= /\x.F	iff ***M***|= $[F]^x_c$ for all c in C
M|= ¬\/x.F	iff ***M***|= $[¬F]^x_c$ for some c in C

It is clear that for every interpretation there is a corresponding boolean valuation function. Likewise, for every boolean valuation function there is a corresponding interpretation.

Tarskian semantics are based on the correspondence theory of truth.

Definitions

- A sentence S of L is **valid**, |= S, if it is true in all structures for ***M***.
- A sentence S of L is a **logical consequence** of a set of sentences Ss of L (Ss |= S), if S is true in every structure in which all of the members of Ss are true.

A set of sentences Ss, is **satisfiable** if there is a structure ***A*** in which all of the members of Ss are true. Such a structure is called a **model** of Ss. If Ss has no model, it is **unsatisfiable**.

Multiple World Semantics (Saul Kripke)

Multiple structures can be used to model *modal* formulas.

Figure 9: **Multiple world structure*U*= (*W*, *A*)**

A set of relational structures	***W*** = {***w*** | ***w*** is a relational structure}
An accessibility relation	***A*** a subset of ***W***×***W***

A model ***M*** = (***W***, ***A***, ***w***, *v*) is a Kripke structure. A Kripke structure where |**W**| = 1 corresponds to traditional logics.
A reachability relation that is symmetric (**Axy** implies **Ayx**) implies that the graph is nondirectional.
A reachability relation that is transitive (**Axy** and **Ayz** implies **Axz**) can be used to model temporal phenomenon.
A reachability relation that is reflexive (**Axx**), symmetric, and transitive, can be used to reason about finite state systems.

There are many modal logics. The table that follows illustrates the approach to semantics for modal logics.

Figure 10: **Model - *M*** |=F
where ***M***= (***U***, ***w***, *v*)

M|= p	iff *v*(p) in ***w***
M|= ¬p	iff *v*(p) *not* in ***w***
M|= ->AB	iff ***M***|= ¬A or ***M*** |= B
M|= ¬->AB	iff ***M***|= A and ***M*** |= ¬B
M|= **[]**A	iff ***M'***|= A for all ***u*** such that ***Awu*** and ***M'*** = (***U***, ***u***, *v*)
M|= **<>**A	iff ***M'***|= A for some ***u*** such that ***Awu*** and ***M'*** = (***U***, ***u***, *v*)
M|= /\x.F	iff ***M***|= $[F]^{x}_{c}$ for all c in C

$\boldsymbol{M}\models \neg\forall x.F$	iff $\boldsymbol{M}\models [\neg F]^{x}_{c}$ for some c in C

A formula F is **valid** (a **tautology**), |= F, iff for all **w** in **W**, ***M***|= F i.e., F *is true in all possible worlds*.

A formula F is said to be **valid** (|=F) iff it is valid in all models ***M*** (***M*** |= F for all ***M***). A valid formula is called a **tautology**. Predicate Logic (or Predicate Calculus or First-Order Logic) is a generalization of Propositional Logic. Generalization requires the introduction of variables.

Linear time temporal logic is an example of a logic that uses multiple world semantics. Each time increment is represented by a world. The accessibility relation is reflexive and transitive but not symmetric as we assume that time does not run backwards. For the formula **[]**A, A holds in the current world and in all future worlds and for the formula **<>**A, A holds in either the current world or some future world.

Proof Theory (Computation) - Ss |- A

A **theory** is a set of sentences and rules of deduction/inference. A theory is **monotonic** if the truth of a proposition does not change when new axioms are added to the system.

Definition:

- A **proof** of A from a set of formulas Ss, Ss |- A, is a contradictory tableau from [¬A | Ss].
- A set of sentences Ss is **inconsistent** iff *there is a proof of* A *and* ¬A *from* Ss for some formula A. Equivalently, A set of sentences is **inconsistent** iff it contains all sentences.
- A set of sentences Ss is **consistent** iff it is not inconsistent. Equivalently, a set of sentences is **consistent** iff it has a model.
- A *proof method* is **complete** iff *every valid formula has a proof*.
- A proof method is **incomplete** iff there is a valid formula for which there can be no proof.

An inference rule first suggests the idea of **forwards proof** (**bottom-up proof**): working from theorems to theorems. In normal mathematics we start with a desired conclusion, or *goal* and work from goals to subgoals - a **backwards proof**, **goal directed proof**, or **top-down proof**.

There are a variety of styles of proofs. *Traditional mathematical proofs* are done in a stilted form of ordinary prose which often contain gaps which the reader is expected to fill in. More formal proofs are done in the *Hilbert style* with each step in the proof justified with a reason why it is true. *Analytical style* proofs pick statements apart until a contradiction occurs. *Natural deduction* style proofs tend to pick apart statements and reassemble them into new statements. A *refutation style* proof tries to refute the statement to be proved. Refutation style proofs often construct a model of the formula as a side-effect of the proof process. This aspect is particularly useful for propositional logics which often have the finite model property.

Hilbert Style Proofs

The Hilbert style of proofs is often used in teaching geometry in high school. It consists of the theorem to be proved followed by a sequence of line each of which contains a theorem and a reason why it is a theorem with the last line the theorem being proved. Subproofs may be indented.

Figure 11: **Hilbert Style Proof**

Theorem to be proved:

Steps **Reasons**

Reasons -assumption, instance of a theorem, inference rule.

Proof Formats

The point of a proof is to provide convincing evidence of the correctness of some statement. The following proof formats make clear the intent of the proof as it is read from beginning to end.

Figure : **Proof Formats**

Natural Deduction	**Hilbert Style Proof Format**	
P, P->Q ——— Q	Q 1 P 2 P -> Q	by **Modus Ponens** ... *explanation* ... *explanation*
A |- B ——— A -> B	A->B 1 ¬B ... i ¬A	by **Contrapositive** Assumption ... ... *explanation*
P, Q |- R ——— P/\Q->R	P/\Q->R 1 P 2 Q ... *i* R	by **Deduction** Assumption Assumption ... ... *explanation*
¬P |- Q/\¬Q ——— P	P 1 ¬P ... *i* Q/\¬Q	by **Contradiction** Assumption ... ... *explanation*
P |- Q/\¬Q ——— ¬P	¬P 1 P ... *i* Q/\¬Q	by **Contradiction** Assumption ... ... *explanation*

P\/Q, P->R, Q->R ——— R	R 1 P\/Q 2 P->R 3 Q->R	by **Case analysis** ... *explanation* ... *explanation* ... *explanation*
P->Q, Q->P ——— P<->Q	P<->Q 1 P->Q 2 Q->P	By **Mutual implication** ... *explanation* ... *explanation*
P(0), P(n) -> P(n+1) ——— /\n.P	/\n.P 1 P(0) 2 P(n) ... *i* P(n+1)	By **Induction** Base step ... *explanation* Assumption (Inductive hypothesis) ... ... *explanation*

Axiom Systems and Theories

Figure 12: **Axioms and rules of inference**

Logical Axioms	All closed tautologies	
Rules of inference	A, A -> B ——— B	Modus Ponens
	gamma(c) -> X ——— gamma -> X	
	delta(d) -> X ——— elta -> X	provided d does not occur in delta nor in X
Non logical Axioms		

natural deduction and sequent systems

Issues in Model Theory and Proof Theory

- *Consistency*: Is there a structure that models the axioms? i.e., Are the axioms valid (S |= A)?
- *Soundness*: Do the proof rules lead to valid theorems i. e., does |- F imply |= F?
- Given a set of formulas, how large a structure is required to satisfy the set of formulas?
- *Completeness*: Are the set of formulas and rules of inference sufficient to insure that every valid formula is a theorem i. e., does |= F imply |- F?
- *Decideability*: Is the theory decidable i. e., for every formula F, is either F or ¬F a theorem? Alternatively, for formula F, is either |- F or |- ¬F?

References

Fagin, Halpern, & Vardi
What is an inference rule? *Journal of Symbolic Logic* **57**:3, 1992, pp. 1018-1045.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Classical Two-valued Monotonic Logic

Connections:

- Related to:
- Prerequisites: Formal Systems
- Requisite for: Multivalued, Modal

Monotonic systems: the truth of a proposition does not change when new axioms are added to the system.

Modern mathematics/axiom systems -- axioms describe the properties of multiple worlds and since the axioms are the subject of study rather than the worlds, the worlds are called an interpretation or model of the axioms.

Supplementary material

Predicate Logic

Syntax - (Language *L*)

The symbols of predicate logic are the following:

1. Infinite list of **predicate** symbols: p^i_j j = 0, 1, ...; i = 0, 1, (note that p_j^0, j = 0, 1, ... correspond to the propositional letters p_i, i = 0, 1, ... of propositional logic);
2. the possibly infinite set of symbols c_i, i = 0, 1, ... called constants;
3. the infinite set of symbols x_i, i = 0, 1, ... called variables;
4. the symbols ¬, ->, /\ ., **negation**, **implication**, and **forall** respectively (alternatively, negation, implication and the universal quantifier.

The set of formulas of predicate logic are defined by the following rules:

1. For any predicate symbol p_i^j and variables or constants $v_0,...,v_j$, $p_i^{j-1}(v_0,...,v_j)$ is an **atomic formula**;
2. if A and B are formulas then so are: ¬A, -> AB
3. If A is a formula and x a variable, then /\x.A is a formula.

For every formula, A, variable x, and constant c, the expression $[A]^x_c$ is defined as follows:

- $[p_i^{j-1}(v_0,...,v_j)]^x_c = p_i^{j-1}([v_0]^x_c, ... , [v_j]^x_c)$ where for distinct variables x and y
  - $v_i = x$; $[x]^x_c = c$
  - $v_i = y$; $[y]^x_c = y$
  - $v_i = c_k$; $[y]^x_c = c_k$
- $[->AB]^x_c = ->[A]^x_c[B]^x_c$
- $[\neg A]^x_c = \neg[A]^x_c$
- $[/\backslash x.A]^x_c = /\backslash x.A$
- $[/\backslash y.A]^x_c = /\backslash y.[A]^x_c$

Figure N.6: **Predicate Logic - The Syntax**

Symbols and Formulas:

$C = \{ c_i | i = 0, 1, ... \}$ A set of constants
$V = \{ x_i | i = 0, 1, ... \}$ A set of variables; x in V
$p = \{ p_0^0, p_0^1, ... , p_1^0, p_1^1, ... , p_2^0, p_2^1, ... , ...\}$ a set of predicate symbols
$At = p_i^{j-1}(k_0,...,k_j)$ where p_i^j in P, $k_0,...,k_j$ in V**u**C; a set of atomic formulas
F ::= A | ¬F | ->FF | /\x.F; a set of formulas

Definitions: A **closed** formula or **sentence** is a formula A such that for every variable x and constant c, $[A]^x_c = A$. This means that in a closed formula A, all of the variables in A are quantified. In the formula /\x.A, A is the **scope** of the quantifier and x is **not free** in /\x.A.

The compound formulas of (with the exception of the negation of an atomic formula) are classified as of type **alpha** with subformulas $\mathbf{alpha}_1$ and $\mathbf{alpha}_2$, type **beta** with subformulas $\mathbf{beta}_1$ and $\mathbf{beta}_2$, type **gamma**, or of type **delta.** The classification scheme is summarized in Figure N.6.

Figure N.6: **Alpha, Beta, Gamma and Delta Formulas**

alpha	**$\mathbf{alpha}_1$**	**$\mathbf{alpha}_2$**
/\AB	A	B
beta	**$\mathbf{beta}_1$**	**$\mathbf{beta}_2$**
\/AB	A	B

gamma	**gamma(c)**
/\x.A	$[A]^x_c$
delta	**delta(d)**
\/x.A	$\neg\ [A]^x_d$

See material on the analytic tableaux method for more details.

Model Theory (Semantics) - (Structure *A*; *A* |= S)

There are precisely two **truth values**: **false** and **true**. Formulas are classified as true or false. These values are the domain of discourse and the interpretation is provided by a **valuation** function v from formulas to the set of boolean values. The function is total on predicates. A type **alpha** formula is classified as true iff both of its subformulas are true. A type **beta** formula classified as false iff one of its subformulas is true. A type **gamma** formula holds iff its subformula holds for each constant. A type **delta** formula holds iff its subformula holds for some constant. Figure N.4 summarizes these concepts.

Figure N.7: **Predicate Logic - The Semantics**

Structure and Interpretation.

M= <B, V> where
B = {**t**, **f**},
V an element of At ->B, The set of valuation functions.
k in C

M **satisfies** a formula F (***M*** |= F) iff the following properties hold:

M |= $p_i^j(k_0,...,k_{j-1})$	iff	$V(p_i^j(k_0,...,k_{j-1}))$ = **t**
M |= $\neg p_i^j(k_0,...,k_{j-1})$	iff	$V(p_i^j(k_0,...,k_{j-1}))$ = **f**
M |= **alpha**	iff	***M*** |= $\mathbf{alpha_1}$ and ***M*** |= $\mathbf{alpha_2}$
M |= **beta**	iff	***M*** |= $\mathbf{beta_1}$ and ***M*** |= $\mathbf{beta_2}$
M |= **gamma**	iff	***M*** |= **gamma(c)** for all c in C
M |= **delta**	iff	***M*** |= **delta(c)** for some c in C

Definitions

- A sentence S of L is **valid**, |= S, if it is true in all structures for ***M***.
- A sentence S of L is a **logical consequence** of a set of sentences Ss of L (Ss |= S), if S is true in every structure in which all of the members of Ss are true.
- A set of sentences Ss, is **satisfiable** if there is a structure ***A*** in which all of the members of Ss are true. Such a structure is called a **model** of Ss. If Ss has no model, it is **unsatisfiable**.

A formula F is said to be **valid** (|=F) iff it is valid in all models ***M*** (***M*** |= F for all ***M***).
A valid formula is called a **tautology**.
Truth tables are an accepted method to determine whether a formula is a tautology (valid), satisfiable, or a contradiction.
The valuation function approach is an alternative to truth tables. ...

Proof Theory - (Ss |- A)

Definition: By a **Hintikka set** we mean a set S such that the following conditions hold for every formula of type alpha, beta, gamma, and delta in S.

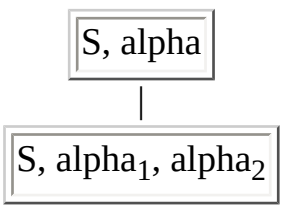
1. No *atomic* formula and its negation are both in S.
2. If alpha is in S, then both $alpha_1$ and $alpha_2$ are in S.
3. If beta is in S, then either $beta_1$ is in S or $beta_2$ is in S.
4. If gamma is in S, then for every c, gamma(c) is in S.
5. If delta is in S, then for some d, delta(d) is in S.

Lemma: (Hintikka's lemma for first-order logic) Every Hintikka set S is satisfiable.
Proof: A valuation function is easily constructed from the Hintikka set. The valuation function maps all atomic formula S to **t** and those not appearing in the set to **f**. The construction rules follow the rules for satisfiability. QED.

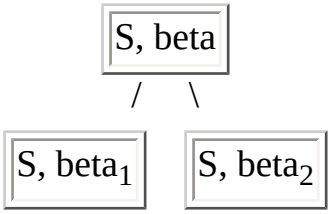
The **alpha**, **beta**, **gamma** and **delta** classification of formulas form the base for the method of proof using analytic tableaux. The method involves searching for contradictions among the formulas generated by application of the analytic properties.

By a **block tableau** for a finite set, **Fs**, of formulas, we mean a tree constructed by placing the set **Fs** at the root, and then continuing according to the following rules:

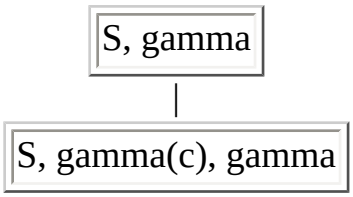
Rule A:



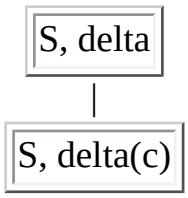
Rule B:



Rule C:



Rule D:



where d is new to the tree

Definition:

- A path in tableau is **closed/contradictory** if a block on the path contains a formula and its negation.
- A path in tableau is **open** if no block on the path contains a formula and its negation.
- A tableau is **contradictory** if every path is contradictory
- A **proof** of A from a set of formulas Ss, Ss |- A, is a contradictory tableau from [¬A | Ss].
- A set of sentences Ss is **inconsistent** iff there is a proof of A and ¬A from Ss for some formula A. Equivalently, A set of sentences is **inconsistent** iff it contains all sentences.
- A set of sentences Ss is **consistent** iff it is not inconsistent. Equivalently, a set of sentences is **consistent** iff it has a model.
- A proof method is **complete** iff every valid formula has a proof.
- A proof method is incomplete iff there is a valid formula for which there can be no proof.
- **Satisfiable, satisfies**
- **model, models**

Theorem 1. For any tableau, every open branch is a Hintikka set.
Proof: The tableau construction rules and the definition of open branch correspond to the rules describing a Hintikka set provided the rules are applied breadth first and the gamma rule is applied fairly to the gamma formula appearing on a branch. QED.

Theorem 2. For any tableau, the open branches are simultaneously satisfiable.
Proof: By the Hintikka lemma and theorem 1. QED.

Theorem 3. (Completeness Theorem for First Order Tableau - Godel). If X is valid, then X is provable. Indeed, if X is valid, then the systematic tableau for ¬X must close after finitely many steps.
Proof: Suppose X is valid. Construct a tableau for ¬X. If the tableau contains an open branch, then ¬X would be satisfiable contrary to the hypothesis. Thus X is provable. QED

Theorem 4. (Lowenheim's Theorem). If X is satisfiable at all, then X is satisfiable in a denumerable domain.
Proof:

Theorem (Soundness): If F is a theorem, then F is valid (If all branches of the tableau proof of A from Ss are closed then Ss |= A).
Proof:

Comment: This theorem shows that the method of proof preserves the truth of statements and may be restated as: If a formula is a theorem, then it is valid. The question then arises: do we have all the axioms and rules we need? The question is the converse of the soundness theorem: If a formula F is valid, then it is a theorem. The other direction is developed in the next theorem.

Theorem (Completeness I - Godel): For every sentence S and set of sentences Ss, the tableau method establishes either Ss |= S or constructs a model of {Ss, ~S}.
Proof:

Comment:

- If S is not a theorem, some branch may not close, i.e., the model may be infinite.
- The purpose of the completeness theorem is to show that every logical consequence of a set of nonlogical axioms can be proved from these nonlogical axioms by means of the logical axioms and rules.

Theorem (Completeness II): The set of sentences Ss is consistent iff it has a model.
Proof:

Comment:

Theorem (Skolem-Lowenheim): If a countable set of sentences Ss is satisfiable, (that is, it has a model), then it has a countable model.
Proof:

Theorem (Compactness): Let Ss be a set of sentences of predicate logic. Ss is satisfiable if and only if every finite subset of Ss is satisfiable.
Proof:

Theorem (Church): The predicate calculus is undecidable, i.e., there is no effective procedure for deciding if a given sentence is valid.
Proof:

Comment: The completeness theorem tells us that the tableau method will find a proof for each valid formula. However, Church's theorem tells us that if a formula is *not valid*, the tableau method may not terminate.

Comment: Incompleteness theorem

Exercises

1. Show that the following formulas are tautologies
    1. A\/¬ A
2. Show that the following formulas are contradictions
    1. A/\¬ A
3. Using truth tables, show that the following equalities hold:
    1. **¬ (A /\ B) = ¬ A \/ ¬B**
    2. **¬ (A \/ B) = ¬ A /\ ¬B**
    3. **A\/(B/\C) = A/\B \/ A/\C**
    4. **A/\(B\/C) = A/\B \/ A/\C**
    5. **A => B = ¬ A \/ B**
    6. **A <=> B = (A => B) /\ (B => A) = A /\ B \/ ¬A /\ ¬B**
4. Construct an alternate semantics for propositional logic by extending the valuation function to formulas.
5. Construct an alternate semantics for propositional logic based on the idea that the valuation function maps atomic formulas to elements in the model.
6. Show that a|b can be used to define all boolean functions
7. Show that all boolean functions can be defined in terms of ...
8. Show that not A.x not Px = E.x Px and not E.x not Px = A.x Px
9. Prenex normal form
10. Conjunctive normal form
11. Disjunctive normal form

Horn Clause Logic

Historical Perspectives and Further Reading

Exercises

1. Define all the logical connectives from --> and _|_
2. Let p|q be defined to mean that p and q are not both true. Define the other logical connectives in terms of this one.
3. Let p|q be defined to mean that p and q are both false. Define the other logical connectives in terms of this one.
4. Show that the following formulas are tautologies
    a. A \/ ¬ A
    b. A \/ (B /\ C) = (A \/ B) /\ (A \/ C)
    c. A /\ (B \/ C) = (A /\ B) \/ (A /\ C)
5. Determine whether or not the following inference rules are sound

   a. from A and A --> B infer B (modus ponens)
   b. from B and ¬A --> ¬B infer A
   c. from A and B --> A infer B
   d. from A and ¬B --> ¬A infer B
6. Show that natural deduction is complete.
7. Show that the method of proof trees is complete.
8. Show that the following are equivalent
   a. *C* is inconsistent (*C* contains _|_)
   b. *C* contains *all* propositions
   c. *C* contains some proposition, p, and its negation, ¬ p
9. Prove: if the initial configuration of consists of a block containing a single formula, the negation of a tautology, then the configuration reduces to a configuration where each block is a contradiction.
10. Prove that there is a proof of T from an axiomatization $\{A_0,...,A_n\}$ iff each block in the final configuration of a sequence of reductions beginning with the configuration $\{A_0,...,A_n, \neg T\}$ contains a contradiction.

- Atomic formulas are mapped to true iff they represent relations in some structure.
- Example (List theory)

  Axiom 0: list([])

  Axiom 1: A(X,L)[list(L) --> list([X|L])]

  Abbrev: [X_0|[...[X_n|[]...] = [X_0,...X_n]

  Axiom 2: A(L)[append([],L,L)]

  Axiom 3: A(X,L0,L1,L0L1)[append(L0,L1,L0L1) --> append([X|L0],L1,[X|L0L1])].

  Thm 0: E(L)[append([0,1],[a,b],L)].

  Thm 1: E(L)[append([0,1],L,[0,1,a,b])].

  Thm 2: E(L)[append([L,[a,b],[0,1,a,b])].

  Thm 4: E(L0,L1)[append([L0,L1,[0,1,a,b])].

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Modal Logics

Connections

- Related to:
- Prerequisites: Formal Systems, Classical
- Requisite for: Tableau rules, Temporal Logic

Modal logics are designed to express possibility, necessity, belief, knowledge, temporal progression and other modalities. It is customary to add the operator [] with the interpretation determined by the logic. A second operator <> is the dual of the first i.e. <>A = ¬[]¬A and []A = ¬<>¬A. Figure 1 illustrates some readings of the formulas []A and <>A.

Figure 1: **Reading of []A and <>A**

	[] A		**<>A**
Necessity	A is necessary.	Possibly	A is possible.
Belief	A is believed.		
Knowledge	A is known.		
Time	A is always true.	Eventually	A is eventually true.

There are modal logics that can be used to express ideas such as:

- It might rain tonight.
- Life is unfair.
- Mary believes that John loves her.
- I know that you know that I know that you know I will be leaving town tomorrow.
- He went to town for some supplies, is now carving a duck, and when he is finished, he will paint it.

The concept of necessity can be understood in several different contexts:

- *Logical necessity*: logic requires it to be so. if A and A->B are true , then B must be true.
- *Epistemic necessity*: reality requires it to be so. What goes up must come down.
- *Moral necessity*: morality requires it to be so. Sinners will be punished.
- *Temporal necessity*: Since Camile was born in 1985, she must be at least 14 years old.

Propositional modal logics provide some of the expressive power of both first and second order logic

and find applications in

- Artificial intelligence research area such as
  - natural language translation and
  - reasoning systems dealing with theories of knowledge, belief, and time.
- Database systems
- Software engineering
  - Program specification
  - Program verification
  - Protocol specification
- Theories of program behavior
  - Algorithmic logic
  - dynamic logic
  - process logic
  - temporal logic

Temporal logic plays an important role in the specification, derivation, and verification of programs as programs may be viewed as progressing through a sequence of states, a new state after each event in the system.

The key notion in the semantics of modal logic is the notion of possible worlds.

Syntax

Figure 2: **The Syntax**

Symbols and Formulas:

C = { _|_, -|-} The propositional constants.
L = { p_0, p_1, p_2, ...} The propositional letters.
P in C union L
F ::= P | ¬F | /\FF | \/FF | ->FF | **[]**F | **<>**F {The set of formulas}

Axioms and Inference Rules:

T = The tautologies are the axioms

$$\frac{A,\ A \rightarrow B}{B}$$ The inference rule, A & B are formulas

Additional information on syntax is available.

Semantics - Multiple Worlds (Saul Kripke)

Multiple structures can be used to model *modal* formulas.

Figure 3: **Multiple world structure** $U = (W, A)$

A set of relational structures: $W = \{w \mid w \text{ is a relational structure}\}$

An accessibility relation: A a subset of $W \times W$

The set of constants in each world is monotonic in the accessibility relation however, the worlds may differ in the atomic formulas that hold for each world. ***U*** is a graph whose edges are labeled with literal formulas (the formulas required to be true by the valuation function). A model ***M*** = (***W***, ***A***, ***w***, *v*) is a Kripke structure. A Kripke structure where |**W**| = 1 corresponds to traditional logics. A reachability relation that is symmetric (**Axy** implies **Ayx**) implies that the graph is nondirectional. A reachability relation that is transitive (**Axy** and **Ayz** implies **Axz**) can be used to model temporal phenomenon. A reachability relation that is reflexive (**Axx**), symmetric, and transitive, can be used to reason about finite state systems.

The propositional modal logics share with classical propositional logic the finite model property; if a collection of formulas is satisfiable, it is satisfiable in a finite graph.

There are many modal logics. The table that follows illustrates the approach to semantics for modal logics.

Figure 4: **Model -** ***M*** |=F
where ***M***= (***U***, ***w***, *v*)

M |= _|_	iff *v*(_|_) = false
M |= -|-	iff *v*(-|-) = true
M |= p	iff *v*(p) in ***w***
M |= ¬A	iff not ***M*** |= A
M |= ->AB	iff ***M*** |= ¬A or ***M*** |= B
M |= ¬->AB	iff ***M*** |= A and ***M*** |= ¬B
M |= **[]**A	iff ***M'*** |= A for all ***u*** such that ***Awu*** and ***M'*** = (***U***, ***u***, *v*)
M |= **<>**A	iff ***M'*** |= A for some ***u*** such that ***Awu*** and ***M'*** = (***U***, ***u***, *v*)
M |= /\x.F	iff ***M*** |= $[F]^x_c$ for all c in C
M |= ¬\/x.F	iff ***M*** |= $[\neg F]^x_c$ for some c in C

A formula F is **valid** (a **tautology**), |= F, iff for all **w** in **W**, ***M***|= F i.e., F *is true in all possible worlds*.

A formula F is said to be **valid** (|=F) iff it is valid in all models ***M*** (***M*** |= F for all ***M***). A valid formula is called a **tautology**. Predicate Logic (or Predicate Calculus or First-Order Logic) is a generalization of Propositional Logic. Generalization requires the introduction of variables.

Linear time temporal logic is an example of a logic that uses multiple world semantics. Each time increment is represented by a world. The accessibility relation is reflexive and transitive but not symmetric as we assume that time does not run backwards. For the formula **[]**A, A holds in the current world and in all future worlds and for the formula **<>**A, A holds in either the current world or some future world.

Program specifications in temporal logic:

- Safety properties: **[]**P
- Liveness properties: **<>**P
- Safe-livenes property: **[]**(A->**<>**B)
- The end of time: **¬[]<>A**

Additional temporal operators include

- OP - next time
- P*U*Q - P until Q

Definition

- A sentence S of L is **valid**, |=S, if it is true in all structures for ***L***.
- A sentence S of L is a **logical consequence** of a set of sentences Ss of L (Ss |= S), if S is true in every structure in which all of the members of Ss are true.
- A set of sentences Ss, is **satisfiable** if there is a structure *A* in which all of the members of Ss are true. Such a structure is called a **model** of Ss. If Ss has no model, it is **unsatisfiable**.

Proofs in classical logic concern truth in a single state while proofs in modal logics may involve several states. Since a formula may refer to a state other than the one in which it appears, once the collection of states has been constructed, the states must be checked to determine that all such references are satisfied.

Tableau rules for modal logic are available.
An implementation for propositional modal logic is available.
An implementation for first-order modal logic is available.

Proof Theory

In classical logic, the idea was to systematically search for a structure agreeing with the starting sentences. The result being that we get such a structure or each possible analysis leads to a contradiction. In modal logic, we try to build a frame agreeing with the sentences or see that all

attempts lead to contradictions.

The Accessibility Relation

Gore 1992 has a wonderful list of axioms, a naming scheme

Figure : **Model operators and the accessibility relation**

M **|= []**A	iff ***M'*** |= A for all ***u*** such that ***Awu*** and ***M'*** = (***U***, ***u***, *v*)
M **|= <>**A	iff ***M'*** |= A for some ***u*** such that ***Awu*** and ***M'*** = (***U***, ***u***, *v*)

Property	**Axiom**	**Tableau rule**
reflexive	**T:** []A => A	
symmetric	**B:** A => []**<>**A	
transitive	**4:** []A => [][]A	
serial	**D:** []A => **<>**A	

Temporal logic and the Next time operator

	Formula		**Recursive definition**
Always A	[]A	=	A /\ 0[]A
Eventually A	**<>**A	=	A \/ 0**<>**A
A Until B	A *U* B	=	B \/ (A /\ 0(A *U* B))

Models

K(ripke) - minimal modal logic

Axioms

1. All propositional tautologies
2. [](p->q) -> ([]p->[]q)

Rules of inference

1. from p and p->q, infer q.

A theorem prover for K is available.

B - self knowledge and knowledge of falsehoods

reflexive and symmetric

Axioms

1. All propositional tautologies
2. []p -> p (reflexivity)
3. [](p->q) -> ([]p->[]q)
4. []p->[][]p (transitivity)

The accessibility relation must be reflexive and transitive.

Rules of inference

1. from p and p->q, infer q.
2. from p, infer []p.

D, D4, K, K4, K5, KB

T - knowledge is true belief

reflexive

M

Axioms

1. All propositional tautologies
2. []p -> p (reflexivity)
3. [](p->q) -> ([]p->[]q)

Rules of inference

1. from p and p->q, infer q
2. from p, infer []p

S4 - positive self reflection

reflexive and transitive

Axioms

1. All propositional tautologies
2. []p -> p
3. [](p->q) -> ([]p->[]q)
4. []p ->[][]p

Rules of Inference

1. from p and p->q, infer q
2. from p, infer []p

S5, E - knowledge of non-knowledge (complete awareness)

reflexive, symmetric, & transitive

Axioms

1. All propositional tautologies
2. []p -> p
3. [](p->q) -> ([]p->[]q)
4. []p ->[][]p
5. <>p -> []<>p

Rules of Inference

1. from p and p->q, infer q
2. from p, infer []p

Belief, Knowledge and Self-Awareness

What kind of logic allows us to reason about our set of beliefs and our own self-awareness?

Connections

- Related to: Modal
- Prerequisites: Formal Systems, Classical
- Requisite for:

The following is adapted from Raymond Smullyan's book *Forever Undecided*.

Reasoning about beliefs requires a set of beliefs and a logic. The following is a description of a propositional belief logic based on propositional logic extended with the symbol ***B***; ***B***X is read `believes *X*' with the meaning that *X* is in the set of beliefs (See Figure N.1 and N.2).

Figure 1: **Syntax**

Symbols and Formulas:

L = { p_0, p_1, p_2, ...} The propositional letters.
P in L
F ::= P | ¬F | ->FF | ***B***F {The set of formulas}

The syntax was chosen to avoid the use of parentheses. Informally, we use the more common infix notation and additional logical operators.

- Conjunction: (A /\ B) = ¬->A¬B
- Disjunction: (A \/ B) = ->¬AB
- Biconditional: (A<->B) = (->AB) /\ (->BA) = (/\AB) \/ (/\¬A¬B)
- Conditional: (A->B) = ->AB

The semantics of Belief Logic include a set of beliefs and extend the valuation function of propositional logic to formulas containing the belief operator. Since beliefs are collections of formulas, we map a formula to true iff it is in the list of beliefs (See Figure N.2).

Figure 5: **Model theory: Semantics**

Structure and Interpretation. <SB, v>

B = {**false**, **true**} The boolean values
V = F ->B, The set of valuation functions.
v in V
SB = a set of formulas called beliefs
P in L
A, B in F

A function v is a valuation function if it is a total function on L and with the structure <v, SB> satisfies the following properties:

<SB, v> |= P	iff v(P) = **true**
<SB, v> |= ¬P	iff v(P) = **false**
<SB, v> |= ¬¬A	iff <SB, v> |= A
<SB, v> |= ¬->AB	iff <SB, v> |= A and <SB, v> |= ¬B
<SB, v> |= ->AB	iff <SB, v> |= ¬A or <SB, v> |= B
<SB, v> |= ***B*** A	iff SB |= A
<SB, v> |= ¬***B*** A	iff SB |= ¬A

A formula F is **satisfiable** iff it is true under some valuation function v and a set of beliefs SB, i.e., <v, SB> |= F. A formula F is a **tautology** iff it is true under all valuation functions. A tautology is said to be **valid** and is written |= F.

Figure 6: **Proof theory: inference**

Axioms	A the tautologies
Rule of inference: *Modus Ponens*	A and A->B --- B

A set of formulas is said to be **logically closed** iff it *contains all tautologies* and is *closed under modus ponens* (for any formulas A and B, if A and ->AB are in the set, then so is B).

A logically closed set of formulas is said to be **inconsistent** iff it *contains both a formula and its negation*, i.e., there is a formula A in the set such that both A and ¬A are in the set. Equivalently, a set of formulas is said to be **inconsistent** iff it *contains all formulas*. A logically closed set of formulas is said to be **consistent** if it is not inconsistent.

Theorem: Prove that the two definitions of inconsistent are equivalent.
Proof: Clearly, the second definition implies the first. So, let ***F*** be a logically closed set of formulas that contains A and ¬A. Let Q be an arbitrary formula. The formulas, A->¬A->A/\¬A, and A/\¬A->Q, are tautologies and so are in ***F***. By three applications of MP, Q is in ***F***.

Definitions

A reasoner is called **accurate** if for any proposition A, if (s)he believes A, then A is true;

$$\boldsymbol{B}A \rightarrow A$$

A reasoner is called **inaccurate** if for some proposition A, if (s)he believes A, then ¬A is true.

$$\boldsymbol{B}A \rightarrow \neg A$$

A reasoner is called **consistent** if the set of all propositions the (s)he believes is a consistent set.

$$\neg(\boldsymbol{B}A \wedge \boldsymbol{B}\neg A)$$

A reasoner is called **normal** if for any proposition P, if (s)he believes A, then (s)he believes that (s)he believes A.

$$\boldsymbol{B}A \rightarrow \boldsymbol{B}\boldsymbol{B}A$$

A reasoner is called **peculiar** if for some proposition A, (s)he believes A and (s)he believes that (s)he doesn't believe A.

$$\boldsymbol{B}A \wedge \boldsymbol{B}\neg\boldsymbol{B}A$$

A reasoner is called **regular** if for any propositions A and B, if (s)he believes A->B, then (s)he also believes (***B***A->***B***B).

$$\text{if } \langle SB, v\rangle \models \boldsymbol{B}(A \rightarrow B) \rightarrow \boldsymbol{B}(\boldsymbol{B}A \rightarrow \boldsymbol{B}B)$$

Observations: A reasoner *believes* that (s)he is consistent if for all formulas, A, (s)he ***B***¬***B***(A/\¬A).

A reasoner *believes* that (s)he is inconsistent if for some formula A, (s)he believes ***BB***(A/\¬A).

Advancing stages of self-awareness

What does it mean for an individual to be self-aware? People are aware of some external and internal events (their thoughts) and are able to recognize their image. They are not aware of their atoms. We interpret the operator ***B*** to mean *aware of*.

Formula	*Means*
BA	I am aware of *A*
BBA	I am aware of my beliefs.
BBBA	I am aware that I am aware of my beliefs

Now we define several types of reasoners.

1. A reasoner is of **type 0** if the set of beliefs are the tautologies.
    1. (S)he believes all tautologies, i.e., if |=X (X is a tautology), then <v, SB> |=***B***X.
2. A reasoner is of **type 1** if the reasoner is of type 0 and the set of the reasoner's beliefs are logically closed i.e.,
    1. If (s)he believes A and believes A->B then (s)he believes B i.e.,

$$\text{from } \langle SB, v\rangle \models \boldsymbol{B}A \text{ and } \langle SB, v\rangle \models \boldsymbol{B}(A \rightarrow B) \quad \text{infer } \langle SB, v\rangle \models \boldsymbol{B}B$$

    for any formulas A and B.
3. A reasoner is of **type 2** if the reasoner is of type 1 and believes that her/his set of beliefs is logically closed i.e.,

$$\langle SB, v\rangle \models \boldsymbol{B}((\boldsymbol{B}A \wedge \boldsymbol{B}(A \rightarrow B)) \rightarrow \boldsymbol{B}B)$$

    for any formulas A and B. Reasoners of type 2 know how they reason -- know their inference rule.
4. A reasoner is of **type 3** if the reasoner is of type 2 and is ***aware of her/his beliefs*** i.e.,

$$\langle SB, v\rangle \models \boldsymbol{B}A \rightarrow \boldsymbol{BB}A$$

    for any formula A. Any reasoner that is aware of her/his beliefs is said to be **normal**.
5. A reasoner is of **type 4** if the reasoner is of type 3 and is ***aware that (s)he is aware of her/his beliefs*** i.e.,

$$\langle SB, v\rangle \models \boldsymbol{B}(\boldsymbol{B}A \rightarrow \boldsymbol{BB}A)$$

    for any proposition A. A reasoner of type 4 *knows*that (s)he is normal.

Exercises/Theorems

1. Explain: Reasoners of type 2 *know* how they reason -- they know their inference rule.
2. Explain: A reasoner of type 4 *knows* that (s)he is normal.
3. Prove that every reasoner of type 3 believes the proposition: (**B**p/**B**¬p)->**B**(p/\¬p).

Proof: The following hold for type 3 reasoners: ***B***A->***BB***A, ***B***((***B***A /\ ***B***(A->B))->***B***B). Assume ¬**B**[(**B**p/**B**¬p)->**B**(p/\¬p)] for some p. **B**p/**B**¬p, ¬**B**(p/\¬p)

4. Prove that every reasoner of type 3 is regular.
5. Prove that if a regular reasoner of type 1 believes ***B***A for some proposition A then (s)he must be normal.
6. Prove that any peculiar normal reasoner of type 1 must be inconsistent.
7. Prove that every reasoner of type 4 knows that (s)he is normal.
8. Prove that any reasoner of type 4 knows that if (s)he should ever be peculiar, (s)he will be inconsistent.

Awareness of Self-Awareness

A reasoner believes that (s)he is of type 1 if (s)he believes all propositions of the form: ***B***X where X is a tautology and believes all propositions of the form: (***B***A /\ ***B***(A->B))->***B***B

BBX - X is a tautology
B((***B***A /\ ***B***(A->B))->***B***B)

A reasoner believes that (s)he is of type 2 if (s)he believes that (s)he is of type 1 and believes all propositions of the form: ***B***((***B***A/***B***(A->B))->***B***B

BBX - X is a tautology
B((***B***A /\ ***B***(A->B))->***B***B)
BB((***B***A/***B***(A->B))->***B***B)

A reasoner believes that (s)he is of type 3 if (s)he believes that (s)he is of type 2 and believes all propositions of the form: ***B***A ->***BB***A

BBX - X is a tautology
B((***B***A /\ ***B***(A->B))->***B***B)
BB((***B***A/***B***(A->B))->***B***B)
B(***B***A -> ***BB***A)

A reasoner believes that (s)he is of type 4 if (s)he believes that (s)he is of type 3 and believes all propositions of the form: ***B***(***B***A ->***BB***A)

BBX - X is a tautology
B((***B***A /\ ***B***(A->B))->***B***B)
BB((***B***A/***B***(A->B))->***B***B)
B(***B***A -> ***BB***A)
BB(***B***A -> ***BB***A)

A reasoner **knows** that (s)he is of type X if (s)he is of type X and believes that (s)he is of that type.

Exercises/Theorems

1. Prove that if a reasoner of type 4 knows that (s)he is regular.
2. Prove that a reasoner of type 4 knows that (s)he is of type 4.
3. Prove that if a reasoner of type 4 ever believes that (s)he cannot be inconsistent, (s)he will become inconsistent.
4. Suppose a normal reasoner of type 1 believes a proposition of the form p<->¬***B***p. Then:
    1. If (s)he ever believes p, then (s)he will become inconsistent.
    2. If (s)he is of type 4, then (s)he knows that if (s)he should ever believe p then (s)he will become inconsistent--i.e., (s)he will believe the proposition ***B***p->***B***_|_.
    3. If (s)he is of type 4 and believes that (s)he cannot be inconsistent, then (s)he will become inconsistent.

The following is based on a paper by Halpern.

Knowledge

A logic of knowledge is used as a tool for analyzing multi-agent systems - players in a poker game, processes in a computer network, or robots on an assembly line.

We introduce three new operators:

- $\boldsymbol{K}_i$A - agent i knows A if A is true in all worlds agent i thinks possible
- $\boldsymbol{E}_G$A - each agent in the group G knows A
- $\boldsymbol{C}_G$A - A is common knowledge among the agents in the group G

Common knowledge is defined as "everyone knows A, and everyone knows that everyone knows A, and ..."

Figure : **Syntax and Semantics**

Symbols and formulas:

The propositional formulas
L = { p_0, p_1, p_2, ...}, P in L
The set of formulas
F ::= P | ¬F | /\FF | $\boldsymbol{K}_i$F | $\boldsymbol{E}_G$F | $\boldsymbol{C}_G$F

M = (S, v, K_1, ... , K_n) - A Kripke structure where

S - a set of states or possible worlds
v in SxL -> {0,1}
K_i an equivalence relation on S

(M, s) |= p	iff	v(s,p) = 1
(M, s) |= ¬A	iff	not (M, s) |= A
(M, s) |= A/\B	iff	(M, s) |= A and (M, s) |= B
(M, s) |= $\boldsymbol{K}_i$A	iff	(M, t) |= A for all t such that (s,t) in K_i
(M, s) |= $\boldsymbol{E}_G$A	iff	(M, s) |= $\boldsymbol{K}_i$A for all i in G
(M, s) |= $\boldsymbol{C}_G$A	iff	(M, s) |= $\boldsymbol{E}^k{}_G$A for k=1, 2, ..., where $\boldsymbol{E}^1{}_G$A = $\boldsymbol{E}_G$A and $\boldsymbol{E}^{k+1}{}_G$A = $\boldsymbol{E}_G\boldsymbol{E}^k{}_G$A

The following axiom system is sound and complete.

Figure : **Axioms**

A1. All propositional tautologies
A2. $\boldsymbol{K}_i$A /\ $\boldsymbol{K}_i$(A->B) ->$\boldsymbol{K}_i$B
A3. $\boldsymbol{K}_i$A -> A
A4. $\boldsymbol{K}_i$A -> $\boldsymbol{K}_i\boldsymbol{K}_i$A
A5. ¬$\boldsymbol{K}_i$A -> $\boldsymbol{K}_i$¬$\boldsymbol{K}_i$A
R1. From A and A->B infer B
R2. From A infer $\boldsymbol{K}_i$A
C1. $\boldsymbol{E}_G$A -> /\$_{i \text{ in } G}$ $\boldsymbol{K}_i$A
C2. $\boldsymbol{C}_G$A -> $\boldsymbol{E}_G$(A/\$\boldsymbol{C}$A)
RC1. From A -> $\boldsymbol{E}_G$(A/\B) infer A ->$\boldsymbol{C}_G$B

A1 and R1 are a sound an complete axiom system of classical propositional logic. A2 says that an agent's knowledge is closed under implication. A3 says that an agent knows only things that are true. This axiom is usually taken to distinguish *knowledge* from *belief* i.e., false statements may be believed but not known. A4 and A5 are axioms of introspection; these are usually rejected by philosophers.

C2 (fixed point axiom) says that common knowledge of A holds exactly when everyone in the group

knows A and that A is common knowledge.

Exercises

1. Rewrite the following in English: $\boldsymbol{K}_1\boldsymbol{K}_2$A /\ ¬$\boldsymbol{K}_2\boldsymbol{K}_1\boldsymbol{K}_2$A.
2. Express symbolically, Dean doesn't know whether Nixon knows that Dean knows that Nixon knows that McCord burgled O'Brien's office at Watergate. Hint: let A be the statement "McCord burgled O'Brien's office at Watergate".
3. Express symbolically, Everyone in G knows p, but p is not common knowledge.
4. Construct a model for the situation where agent 1 does not know "it is sunny in San Francisco" but agent 2 does.
5. Interpret axiom A4: $\boldsymbol{K}_i$A -> $\boldsymbol{K}_i\boldsymbol{K}_i$A
6. Interpret axiom A5: ¬$\boldsymbol{K}_i$A -> $\boldsymbol{K}_i$¬$\boldsymbol{K}_i$A.
7. Show that A3 - A5 are valid.
8. Show that C2 is necessary for agreement and coordination.
9. Show that the theory is decideable and decidability is of exponential complexity.

Multi-Agent Systems

Logical Omniscience

An agent is **logically omniscient** iff it knows all tautologies and its knowledge is closed under modus ponens.

What is logically knowable is not realizable in practice since real agents are *resource-bounded*. Attempts to define knowledge in the presence of bounds include

- restricting what an agent knows to a set of formulas which is not closed either under inference or all instances of the a given axiom.
- defining a set of possible worlds which in turn, defines a set of formulas.
- change the logic to a non-standard logic such as relevance logic
- a impossible worlds to the list of worlds
- restrict the depth of ***K***s found in formulas
- add an operator for awareness so that the formulas that an agent is aware of is a subset of the formulas. An agent knows a formula iff it is true in each possible world of the agent.
- awareness can be defined to mean that an agent can use a local algorithm to compute an answer.

Future Directions

- Implement a logical agent.
- Reason about knowledge/belief change over time
- Knowledge based programming: The goal is to allow the programmer to write a program by saying what she wants rather than painfully describing how to compute what she wants.
- Analyze protocols and construct logical agents to implement the protocol
- More realistic models of knowledge that incorporate resource-bounded reasoning, probability, and the possibility of errors.
- A deeper understanding of the interplay between various modes of reasoning under uncertainty.

References

Halpern, Joseph Y
Reasoning about Knowledge: A Survey (1995)
Gore Rajeev Prabharkar
Cut-free Sequent and Tableau Systems for Propositional Normal Modal Logics. (1992)
Smullyan, Raymond (1987)
Forever Undecided Alfred A. Knopf Inc.
Beckert, Bernhard and Gore, Rajeev
ModLeanTAP
Advances in Modal Logic

Author: Anthony A. Aaby
Last Modified - .
Comments and content invited aabyan@wwc.edu

Multi-valued Logics

Connections

- Related to:
- Prerequisites: Overview, Classical
- Requisite for:

Multi-valued logics have valuation functions that map atomic formulas to more than two values. Figure 1 provides several difference definitions of valuation functions.

Figure 1: **Valuation functions for classical and multi-valued logics**

v : At -> {0, 1}	*v* is boolean valued as in classical logic
v : At -> {_|_, 0, 1}	*v* identifies undefined formulas such as - y > 1/x.
v : At -> {0, ... , n}	*v* is a multivalued logic with a range of values.
v : At -> [0, 1]	*v* is infinite valued and is suitable for use in a probabilistic, fuzzy and other continuous logics.

The challange for infinite valued logics if to find a way to determine the value of a proposition.

Syntax and Semantics

The syntax of the language ***L*** is described using the symbols of the language ***L***, standard mathematical notation, and meta symbols from natural languages.

Figure 2: **Language - *L* = (C, V, P, F)**

C = { c_i | i = 0, 1, ... } A set of constants k_i in C

V = { x_i | i = 0, 1, ... } A set of variables; x in V

P = { p_0^0, p_0^1, ... , p_1^0, p_1^1, ... , p_2^0, p_2^1, ... , ...} A set of predicate symbols.

At = { $p_i^j k_0...k_{j-1}$ | p_i^j in P and $k_0,...,k_{j-1}$ in C } A set of **atomic formulas**; P in At.

F ::= At | ¬F | /\FF | \/FF | ->FF | <->FF -- *propositional formulas*

| /\x.[F]^k_x | \/x.[F]^k_x-- *first-order formulas*

where

Textual substitution, $[F]^k_x$, is part of the meta language.

For every formula, A and B, predicate symbol, p_i^{j-1}, symbols x and y, and c,

the **substitution** of c for x in A, $\mathbf{[A]^x_c}$ is defined as follows:

- $[p_i^{j-1}v_0,...,v_j]^x_c = p_i^{j-1}[v_0]^x_c, ... , [v_j]^x_c$ where for distinct variables x and y
  - $v_i = x$; $[x]^x_c = c$
  - $v_i = y$; $[y]^x_c = y$
  - $v_i = c_k$; $[y]^x_c = c_k$
- $[¬A]^x_c = ¬[A]^x_c$
- $[/\AB]^x_c = /\[A]^x_c[B]^x_c$
- $[\/AB]^x_c = \/[A]^x_c[B]^x_c$
- $[->AB]^x_c = ->[A]^x_c[B]^x_c$
- $[<->AB]^x_c = <->[A]^x_c[B]^x_c$
- $[/\xA]^x_c = /\x.A$
- $[/\y.A]^x_c = /\y.[A]^x_c$

There are several alternate notations for textual substitution.

There are several alternatives expressions of syntax.

There is a **valuation** function ***v*** from formulas to the interval [0,1]. The function ***v*** is a total function on the set of atomic formulas. It is assumed that the valuation functions must be *generalizations of the valuation functions for classical logic*. Figure N.1 summarizes these concepts.

Figure 3: **Valuation function for a Multi-valued Logic**

v : F --> B i.e. ***v*** in V

Semantic Equations:

Every valuation function ***v*** is a total function on At
v(**f**) = 0
v(¬A) = 1 if ***v***(A) = 0
v(¬A) = 0 if ***v***(A) > 0
v(A /\ B) = min(***v***(A), ***v***(B))
v(A \/ B) = max(***v***(A), ***v***(B))
v(->AB) = 1 if ***v***(A) <= ***v***(B)
v(->AB) = ***v***(B) if ***v***(B) < ***v***(A)
v(<->AB) = 1 - | ***v***(A) - ***v***(B) |
v(/\x.A) = $\min_{c \text{ in } C}$(***v***(P(c))
v(\/x.A) = $\max_{c \text{ in } C}$(***v***(P(c))

Some definitions of suitable valuation functions are given in the table below. Some of these definitions do not preserve deMorgan's laws.

Figure 4: **Alternate Valuation functions**

A	¬ A
x	1- x

A	B	A \/ B
x	y	max(x,y)
x	y	min(1,x+y)
x	y	x + y - xy

A	B	A /\ B
x	y	min(x,y)
x	y	max(0,x+y-1)
x	y	xy

A	B	A -> B
x	y	max(1-x,y)
x	y	min(1,1-x+y)
x	y	1- x + xy

A	B	A <-> B
x	y	1 - |x - y|
x	y	min(max(1-x,y),max(1-y,x))
x	y	max(min(x,y),min(1-x,1-y))
x	y	max(0,min(1,1-x+y)+min(1,1-y+x)-1)

Truth value	**Valuation functions (Maple definition)**
v(A <=> B)	= iff1a := (a,b) -> and1(if1(a,b),if1(b,a)); = iff1b := (a,b) -> or1(and1(a, b), and1(neg(a), neg(b))); = iff2a := (a,b) -> and2(if2(a,b),if2(b,a)); = iff2b := (a,b) -> or2(and2(a,b),and2(neg(a),neg(b))); = iff3a := (a,b) -> and3(if3(a,b),if3(b,a)); = iff3b := (a,b) -> or3(and3(a,b),and3(neg(a),neg(b)));

universal
quantification:
All x.P(x)

$\min_{c \text{ in } C}(v(P(c))$

existential
quantification:
Exist x.P(x)

$\max_{c \text{ in } C}(v(P(c))$

Inference Rules

	$\frac{A \vee B}{A \mid B}$	
Modus Ponens	$\frac{A,\ A \rightarrow B}{B}$	If A holds and the rule `if A then B' holds, conclude B holds.
Modus Tollens	$\frac{\neg B,\ A \rightarrow B}{\neg A}$	If B is false and the rule 'if A then B' holds, conclude A is false.
Gamma	$\frac{\text{forall } x.A(x)}{A(c)}$	If A(x) holds for all x, then A(c) holds for any constant c.
Delta	$\frac{\text{exists } x.A(x)}{A(c)}$	If there exists and x such that A(x) holds, then conclude A(c) holds for c a new constant.
Default	$\frac{A: B,...}{C}$	If A holds and B is consistent, then conclude C holds.

Lukasiewicz valuation function for multi-valued logics

Exercises

1. Show how to derive the various valuation functions using deMorgan's rules.

Reasoning with multi-valued logic

Inference rules

- *Modus ponens:* If A and A->B are valid then so is B i.e.,

  A[x], (A[x] --> B[y])[z] | B[y] so y = f(x,z).
- *Modus tollens:* If not B and A -> B are valid, then so is not A. i.e.,

  not B[y], (A[x] --> B[y])[z] | not A[x] so x = f(y,z).

A small implementation is available.

Fuzzy Logic

Fuzzy Predicates: tall, young, small, medium , normal, expensive near, intelligent, ...
Fuzzy Truth values: true false, fairly true, very true
Fuzzy probabilities: likely, unlikely, very likely highly unlikely
Fuzzy quantifiers: many few, most, almost all

References

Continuous Logic

Poli, R., Ryan, M., Sloman, A
A New Continous Propositional Logic Technical Report: CSRP-95-9; University of Birmingham 1995

Default logic

Besnard, Philippe
An Introduction to Default Logic Springer-Verlag 1989

Fuzzy logic

Klir, St. Clair, & Yuan (1997)
Fuzzy Set Theory: Foundations and Applications Prentice-Hall 1997
Klir & Yuan (1995)
Fuzzy Sets and Fuzzy Logic: Theory and Applications Prentice-Hall 1995

Implementation

Sterling and Shapiro (1986)
The Art of Prolog MIT Press 1986

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Constructive (Intuitionistic) Logic

Connections

- Related to: Modal logics, Natural deduction and sequents
- Prerequisites: Overview, Classical
- Requisite for:

Intuitionists conclude that *the meaning of a statement resides not in its truth conditions but in the means of proof or verification*. In classical logic, disjunctive formulas of the form P \/ ¬P are proveable without providing a proof of either P or of ¬P and some formulas of the form \/x.F(x) (such as \/x./\y(p(x) -> p(y)) is proveable without providing a proof of F(t) for some particular t. *Intuitionism* is a school of philosophy of mathematics that questions these tenets of classical logic. Intuitionism demands a *constructive* interpretation of the quantifiers: if \/*x*.*F* is true, then the value of *x* satifying *F* should be effectively computable. Thus intuitionistic proofs contain more information than classical proofs. Hence intuitionistic logic can be used for program synthesis. In intuitionistic type theory, a proof of \/*x*.*F* constructs a function to compute *x*. However, proof search in intuitionistic logic is more difficult than in first-order classsical logic; there are no normal forms like conjunctive normal form or prenex form and Skolemization cannot, in general, be applied to intuitionistic formulas.

The following principles were formulated by Brouwer, Heyting, and Kolmogorov and are called the *BHK*-interpretation of constructive logic.

1. A proof of A/\B is given by presenting a proof of A and a proof of B.
2. A proof of A\/B is given by presenting either a proof of A or a proof of B and indicating which proof it is.
3. A proof of A->B is a procedure which permits us to transform a proof of A into a proof of B.
4. The constant *false*, a contradiction, has no proof.
5. A proof of ¬A is a procedure that transforms any hypothetical proof of A into a proof of a contradiction.q

Syntax

Figure 1: **Intuitionistic Logic - The Syntax**

Symbols and Formulas:

L = { p_0, p_1, p_2, ... } The propositional letters.
P in L
F ::= **f** | P | /\FF | \/FF | ->FF | /\x.A | \/x.A {The set of formulas}

¬A abbreviates ->A**f**.
<->AB abbreviates /\->AB->BA.

Kripke Model (Possible Worlds)

Let ***W*** be a set of worlds and ***A*** a reflexive, transitive accesibility relation on ***W***×***W***. Each ***w*** in ***W*** is a valuation function on the language ***L***. ***W*** is a graph whose edges are labeled with literal formulas (the formulas required to be true by the valuation function).

For all **a**, **b**, **c** in **W**

- **A** is reflexive and transitive.
- **Aab** implies A_a subset of A_b. Any sequence of constants is monotonic.
- **Aab** implies v_a(At) subset of v_b(At). Any sequence of atomic formulas is monotonic.

For some Kripke model ***M***, **a** in **W**, and a formula A, $M \models_a A$ is defined as follows:

Figure 1: **Intuitionistic Logic**

not $M \models_a$ **f**	for all **a** in **W**.
$M \models_a$ p	iff p in C_b all **b** such that **Aab**
$M \models_a$ ¬p	iff p *not* in C_a
$M \models_a$ ¬¬F	iff for some **b** such that **Aab,** $M \models_b$¬F
$M \models_a$ /\AB	iff $M \models_a$ A and $M \models_a$ B
$M \models_a$ \/AB	iff $M \models_a$ A or $M \models_a$ B
$M \models_a$ ->AB	iff for all **b** such that **Aab,** $M \models_b$¬A or $M \models_b$ B
$M \models_a$ ¬->AB	iff for some **b** such that **Aab,** $M \models_b$A and $M \models_b$ ¬B
$M \models_a$ /\x.F	iff for all **b** such that **Aab,** $M \models_b [F]^x_c$ all c in C
$M \models_a$ ¬/\x.F	iff for some **b** such that **Aab,** $M \models_b [F]^x_c$ some c in C
$M \models_a$ \/x.F	iff $M \models_a [F]^x_c$ some c in C

$\boldsymbol{M} \models_{\mathbf{a}} \neg\forall x.F$	iff $\boldsymbol{M} \models_{\mathbf{a}} [\neg F]^{x}_{c}$ all in C

A formula is **intuitionistically valid** iff $\boldsymbol{M} \models_a A$ for $\boldsymbol{M}$ and every **a**.
A formula F is said to be **valid** (|=F) iff it is valid in all models $\boldsymbol{M}$ ($\boldsymbol{M}$ |= F for all $\boldsymbol{M}$). A valid formula is called a **tautology**.

Exercises

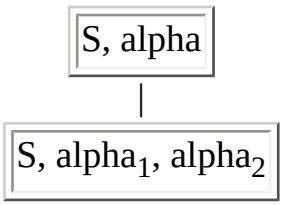
1. Show that for some **a**, $\boldsymbol{M} \models_a \neg A$ iff for every b >= a, not $\boldsymbol{M} \models_b A$.
2. Show that if $\boldsymbol{M} \models_a A$ and b >= a, then $\boldsymbol{M} \models_b A$.
3. Show that for some **a**, not $\boldsymbol{M} \models_a \forall A \neg A$.
4. Show that for some **a**, not $\boldsymbol{M} \models_a -> \neg\neg AA$
5. Show that for some **a**, not $\boldsymbol{M} \models_a -> \neg\wedge x.A\forall x.\neg A$

Analytic Tableau

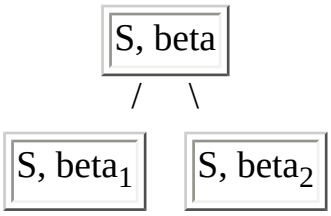
Figure N.4: **Alpha and Beta Rules**

alpha	**$alpha_1$**	**$alpha_2$**
¬ ¬ A	A	A
A /\ B	A	B
¬ (A \/ B)	¬ A	¬ B
¬ (A -> B)	A	¬ B
beta	**$beta_1$**	**$beta_2$**
A \/ B	A	B
¬ (A /\ B)	¬ A	¬ B
A -> B	¬ A	B
A <-> B	A /\ B	¬ B /\ ¬ A
¬ (A <-> B)	¬ A /\ B	¬ B /\ A
gamma	**gamma(c)**	
/\x.F		
delta	**delta(c)**	

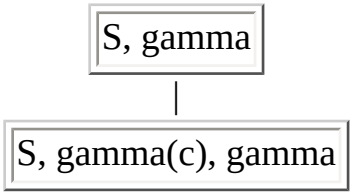
Rule A:



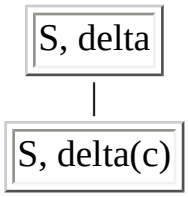
Rule B:



Rule C:



Rule D:



where d is new to the tree

Natural Deduction

Natural deduction has an intuitionistic orientation.

Figure : **Intuitionistic Natural Deduction Rules**

	Introduction		**Elimination**	
/\	$\dfrac{\vdash A, \vdash B}{A \wedge B}$		$\dfrac{\vdash A \wedge B}{A}$	$\dfrac{\vdash A \wedge B}{B}$
\/	$\dfrac{\vdash A}{A \vee B}$	$\dfrac{\vdash B}{A \vee B}$	$\dfrac{\vdash A \vee B,\ A \vdash C,\ B \vdash C}{C}$	

$$\text{->}\ \frac{A \vdash B}{A \text{ -> } B} \qquad \frac{\vdash A,\ \vdash A \text{ -> } B}{B}$$

$$\neg\ \frac{\vdash \mathbf{f}}{B}$$

$$\textbf{Contradiction}\ \frac{\mathbf{f}}{B} \qquad \text{Classical rule is:}\ \frac{[\neg B]\ \mathbf{f}}{B}$$

$$\forall x.\ \frac{A}{\wedge x.A} \qquad \frac{\wedge x.A}{A}$$

$$\forall x.\ \frac{A}{\forall x.A}$$

$$\textbf{Induction}\ \frac{[A]^x_0,\ \wedge xA \text{ -> } [A]^x_{x+1}}{\wedge x.A}$$

Constructive Type Theory

Type theory was originally developed with the aim of being a clarification of constructive mathematics. An introduction to type theory as a theory for program construction. Evaluation of a well typed program always terminates.

References

Nordstrom, Petersson, & Smith
Programming in Martin-Lof's Type Theory Oxford University Press 1990.
Otten, Jens
ileanTAP: An Intuitionistic Theorem Prover

Author: Anthony A. Aaby
Last Modified - .
Comments and content invited aabyan@wwc.edu

Non-monotonic Logic

Connections

- Related to:
- Prequisites:
- Requisite for:

Traditional logics are based on *deduction*, a method of exact inference with the advantage that its conclusions are exact -there is no possibility of mistake if the rules are followed exactly. *Deduction requires that information be complete, precise, and consistent*. By contrast, the real world is mostly made of incomplete, inexact and inconsistent information.

A logic is **monotonic** if the truth of a proposition does not change when new information (axioms) are added to the system. In contrast, a logic is **non-monotonic** if *the truth of a proposition may change* when new information (axioms) is added to or old information is deleted from the system.

Abduction, infer plausible causes of an effect. The abduction inference rule is:

$$\frac{Q(c),\ P(a) \Rightarrow Q(a)}{P(c)}$$

where => is a causal implication and the conclusion is plausible rather than necessary thus differing from modus tollens.

Deduction, exact inverence. Modus ponens and specialization are the two primary inference rules of deduction.

$$\frac{P,\ P \Rightarrow Q}{Q} \qquad \frac{\forall x.P(x)}{P(c)}$$

Induction, infer generalizations from a set of events. The induction inference rule is:

$$\frac{P(c)}{\forall x.P(x)}$$

Default logic: Inference rule

$\frac{P : D}{C}$	if P is true but D is unknown then C.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Functional Programming

*A **functional program** consists of an expression E (representing both the algorithm and the input). This expression E is subject to some rewrite rules. **Reduction** consists of replacing some part P of E by another expression P' according to the given rewrite rules. ... This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. The expression E* thus obtained is called the **normal form** of E and constitutes the output of the functional program* -H. P. Barendregt

Functional programming is characterized by the programming with values, functions and functional forms.

Keywords and phrases: Lambda calculus, free and bound variables, scope, environment, functional programming, combinatorial logic, recursive functions, functional, curried function.

Functional programming languages are the result of both abstracting and generalizing the data type of maps. Recall, the mapping *m* from each element *x* of *S* (called the domain) to the corresponding element *m(x)* of *T* (called the range) is written as:

m : S --> T

For example, the squaring function is a function of type:

sqr : Num --> Num

and may be defined as:

sqr where x |--> x*x

A linear function f of type

f : Num --> Num

may be defined as:

f where x |--> 3*x + 4

The function:

g where x |--> $3*x^2 + 4$

may be written as the composition of the functions f and sqr as:

$$f \circ sqr$$

where

$$f \circ sqr\ (x) = f(sqr(x)) = f\ (x*x) = 3 * x^2 + 4$$

The compositional operator is an example of a *functional form*. Functional programming is based on the mathematical concept of a function and functional programming languages include the following:

- A set of primitive functions.
- A set of functional forms.
- The *application* operation.
- A set of data objects and associated functions.
- A mechanism for binding a name to a function.

LISP, FP, Scheme, ML, Miranda and Haskell are just some of the languages to implement this elegant computational paradigm.

The basic concepts of functional programming originated with LISP. Functional programming languages are important for the following reasons.

- Functional programming dispenses with the assignment command freeing the programmer from the rigidly sequential mode of thought required with the assignment command.
- Functional programming encourages thinking at *higher levels of abstraction* by providing higher-order functions -- functions that modify and combine existing programs.
- Functional programming has natural implementation in concurrent programming.
- Functional programming has important application areas. Artificial intelligence programming is done in functional programming languages and the AI techniques migrate to real-world applications.
- Functional programming is useful for developing *executable specifications* and *prototype implementations*.
- Functional programming has a close relationship to computer science theory. Functional programming is based on the lambda-calculus which in turn provides a framework for studying decidability questions of programming. The essence of denotational semantics is the translation of conventional programs into equivalent functional programs.

  Terminology. Functional programming languages are called *applicative* since the functions are applied to their arguments, *declarative* and *non-procedural* since the definitions specify what is computed and not how it is computed.

1 The Lambda Calculus

Functional programming languages are based on the lambda-calculus. The lambda-calculus grew out of

an attempt by Alonzo Church and Stephen Kleene in the early 1930s to formalize the notion of computability (also known as *constructibility* and *effective calculability*). It is a formalization of the notion of functions as rules (as opposed to functions as tuples). As with mathematical expressions, it is characterized by the principle that *the value of an expression depends only on the values of its subexpressions.* The lambda-calculus is a simple language with few constructs and a simple semantics. But, it is expressive; it is sufficiently powerful to express all computable functions.

As an informal example of the lambda-calculus, consider the function defined by the polynomial expression

$$x^2 + 3x - 5.$$

The variable x is a parameter. In the lambda-calculus, the notation \x.M is used to denote a function with parameter x and body M. That is, x is mapped to M. We rewrite our function in this format

$$\x.(x^2+ 3x - 5)$$

and read it as ``the function of *x* whose value is defined by $x^2 + 3x - 5$". The lambda-calculus uses prefix form and so we rewrite the body in prefix form,

$$\x. (- (+ (\times\ x\ x) (\times\ 3\ x))\ 5).$$

The lambda-calculus *curries* its functions of more than one variable i.e. (+ x y) is written as ((+ x) y), the function (+ x) is the function which adds something to x. Rewriting our example in this form we get:

$$\x.((- ((+ ((\times\ x)\ x))\ ((\times\ 3)\ x)))\ 5)$$

To denote the application of a function f to an argument a we write

$$f\ a$$

To apply our example to the value 1 we write

$$\x.((- ((+ ((\times\ x)\ x))\ ((\times\ 3)\ x)))\ 5)\ 1.$$

To evaluate the function application, we remove the \x. and replace each remaining occurence of x with 1 to get

$$((- ((+ ((\times\ 1)\ 1))\ ((\times\ 3)\ 1)))\ 5)$$

then evaluate the two multiplication expressions

$$((- ((+ 1)\ 3))\ 5)$$

then the addition

((- 4) 5)

and finally the subtraction

-1.

We say that the variable x is *bound* in the lambda-expression \x.B. A variable occuring in the lambda-expression which is not bound is said to be *free*. The variable x is free in the lambda-expression \y.((+ x) y). The *scope* of the variable introduced (or bound) by lambda is the entire body of the lambda-abstraction.

The lambda-notation extends readily to functions of several arguments. Functions of more than one argument can be *curried* to produce functions of single arguments. For example, the polynomial expression xy can be written as

\x. \y. xy

When the lambda-abstraction \x. \y. xy is applied to a single argument as in (\x. \y. xy 5) the result is \y. 5y, a function which multiplies its argument by 5. A function of more than one argument is reguarded as a *functional* of one variable whose value is a function of the remaining variables, in this case, ``multiply by a constant function."

The special character of the lambda-calculus is illustrated when it is recognized that functions may be applied to other functions and even permit self application. For example let C = \f. \x . (f(fx))

The pure lambda-calculus does not have any built-in functions or constants. Therefore, it is appropriate to speak of the lambda-calculi as a family of languages for computation with functions. Different languages are obtained for different choices of functions and constants.

We will extend the lambda-calculus with common mathematical operations and constants so that \x.((+ 3) x) defines a function that maps x to x+3. We will drop some of the parentheses to improve the readability of the lambda expressions.

A lambda-*expression* is *executed* by *evaluating* it. Evaluation proceeds by repeatedly selecting a *reducible expression* (or *redex*) and reducing it. For example, the expression (+ (* 5 6) (* 8 3)) reduces to 54 in the following sequence of reductions.

```
(+  (*  5 6) (* 8 3))   --> (+  30  (* 8 3))
                         --> (+  30   24)
                         --> 54
```

When the expression is the application of a lambda-abstraction to a term, the term is substituted for the bound variable. This substitution is called \beta-*reduction*. In the following sequence of reductions, the first step an example of \beta-*reduction*. The second step is the reduction required by the addition

operator.

```
(\x.((+ 3) x)) 4

((+  3) 4)

7
```

The pure lambda-calculus has just three constructs: primitive symbols, function application, and function creation. Figure N.1 gives the syntax of the lambda-calculus.

Figure N.1: **The Lambda Calculus**

Syntax:

L in Lambda Expressions
x in Symbols

L ::= x | (L L) | (\x.L)

(L L) is function application, and
(\x.L) is a lambda-abstraction which defines a function with argument x and body L.

We say that the variable x is *bound* in the lambda-expression \x.B. A variable which occurs in but is not bound in a lambda-expression is said to be *free*. The *scope* of \x. is B. In the lambda-expression \y.x+y, x is free and y is bound.

We adopt the following notational conventions:

- We extend the lambda-calculus with the usual constants and functions so we allow

  (\x.((+ x) 3)) to represent the function x + 3

- We usually drop the outermost parentheses so we may write

  \x.((+ x) 3) instead of (\x.((+ x) 3)) and
  \x.((+ x) 3) 4 instead of (\x.((+ x) 3) 4)

- Function application associates to the left so we may write

  (+ x 3) instead of ((+ x) 3) that is, we may write
  \x.+ x 3 instead of \x.((+ x) 3)

- The body of a lambda-abstraction extends as far right as possible so we *must* write

  (\x.+ x 3) 4 instead of \x.+ x 3 4

- Replace the body of a lambda-abstraction with conventional infix notation so we may write

  (\x.x + 3) 4 instead of (\x.+ x 3) 4

- Multiple parameters are written together so we may write

  \xy.x + y instead of \x.\y.x + y

Operational Semantics

Calculation in the lambda-calculus is by rewriting (reducing) a lambda-expression to a normal form. For the pure lambda-calculus, lambda-expressions are reduced by substitution. That is, occurrences of the parameter in the body are replaced with (copies of) the argument. In our extended lambda-calculus we also apply the usual reduction rules. For example,

1.	\x.(x^2 - 5) 3	f(3) where f(x) = x^2 - 5
2.	3^2 - 5	by substitution
3.	9 - 5	power
4.	4	subtraction

The normal form is formally defined in the following definition.

> **Definition:** A lambda-expression is said to be in **normal form** if no **beta-redex**, a subexpression of the form (\x.P Q), occurs in it.

Non-terminating computations are examples of expressions that do not have normal forms. The lambda-expression

(\x.x x) (\x.x x)

does not have a normal form as we shall soon see.

We define substitution, B[x:M], to be the replacement of all free occurences of x in B with M. Figure N.2 contains a formal definition of substitution.

Figure N.2: **Substitution**

s[x:M] = if (s=x) then M else s
(A B)[x:M] = (A[x:M] B[x:M])
(\x.B)[x:M] = (\x.B)
(\y.B)[x:M] = if (z is a symbol not free in B or M) then \z.(B[y:z][x:M])

where s is a symbol, M, A and B are lambda-expressions.

Lambda expressions are simplified using beta-reduction. Beta-reduction applies a lambda-abstraction to an argument producing an instance of the body of the lambda-abstraction in which (free) occurrences of the formal parameter in the body are replaced with (copies of) the argument. With the definition of substitution in Figure N.2 and the formal definition of beta-reduction in Fugure N.3, we have the tools needed to reduce lambda-expressions to normal forms.

Figure N.3: **Beta-reduction**

(\x.B) e ==> B[x:e]

It is easy to see that the lambda-expression

(\x.x x) (\x.x x)

does not have a normal form because when the second expression is substituted into the first, the resulting expression is identical to the given lambda-expression.

Figure 2 defines the operational semantics of the lambda-calculus in terms of beta-reduction.

Figure N.4: **Operational semantics for the lambda-calculus**

Interpreter: reduce expression E to normal form.

Reduce in **L** --> **L**

Reduce[s] = s
Reduce[lambda-x.B M] = *Reduce*[B[x:M]]
Reduce[L_1 L_2] = (*Reduce*[L_1] *Reduce*[L_2])

where

s is a symbol and B, L_1, L_2, and M are lambda-expressions

The operational semantics of Figure N.4 describe a syntactic transformation of the lambda-expressions.

Reduction Order

Given a lambda-expression, the substitution and beta-reduction rules provide the tools required to reduce a lambda-expression to normal form but do not tell us what order to apply the reductions when more than one redex is avaliable. The following theorem, due to Curry, states that if an expression has a normal form, then that normal form can be found by leftmost reduction.

Theorem: If E has a normal form N then there is a leftmost reduction of E to N.

The leftmost outermost reduction (*normal order reduction*) strategy is called *lazy reduction* because it does not first evaluate the arguments but substitutes the arguments directly into the expression. *Eager reduction* is when the arguments are reduced before substitution.

A function is *strict* if it is sure to need its argument. If a function is non-strict, we say that it is *lazy*.

parameter passing: by value, by name, and lazy evaluation

Infinite Data Structures

call by need

streams and perpetual processes

A function f is *strict* if and only if (f _|_) = _|_

Scheme evaluates its parameters before passing (eliminates need for renaming) a space and time efficiency consideration.

Denotational Semantics

In the previous section we looked at the *operational* semantics of the lambda-calculus. It is called operational because it is `dynamic', it sees a function as a sequence of operations. A lambda-expression was evaluated by purely *syntactic* transformations without reference to what the expressions `mean'. The purpose of the *denotational semantics* of a language is to assign a value to every expression in the langauge.

We can express the semantics of the lambda-calculus as a mathematical function, **Eval**, from expressions to values. For example,

Eval[+ 3 4] = **7**

defines the value of the expression (+ 3 4) to be **7**. Actually something more is required, in the case of variables and function names, the function **Eval** requires a second parameter containing the environment *rho* which contains the associations between variables and their values. Some programs go into infinite loops, some abort with a runtime error. To handle these situations we introduce the symbol _|_ pronounced `bottom'.

Figure N.5 gives a denotational semantics for the lambda-calculus.

Figure N.5: **Denotational semantics for the lambda-calculus**

Semantic Domains:

s in **D**

Semantic Function:

Eval in **L** --> **D**

Semantic Equations:

Eval [s] = **s**
Eval [(\x.B M)] = *Eval* [B[x:M]]
Eval [(L_1 L_2)] = (*Eval* [L_1] *Eval* [L_2])
Eval [E] = _|_

where s is a symbol, B, L_1, L_2, and M are expressions, B[x:M] is substitution as in Figure N.2, E is an expression which does not have a normal form, and _|_ is pronounced bottom.

The denotational semantics of Figure N.5 describe a mapping of lambda expressions to values in some semantic domain.

Recursive Functions

We extend the syntax of the lambda-calculus to include named expressions as follows:

Lambda Expressions

L ::= ...| x : L | ...

where x is the name of the lambda-expression L.

With the introduction of named expressions we have the potential for recursive definitions since the extended syntax permits us to name lambda-abstractions and then refer to them within a lambda-expression. Consider the following recursive definition of the factorial function.

FAC : \n.(if (= n 0) 1 (* n (FAC (- n 1))))

which with syntactic sugaring is

FAC : \n.if (n = 0) then 1 else (n * FAC (n - 1))

We can treat the recursive call as a free variable and replace the previous definition with the following.

FAC : (\fac.(\n.(if (= n 0) (* n (fac (- n 1))))) FAC)

Let

H : \fac.(\n.(if (= n 0) 1 (* n (fac (- n 1)))))

Note that H is not recursively defined. Now we can redefine FAC as

FAC : (H FAC)

This definition is like a mathematical equation. It states that when the function H is applied to FAC, the result is FAC. We say that FAC is a *fixed point* or *fixpoint* of H. In general functions may have more than one fixed point. In this case the desired fixed point is the mathematical function factorial. In general, the `right' fixed point turns out to be the unique *least fixed point*.

It is desirable that there be a function which applied to a lambda-abstraction returns the least fixed point of that abstraction. Suppose there is such a function Y where,

FAC : Y H

Y is called a *fixed point combinator*. With the function Y, this definition of FAC does not use of recursion. From the previous two definitions, the function Y has the property that

Y H = H (Y H)

As an example, here is the computation of FAC 1 using the Y combinator.

```
FAC 1 = (Y H) 1
      = H  (Y H) 1
```

```
= \fac.(\n.(if (= n 0) 1 (*  n (fac (- n 1)))))  (Y H) 1
= \n.(if (= n 0) 1 (*  n((Y H)(- n 1)))) 1
= if (= 1 0) 1 (*  1 ((Y H)(-11)))
= (* 1 ((Y H)(-11)))
= (* 1 ((Y H)0))
= (* 1 (H  (Y H) 0))
...
= (* 1 1)
=  1
```

The function Y can be defined in the lambda-calculus.

Y : \h.(\x.(h (x x)) \x.(h (x x)))

It is especially interesting because it is defined as a lambda-abstraction without using recursion. To show that this lambda-expression properly defines the Y combinator, here it is applied to H.

```
(Y H)  =  (\h.(\x.(h  (x  x)) \x.(h  (x x))) H)
       =  (\x.(H  (x  x)) \x.(H  (x x)))
       =  H  ( \x.(H  (x x))\x.(H  (x x)))
       =  H  (Y  H)
```

Lexical Scope Rules

Blocks with local definitions may be defined in the lambda-calculus. We introduce two kinds of blocks, let and letrec expressions. Nonrecursive definitions are introduced with let expressions:

let n : E in B is an abbreviation for (\n.B) E

Here is an example using the let-extension.

let x : 3 in (* x x)

Lets may be used where ever a lambda-expression is permitted. For example,

\y. let x : 3 in (* y x)

is equivalent to

\y. (* y 3)

Simple recursive definitions are introduced with letrec expressions which are defined in terms of let expressions and the Y combinator:

letrec n : E in B is an abbreviation for let n : Y (\n.E) in B

Let and letrec expressions may be nested. The definitions of the let and letrec expressions are restated in Figure N.6.

Figure M.6: **Lexical Scope Rules**

let n : E in B = (\n.B) E
letrec n : E in B = let n : Y (\n.E) in B

Mutual recursion may also be defined but is beyond the scope of this text.

Translation Semantics and Combinators

The beta-reduction rule is expensive to implement. It requires the textual substitution of the argument for each occurrence of the parameter and further requires that no free variable in the argument should become bound. This has lead to the study of ways in which variables can be eliminated.

Curry, Feys, and Craig define a number of *combinators* among them the following:

S = \f .(\g .(\x. f x (g x)))
K = \x .\y. x
I = \x.x
Y = \f. \x.(f(x x)) \x.(f (x x))

These definitions lead to transformation rules for sequences of combinators. The reduction rules for the SKI calculus are given in Figure N.7.

Figure N.7: **Reduction rules for SKI calculus**

S f g x --> f x (g x)
K c x --> c
I x --> x
Y e --> e (**Y** e)
(A B) --> A B
(A B C) --> A B C

The reduction rules require that reductions be performed left to right. If no **S**, **K**, **I**, or **Y** reduction applies, then brackets are removed and reductions continue.

The SKI calculus is computationally complete; that is, these three operations are sufficient to implement any operation. This is demonstrated by the rules in Figure N.8.

Figure N.8: **Translation Semantics for the Lambda calculus**

Compile [s]	--> s
Compile [(E_1 E_2)]	--> (*Compile* [E_1] *Compile* [E_2])
Compile [\x.E]	--> *Abstract* [(x, *Compile* [E])]
Abstract [(x, s)]	--> if (s=x) then **I** else (**K** s)
Abstract [(x, (E_1 E_2))]	--> ((**S** *Abstract* [(x, E_1)]) *Abstract* [(x, E_2)])

where s is a symbol.

which translate lambda-expressions to formulas in the SKI calculus.

Any functional programming language can be implemented by a machine that implements the SKI combinators since, functional languages can be transformed into lambda-expressions and thus to SKI formulas.

Function application is relatively expensive on conventional computers. The principle reason is the complexity of maintaining the data structures that support access to the bound identifiers. The problems are especially severe when higher-order functions are permitted. Because a formula of the SKI calculus contains no bound identifiers, its reduction rules can be implemented as simple data structure manipulations. Further, the reduction rules can be applied in any order, or in parallel. Thus it is possible to design massively parallel computers (*graph reduction machines*) that execute functional languages efficiently.

Recursive functions may be defined with the Y operator.

Optimizations

Notice that the size of the SKI code grows quadratically in the number of bound variables. Figure N.9.

B = \x .(\y .(\z. ((x y) z)))

C = \x .(\y.(\z((x z) y)))

with the corresponding reduction rules.

B a b c -->((a b) c)
C a b c -->((a c) b)

Having these combinators we can simplify the expressions obtained by applying the rules in Figure N.9.

Figure N.9: **Optimizations for SKI code**

S (**K** e) (**K** f) --> **K** (e f)
S (**K** e) **I** --> e
S (**K** e) f --> (**B** e) f
S e (**K** f) --> (**C** e) f

The optimizations must be applied in the order given.

Just as machine language (assembler) can be used for programming, combinatorial logic can be used as a programming language. The programming language FP is a programming language based on the idea of combinatorial logic.

2 Scheme

Scheme, a descendent of LISP, is based on the lambda-calculus. Although it has imperative features, in this section we ignore those features and concentrate on the lambda-calculus like features of Scheme. Scheme has two kinds of objects, **atoms** and **lists**. Atoms are represented by strings of non-blank characters. A list is represented by a sequence of atoms or lists separated by blanks and enclosed in parentheses. **Functions** in Scheme are also represented by lists. This facilitates the creation of functions which create other functions. A function can be created by another function and then the function applied to a list of arguments. This is an important feature of languages for AI applications.

Syntax

The syntax of Scheme is similar to that of the lambda calculus.

Scheme Syntax

E in Expressions

A in Atoms (variables and constants)
...
E ::= A | (E...) | (lambda (A...) E) | ...

Expressions are atoms which are variables or constants, lists of arbitrary length (which are also function applications), lambda-abstractions of one or more parameters, and other built-in functions.

Scheme provides a number of built in functions among which are +, -, *, /, <, <=, =, >=, >, and not. Scheme provides for conditional expressions of the form (if E_0 E_1 E_2) and (if E_0 E_1). Among the constants provided in Scheme are numbers, #f and the empty list () both of which count as false, and #t and any thing other than #f and () which count as true. nil is also used to represent the empty list.

Definitions

Scheme implements definitions with the following syntax

E ::= ...| (define I E) | ...

Lists with `nil, cons, car` and `cdr`

The list is the basic data structure with `nil` repesenting the empty list. Among the built in functions for list manipulation provided in Scheme are `cons` for attaching an element to the head of a list, `car` for extracting the first element of a list, and `cdr` which returns a list minus its first element.

Figure N.10: **Stack operations in Scheme**

```
( define empty_stack
   ( lambda ( stack ) ( if ( null? stack ) \#t \#f )))

( define push
   ( lambda ( element stack ) ( cons element stack ) ))

(define pop
   ( lambda ( element stack ) ( cdr stack )))

(define top
   ( lambda ( stack ) ( car stack )))
```

Figure N.10 contains an example of stack operations writtem in Scheme. The figure illustrates definitions, the conditional expression, the list predicate `null?` for testing whether a list is empty, and the list manipulation functions `cons, car,` and `cdr`.

Local Definitions

Scheme provides for local definitions with the following syntax

Scheme Syntax

...
B in Bindings
...

E ::= ...| (let B_0 E_0) | (let* B_1 E_1) | (letrec B_2 E_2) |...
B ::= ((I E)...)

The `let` definitions are done independently of each other (collateral bindings), the `let*` values and bindings are computed sequentially and the `letrec` bindings are in effect while values are being computed to permit mutually recursive definitions.

3 ML

4 Haskell

In contrast with LISP and Scheme, Haskell is a modern functional programming language.

Figure N.11: **A sample program in Haskell**

```
module AStack( Stack, push, pop, top, size ) where
data Stack a = Empty
             | MkStack a (Stack a)
push :: a -> Stack a -> Stack a
push x s = MkStack x s

size :: Stack a -> Integer
size s = length (stkToLst s) where
            stkToLst Empty         = []
            stktoLst (MkStack x s) = x:xs where xs = stkToLst s

pop :: Stack a -> (a, Stack a)
pop (MkStack x s) = (x, case s of r -> i r where i x = x)

top :: Stack a -> a
top (MkStack x s) = x
```

```
module Qs where

qs :: [Int] -> [Int]
qs [] = []
qs (a:as) = qs [x | x <- as, x <= a] ++ [a] ++ qs [x | x <- as, x > a]


module Primes where

primes :: [Int]
primes = map head (iterate sieve [2 ..])

sieve :: [Int] -> [Int]
sieve (p:ps) = [x | x <- ps, (x `mod` p) /= 0]

module Fact where

fact :: Integer -> Integer
fact 0 = 1
fact (n+1) = (n+1)*fact n -- * "Foo"
fact _ = error "Negative argument to factorial"

module Pascal where

pascal :: [[Int]]
pascal = [1] : [[x+y | (x,y) <- zip ([0]++r) (r++[0])] | r <- pascal]

tab :: Int -> ShowS
tab 0 = \x -> x
tab (n+1) = showChar ' ' . tab n

showRow :: [Int] -> ShowS
showRow [] = showChar '\n'
showRow (n:ns) = shows n . showChar ' ' . showRow ns

showTriangle 1 (t:_) = showRow t
showTriangle (n+1) (t:ts) = tab n . showRow t . showTriangle n ts

    module Merge where

    merge :: [Int] -> [Int] -> [Int]
    merge [] x = x
    merge x [] = x
    merge l1@(a:b) l2@(c:d) = if a < c then a:(merge b l2)
                                       else c:(merge l1 d)

    half [] = []
```

```
half [x] = [x]
half (x:y:z) = x:r where r = half z

sort [] = []
sort [x] = [x]
sort l = merge (sort odds) (sort evens) where
             odds = half l
             evens = half (tail l)
```

5 Historical Perspectives and Further Reading

In the 1930s Alonso Church developed the lambda-calculus as an alternative to set theory for the foundations of mathematics and Haskell B. Curry developed conbinatory logic for the same reason. While their goal was not realized, the lambda-calculus and combinators capture the most general formal properties of the notion of a mathematical function.

The lambda-calculus and combinatory logic are abstract models of computation equivalent to the Turing machine, recursive functions, and Markov chains. Unlike the Turning machine which is sequential in nature, they retain the implicit parallelism that is present in mathematical expressions.

The lambda-calculus is a direct influence on the programming language LISP, the *call by name* parameter passing mechanism of Algol-60, and textual substitution performed by macro generators.

Explicit and systematic use of the lambda-calculus in computer science was initiated in the early 1960s by Peter Landin, Christopher Strachy and others who started a formal theory of semantics for programming languages called *denotational semantics*. Dana Scott (1969) discovered the first mathematical model for the type-free lambda-calculus.

New hardware designs are appearing to support the direct execution of the lambda-calculus or combinators which support parallel execution of functional programs, removing the burden (side-efficts, synchonization, communication) of controlling parallelism from the programmer.

LISP (LISt Processing) was designed by John McCarthy in 1958. LISP grew out of interest in symbolic computation. In particular, interest in areas such as mechanizing theorem proving, modeling human intelligence, and natural language processing. In each of these areas, list processing was seen as a fundamental requirement. LISP was developed as a system for list processing based on recursive functions. It provided for recursion, first-class functions, and garbage collection. All new concepts at the time. LISP was inadvertantly implemented with dynamic rather than static scope rules. Scheme is a modern incarnation of LISP. It is a relatively small language with static rather than dynamic scope rules. LISP was adopted as the language of choice for artificial intelligence applications and continues to be in wide use in the aritficial intelligence community.

ML

Miranda

Haskell is a modern language named after the logician Haskell B. Curry, and designed by a 15-member international committee. The design goals for Haskell are have a functional language which incorporates all recent ``good ideas'' in functional language research and which is suitable for for teaching, research and application. Haskell contains an overloading facility which is incorporated with the polymorphic type system, purely functional i/o, arrays, data abstraction, and information hiding.

Functional programming languages have been presented in terms of a sequence of virtual machines. Functional programming languages can be translated into the lambda-calculus, the lambda-calculus into combinatorial logic and combinatorial logic into the code for a graph reduction machine. All of these are virtual machines.

Models of the lambda-calculus.

History \cite{McCarthy60} For an easily accessable introduction to functional programming, the lambda-calculus, combinators and a graph machine implementation see Revesz (1988). For Backus' Turing Award paper on functional programming see \cite{Backus78}. The complete reference for the lambda-calculus is \cite{Bare84}. For all you ever wanted to know about combinatory logic see \cite{CF68,CHS72,HS86}. For an introduction to functional programming see Henderson (1980), BirdWad88, MLennan90. For an intoduction to LISP see \cite{McCarthy65} and for common LISP see \cite{Steele84}. For a through introduction to Scheme see \cite{AbSus85}. Haskell On the relationship of the lambda-calculus to programming languages see \cite{Landin66}. For the implementation of functional programming languages see Henderson (1980) and Peyton-Jones (1987).

Henderson, Peter (1980)
Functional Programming: Application and Implementation Prentice-Hall International.
Peyton-Jones, Simon L (1987)
The Implementation of Functional Programming Languages Prentice-Hall International.
Revesz, G. E. (1988)
Lambda-Calculus, Combinators, and Functional Programming Cambridge University Press.

6 Exercises

1. [Time/Difficulty](section)
2. Simplify the following expressions to a final (*normal*) form, if one exists. If one does not exist, explain why.
    1. ((\x. (xy))(\z.z))
    2. ((\x. ((\y.(xy))x))(\z.w))
    3. ((((\f.(\g.(\x.((fx)(gx)))))(\m.(\n.(nm))))(\n.z))p)
    4. ((\x.(xx))(\x.(xx)))
    5. ((\f.((\g.((ff)g))(\h.(kh))))(\y.y)))
    6. (\g.((\f.((\x.(f(xx)))(\x.(f(xx)))))g))
    7. (\x.(\y.((-y)x)))45
    8. ((\f.(f3))(\x.((+1)x)))
3. Find a lambda-expression that not only does not have a normal form but grows in length as well.
4. In addition to the \beta-rule, the lambda-calculus includes the following two rules:

```
\alpha-rule: (\x.E) ==> (\y.E[x:y])
\eta-rule: (\x.E  x) ==> E where x does not occur free in  E
```

Redo the previous exercise making use of the \eta-rule whenever possible. What value is there in the \alpha-rule?

5. The lambda-calculus can be used to simulate computation on truth values and numbers.
    1. Let **true** be the name of the lambda-expression \x. \y. x and **false** be the name of the lambda-expression \x. \y. y. Show that ((\mbox{true} E_1)E_2) ==> E_1 and ((\mbox{false} E_1)E_2) ==> E_2. Define lambda-expressions **not, and,** and **or** that behave like their Boolean operation counterparts.
    2. Let **0** be the name of the lambda-expression \x. \y. y, **1** be the name of the lambda-expression \x. \y. (xy), **2** be the name of the lambda-expression \x. \y. (x(xy)), **3** be the name of the lambda-expression \x. \y. (x;(x(xy))), and so on. Prove that the lambda-expression **succ** defined as \z. \x. \y.(x ((zx)y)) rewrites a number to its successor.
6. Recursively defined functions can also be simulated in the lambda-calculus. Let **Y** be the name of the expression \f. \x.(f(xx)) \x.(f(xx))
    1. Show that for any expression E, there exists an expression W such that (**Y**E) ==> (WW), and that (WW) ==> (E(WW)). Hence, (**Y**E) ==> E(E(E(...E(WW)...)))
    2. Using the lambda-expressions that you defined in the previous parts of this exercise, define a recursive lambda-expression **add** that performs addition on the numbers defined earlier, that is, ((**add**m)n) ==> m+n.
7. Let T = AA where A = \xy.y(xxy). Show T F = F (T F). T is Turing's fixed point combinator.
8. Data constructors can be modeled in the lambda-calculus. Let **cons** = (\a. \b. \f. f a b), **head** = (\c. c (\a. \b. a)) and **tail** = (\c. c (\a. \b. b)). Show that
    1. **head** (**cons** a b) = a
    2. **tail** (**cons** a b) = b
9. Show that (((S(KK))I)S) is (KS).
10. What is (((SI)I)X) for any formula X?
11. Compile (\x.+xx) to SKI code.
12. Compile \x. (F (xx)) to SKI code.
13. Compile \x. \y. xy to SKI code. Check your answer by reducing both ((\x. \y. xy) a b) and the SKI code applied to a b.
14. Apply the optimizations to the SKI code for \x. \y. xy and compare the result with the unoptimized code.
15. Apply the optimizations to the SKI code for \x. (F (xy)) and \y. (F (xy)).
16. Association lists etc
17. HOF
18. Construct an interpreter for the lambda calculus.
19. Construct an interpreter for combinatorial logic.
20. Construct a compiler to compile lambda expressions to combinators.

General Setting for Incompleteness

Connections

- Related to:
- Prerequisites:
- Requisite for:

From Smullyan, see reference.

The quintuple, $\boldsymbol{M} = (S, \mathbf{f}, \supset, P, \mathbf{B})$, is a *abstract provability system* where

- *S*is a set of sentences or propositions
- **f**, is a distinguished element of *S* called *falsehood* .
- $\supset$ is a binary operation on elements of *S* such that if X, Y $\in$ *S* then X$\supset$Y $\in$ *S*.
- *P* is a subset of *S* whose elements are called *provable* elements of ***M***.
- **B** is a mapping that assigns to every element X of S an element **B**X of *S*.

A subset *V* of *S* is a *valuation set* if

1. **f** $\notin$ *V*
2. For any X, Y $\in$ *S*, X$\supset$Y $\in$ *V* iff either X $\notin$ *V* or Y $\in$ *V*
3. X in *S* is called a *tautology* if it belongs to every valuation set.

A subset *T* of *S* is called a *truth set*

- if *T* is a valuation set and
- if for every sentence X, the sentence **B**X is in *T* iff X is provable in ***M*** i.e., X $\in$ *P*.

Abbreviations:

- ~X is X$\supset$**f**
- X$\wedge$Y is ~(X$\supset$~Y)
- X$\vee$Y is ~X$\supset$Y
- X$\equiv$Y is (X$\supset$Y)$\wedge$(Y$\supset$X)

Definitions:

M is of *type 1* if the set of provable elements contains all tautologies and is closed under modus ponens (if X and X⊃Y are both provable, then so is Y).

M is *normal* if for every provable X, the sentence **B**X is also provable.

M is *stable* if **B**X is provable, then X is also provable.

M is *consistent* if **f** is not provable.

Let consis be the sentence ~**Bf**.

A mapping *Q* from sentences to sentences will be called a *Rosser mapping* if for every sentence X, if X is provable, then so is *QX*, and if X is provable, then so is ~*QX*.

M is of *type 4* if for any sentences X and Y, the following conditions hold.

1. If X is provable, then so is **B**X (***M*** is normal).
2. **B**(X⊃Y)⊃(**B**X⊃**B**Y) is provable in ***M***.
3. **B**X⊃**BB**X is provable in ***M***.

Theorem 1 - After Tarski-Gödel. Suppose there exists a truth set T for ***M*** such that every provable element is in T, and suppose X is an element such that X≡~**B**X i is in T. Then nether X nor ~X is provable in ***M*** (yet X∈ T).

Theorem 2 - After Gödel. Suppose ***M*** is a normal system of type 1 and G is a sentence such that G≡ ~**B**G is provable in ***M.*** Then

1. If G is provable in ***M***, then ***M*** is inconsistent.
2. If ~G is provable in ***M***, then ***M*** is either inconsistent or unstable.

Theorem 3 - After Rosser. Suppose ***M*** is a system of type 1 and *Q* is a Rosser mapping for ***M***. Then for any sentence X, if X≡~*Q*X is provable in ***M*** and ***M*** is consistent, then neither X nor ~X is provable in ***M.***

Theorem 4 - After Gödel's Second Theorem. Suppose ***M*** is of type 4, and there is a sentence G such that G≡~**B**G is provable in ***M*** . Then if ***M*** is consistent, the sentence consis (i.e., the sentence ~**Bf**) is not provable in ***M***.

Theorem 5 - After Löb. Suppose ***M*** is of type 4, **B**X⊃X is provable in ***M,*** and there is a sentence Y such that Y≡(**B**Y⊃X) is provable in ***M***. Then X is provable in ***M***.

References

Smullyan, Raymond
Godel's Incompleteness Theorems. Oxford Logic Guides. 19. Oxford University Press.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Analytic Proof Methodology

Connections

- Related to:
- Prerequistes:
- Requisite for:

Analytic methods construct proofs by focusing on the semantics (meaning) of formulas rather then their syntax.

The method takes formulas apart and searches for contradictions among the resulting subformulas. Thus analytic methods are associated with refutation style theorem proving. The compound formulas (with the exception of the negation of an atomic formula) are classified as of type **alpha** with subformulas **alpha**$_1$ and **alpha**$_2$, type **beta** with subformulas **beta**$_1$ and **beta**$_2$, type **gamma**, or of type **delta.** The classification scheme for fomulas of classical first-order logic is summarized in Figure 1. The classification can also be applied to modal logics. Analytic methods are utilized the the tableaux method and in sequent systems.

Analytic Classification of Formulas

Figure 1 lists the anlytical properties of the classical logical connectives.

Figure 1: **Alpha, Beta, Gamma and Delta Formulas**

And	**alpha**	**alpha$_1$**	**alpha$_2$**
	¬ ¬A	A	A
	/\AB	A	B
	¬ \/AB	¬A	¬B
	¬->AB	A	¬B
Or	**beta**	**beta$_1$**	**beta$_2$**
	\/AB	A	B
	¬ /\AB	¬A	¬B
	->AB	¬A	B
	<->AB	/\AB	/\¬A¬B
	¬<->AB	/\¬AB	/\A¬B

Universal	**gamma**	**gamma(c)**
	/\x.A	$[A]^x_c$
	¬ \/x.A	¬ $[A]^x_c$
Existential	**delta**	**delta(d)**
	\/x.A	$[A]^x_d$
	¬ /\x.A	¬ $[A]^x_d$

The classification of the modal operators depends on the underlying model.

Definition: By a **Hintikka (downward saturated) set** we mean a set S such that the following conditions hold for every formula of type alpha, beta, gamma, and delta in S.

1. No *atomic* formula and its negation are both in S.
2. If alpha is in S, then both $alpha_1$ and $alpha_2$ are in S.
3. If beta is in S, then either $beta_1$ is in S or $beta_2$ is in S.
4. If gamma is in S, then for every c, gamma(c) is in S.
5. If delta is in S, then for some d, delta(d) is in S.

Downward saturated sets are guaranteed to be coherent and consistent. The construction of downward saturated sets is a purely syntactic procedure which produces a semantic truth assignment (truth function) for the set.

Lemma: (*Hintikka's lemma for first-order logic*) Every Hintikka set S is satisfiable.
Proof: A valuation function is easily constructed from the Hintikka set. The valuation function maps all atomic formula S to **t** and those not appearing in the set to **f**. The construction rules follow the rules for satisfiability. QED.

Last Modified - . Comments and content invited aabyan@wwc.edu

The Method of Analytic Tableaux

Connections

- Related to: Classical logic
- Prerequisites: Overivew, Analytic properties
- Requisite for:

The method of analytic tableaux builds a proof tree using the analytic properties of formulas which involves replacing a compound formula with one or more subformulas. The the proof terminates when a contradiction is found. Thus, like resolution, the method is based on refutation but is interesting because it builds a model of the formula under proof.

Tableau Construction

The **tableau method** is a *backward-chaining* proof search method. The tableau is a tree of with sets of formulas (a block) at each node and leaf. The construction begins with a set of formulas placed at the root of the tree (the negation of the theorem to be proved is placed in the set of formulas). The tree is extended by adding a new block as required by one of four reduction rules. The construction of a branch is terminated when a contradictory block is constructed or when no reduction rule applies. The construction of the tree is terminated when all branches are terminated.

We use the following conventions:

- p, q denote atomic formulas
- P, Q, and R denote formulas
- X, Y, and Z denote sets of formulas
- X, Y stands for X u Y and X, P stands for X u {P}
- Lit stands for a set of literal formulas - atomic formulas and negations of atomic formulas.

In addition, we assume (though it is not necessary) that formulas are in negation normal form.

The form of the tableau rules for extending a branch, creating a new branch, and terminating a branch are given in Figure 1:

Figure 1: **Tableau construction**

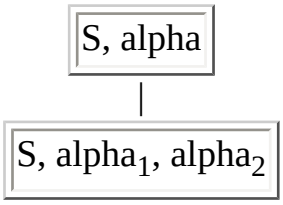
Linear Extension:	Current Block | Child Block
Branching Extension:	Current Block / \\ Left Branch Right Branch
Termination:	Lit or Current Block, p, ¬p

Each reduction rule corresponds to one of the analytic properties. Given a block of formulas containing a formula of type **alpha**, **beta**, **gamma**, or **delta** the reduction rules specify the replacement of a block with one or more blocks in which the formula is replaced with its subformulas. For example, **Rule A** permits the replacement of a conjunction with the conjuncts and **Rule B** requires the block to be replaced with two blocks each containing one of the disjuncts.

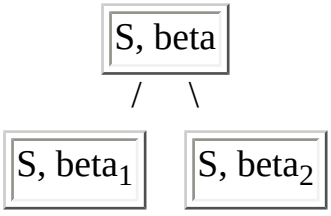
By a **block tableau** for a finite set, **Fs**, of formulas, we mean a tree constructed by placing the set **Fs** at the root, and then continuing according to the following rules:

Figure 2: **Block Tableau Rules**

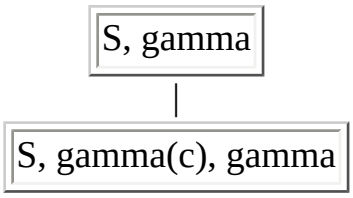
Rule A:



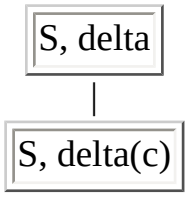
Rule B:



Rule C:



Rule D:



where d is new to the tree

Definition:

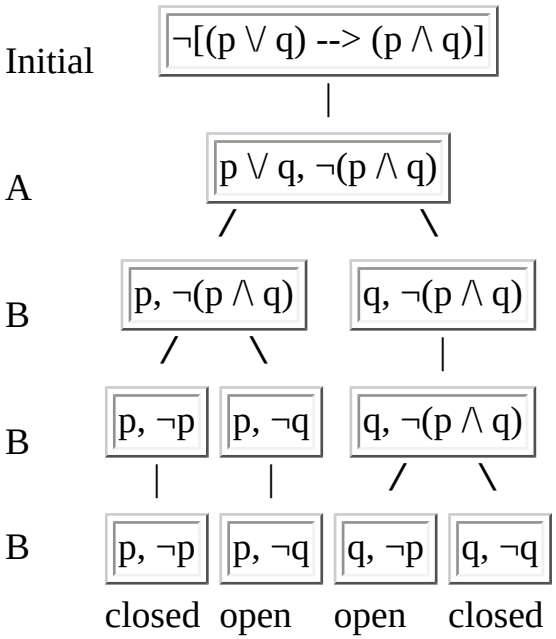
- A *path* in tableau is **closed/contradictory** if *a block on the path contains a formula and its*

negation.

- A *path* in tableau is **open** if no block on the path contains a formula and its negation.
- A *tableau* is **contradictory** if *every path is contradictory*.
- A **proof** of A from a set of formulas Ss, Ss |- A, is a contradictory tableau from [¬A | Ss].

Example

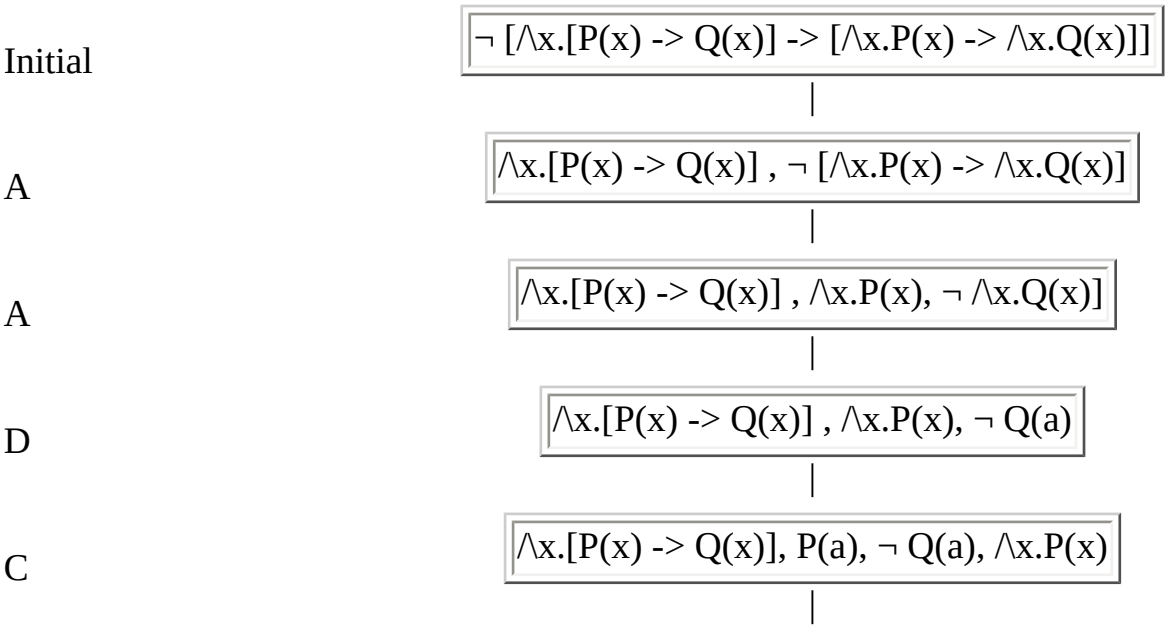
Figure 3: **Tableau for [(p ∨ q) --> (p ∧ q)]**

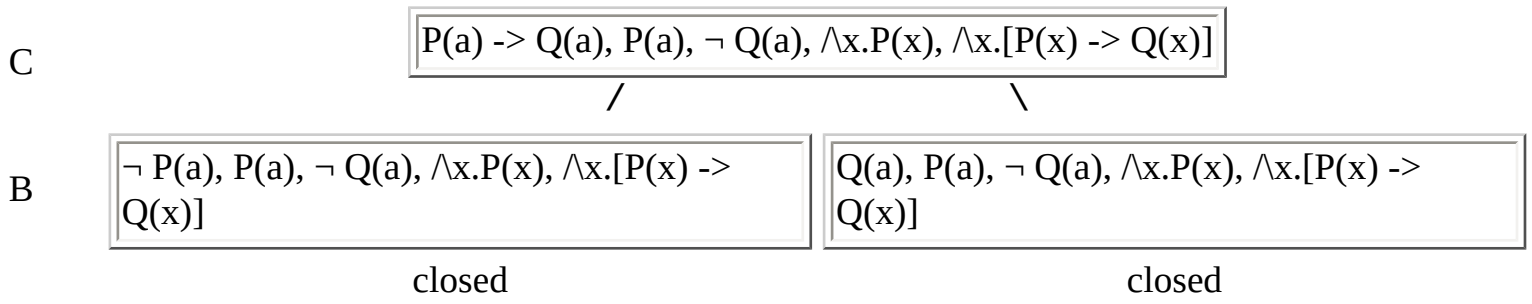


The open blocks provide a model for the formula.

Figure 3 is a tableau proof of A.x.[P(x) -> Q(x)] -> [A.x.P(x) -> A.x.Q(x)].

Figure 4: **Tableau Proof of A.x.[P(x) -> Q(x)] -> [A.x.P(x) -> A.x.Q(x)]**





Since all branches of the tableau are closed, the formula is proved.

For efficiency, apply the rules in the following order:

- rule A,
- rule C (but do not reuse a formula until other rules have been applied),
- rule D,
- rule B, and
- place used gamma formulas last in a list of formulas to be used.

Model Construction

Classical propositional logic has the finite model property - there is a finite set of finite sets of atomic formulas which determine the truth value of a formula. For example the formula ¬ a \/ b is true in either of the two sets in {{¬a}, {b}}. The tableau method can be used to construct these models. If all branches in the tableau are contradictory, the formula is unsatisfiable and any open branch is a model of the formula. An implementation for classical propositional logic and one for propositional modal logic is available.

References

Beckert, Bernhard and Goré, Rajeev
ModLeanTAP: Propositional Modal Logics
Beckert, Bernhard and Posegga, Joachim
LeanTAP - an implementation that uses the negation normal form and Skolem functions.
Fitting, Melvin
Otten, Jen
ileanTAP: an intuitionistic theorem prover
Smullyan, Raymond M.
First-Order Logic Springer-Verlag New York Inc. 1968.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

The Axiomatic Method

Connections

- Related to: Natural deduction, Hilbert style proofs
- Prerequisites:
- Requisite for:

The goal of the axiomatic method is to determine the set of formulas **Thm** (*theorems*) that are derivable from a (usually finite) set ***A*** of formulas called *axioms* by means of *inference rules*. The task of determining whether or not some arbitrary formula ***f*** is a member of **Thm** is called *theorem proving*.

In terms of sets, the set of theorems **Thm** is a subset of formulas ***Fml*** which is a subset of the set of strings ***S**** of some language ***L*** (**Thm** $\subset$ ***Fml*** $\subset$ ***S****) .

The language ***L*** = (S, G) consists of

- S, a set of *symbols* (***S****, is the set of all strings of symbols in S) and
- G, a set of grammar rules (often called formation rules; ***Fml***, is the set of *formulas* defined by the grammar rules).

The set of theorems , **Thm** is constructed incrementally beginning with the set of axioms **A**. A formula is added to **Thm** if it can be derived from the formulas in **Thm** by the application of a *inference rule*. The derived formula is called a *theorem*. A sequence of applications of the inference rules is called a *proof*. The sets of formulas may be ordered as follows:

$$\mathbf{A} \subset \mathbf{Thm} \subset \mathbf{Fml} \subset \mathbf{S}^*$$

If (**Thm = *Fml***), then the axiom system is of little interest and in logic is considered contradictory.

Axiom systems have few inference rules and often many axioms and reason *forward* (or *bottom up*) from axioms to theorems by applications of the inference rules. The disadvantage with forward reasoning is that it gives no insight on how to prove an arbitrary formula, thus requiring (considerable) experience. Proofs, however, are often shorter than those in other reasoning systems.

Substitution
Modus Ponens

In contrast, sequent systems use *backward* (or top-down) reasoning and one axiom and many inference rules.

Classical logic

1

Language - *L* = (C, V, P, F)

Symbols

C = { **f** } u { c_i | i = 0, 1, ... }, a set of constants k_i in C
V = { x_i | i = 0, 1, ... }, a set of variables; x in V
P = { p_0^0, p_0^1, ... , p_1^0, p_1^1, ... , p_2^0, p_2^1, ... , ...}, a set of predicate symbols.

Grammar rules

At = { $p_i^j k_0...k_{j-1}$ | p_i^j in P and $k_0,...,k_{j-1}$ in C }, a set of **atomic formulas**; f in At.
F ::= f |**f** | ->FF -- *propositional formulas*
| /\x.[F]^k_x -- *first-order formulas*
where
Textual substitution, $\mathbf{[F]^k_x}$, is part of the meta language.
For every formula, A and B, predicate symbol, p_i^{j-1}, symbols x and y, and c,
the **substitution** of c for x in A, $\mathbf{[A]^x_c}$ is defined as follows:

- $[p_i^{j-1}v_0,...,v_j]^x_c = p_i^{j-1}[v_0]^x_c, ... , [v_j]^x_c$ where for distinct variables x and y
  - $v_i = x; [x]^x_c = c$
  - $v_i = y; [y]^x_c = y$
  - $v_i = c_k; [y]^x_c = c_k$
- $[\mathbf{f}]^x_c = \mathbf{f}$
- $[->AB]^x_c = ->[A]^x_c[B]^x_c$
- $[/\backslash xA]^x_c = /\backslash x.A$
- $[/\backslash y.A]^x_c = /\backslash y.[A]^x_c$

Abbreviations

- ¬F for ->F**f**
- \/x.F for ¬/\x.¬F

Axioms

1. ->A->BA
2. ->->A->BC->->AB->AC
3. ->¬¬AA
4. ->/\x.A[A]^x_c where x
5. ->/\x.->AB->A/\x.B where x is not free in A.

Inference Rules

1. (*modus ponens*) from A and ->AB infer B
2. (*generalization*) from A, if x is a variable, infer /\x.A.

Exercises

1. Rewrite the axioms in infix form.

2 Hilbert's Formulation

Language - *L* = (C, V, P, F)

Symbols

C = { **f** } u { c_i | i = 0, 1, ... }, a set of constants k_i in C
V = { x_i | i = 0, 1, ... }, a set of variables; x in V
P = { p_0^0, p_0^1, ... , p_1^0, p_1^1, ... , p_2^0, p_2^1, ... , ...}, a set of predicate symbols.

Grammar Rules

At = { $p_i^j k_0...k_{j-1}$ | p_i^j in P and $k_0,...,k_{j-1}$ in C }, a set of **atomic formulas**; f in At.
F ::= f | ¬F | /\FF | \/FF |->FF | <->FF -- *propositional formulas*
| /\x.[F]^k_x -- *first-order formulas*
where
Textual substitution, **[F]$^{\mathbf{k}}_{\mathbf{x}}$**, is part of the meta language.
For every formula, A and B, predicate symbol, p_i^{j-1}, symbols x and y, and c,
the **substitution** of c for x in A, **[A]$^{\mathbf{x}}_{\mathbf{c}}$** is defined as follows:

- $[p_i^{j-1}v_0,...,v_j]^x_c = p_i^{j-1}[v_0]^x_c, ... , [v_j]^x_c$ where for distinct variables x and y
  - $v_i = x$; $[x]^x_c = c$
  - $v_i = y$; $[y]^x_c = y$
  - $v_i = c_k$; $[y]^x_c = c_k$

- $[\mathbf{f}]^x_c = \mathbf{f}$
- $[->AB]^x_c = ->[A]^x_c[B]^x_c$
- $[/\backslash xA]^x_c = /\backslash x.A$
- $[/\backslash y.A]^x_c = /\backslash y.[A]^x_c$

Axioms

1. ->A->BA
2. ->->A->BC->->AB->AC
3. ->/\ABA
4. ->/\ABB
5. ->A->B/\AB
6. ->A\/AB
7. ->B\/AB
8. ->->AC->->BC->\/ABC
9. -><->AB->AB
10. -><->AB->BA
11. ->->AB->->BA<->AB
12. ->->¬A¬B->BA
13. ->/\x.A$[A]^x_c$ where x
14. ->/\x.->AB->A/\x.B where x is not free in A.

Inference Rules

1. (*modus ponens*) from A and ->AB infer B
2. (*generalization*) from A, if x is a variable, infer /\x.A.

Exercises

Rewrite the axioms in infix form.

Intitionistic Logic

Language - *L* = (C, V, P, F)

Symbols

C = { **f** } u { c_i | i = 0, 1, ... }, a set of constants k_i in C
V = { x_i | i = 0, 1, ... }, a set of variables; x in V
P = { p_0^0, p_0^1, ... , p_1^0, p_1^1, ... , p_2^0, p_2^1, ... , ...}, a set of predicate symbols.

Grammar Rules

At = { $p_i^j k_0...k_{j-1}$ | p_i^j in P and $k_0,...,k_{j-1}$ in C }, a set of **atomic formulas**; f in At.
F ::= f |**f** | /\FF | \/FF | ->FF -- *propositional formulas*
| /\x.[F]^k_x | \/x..[F]^k_x-- *first-order formulas*
where
Textual substitution, **[F]$^{\mathbf{k}}_{\mathbf{x}}$** , is part of the meta language.
For every formula, A and B, predicate symbol, p_i^{j-1}, symbols x and y, and c,
the **substitution** of c for x in A, **[A]$^{\mathbf{x}}_{\mathbf{c}}$** is defined as follows:

- $[p_i^{j-1}v_0,...,v_j]^x_c = p_i^{j-1}[v_0]^x_c, ... , [v_j]^x_c$ where for distinct variables x and y
  - $v_i = x; [x]^x_c = c$
  - $v_i = y; [y]^x_c = y$
  - $v_i = c_k; [y]^x_c = c_k$
- $[\mathbf{f}]^x_c = \mathbf{f}$
- $[/\backslash AB]^x_c = ->[A]^x_c[B]^x_c$
- $[\backslash/AB]^x_c = ->[A]^x_c[B]^x_c$
- $[->AB]^x_c = ->[A]^x_c[B]^x_c$
- $[/\backslash xA]^x_c = /\backslash x.A$
- $[/\backslash y.A]^x_c = /\backslash y.[A]^x_c$
- $[\backslash/xA]^x_c = \backslash/x.A$
- $[\backslash/y.A]^x_c = \backslash/y.[A]^x_c$

Abbreviations

- ¬F for ->F**f**
- <->AB for /\->AB->BA

Axioms

1. ->A->BA
2. ->->A->BC->->AB->AC
3. ->/\ABA
4. ->/\ABB

5. ->A->B/\AB
6. ->A\/AB
7. ->B\/AB
8. ->->AC->->BC->\/ABC
9. ->**f**A
10. ->/\x.A[A]$^{x}_{c}$ where x
11. ->[A]$^{x}_{c}$ \/x.A where x
12. ->/\x.->AB->A/\x.B where x is not free in A.
13. ->/\x.->BA->\/x.BA where x is not free in A.

Inference Rules

1. (*modus ponens*) from A and ->AB infer B
2. (*generalization*) from A, if x is a variable, infer /\x.A.

Exercises

1. Rewrite the axioms in infix form.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Free and Bound Variables

Connections

- Related to: Classical logics, substitution, Normal forms (Skolem functions)
- Prerequisites:
- Requisite for:

If A is a formula and x is a variable but not a variable in A then so are: /\x.A and \/x.A.

If A(x) is formed from A by replacing any number of occurrences of some constant c with x. The variable x is said to be **free** in A(x) and is said to be **bound** in /\x.A(x) and \/x.A(x).

- x is *free* in P^i_j (t_1, ... , t_j) iff x is identical with on of t_1, ... , t_j where the t_i are terms.
- x is *free* in ¬A iff x is free in A.
- x is *free* in ->AB iff x is *free* in A or x is *free* in B.
- x is *not free* in /\x.A and is said to be *bound*.
- y is *free* in /\x.A iff y is free in A.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Hilbert Style Proofs

The Hilbert style of proofs is often used in teaching geometry in high school. It consists of the theorem to be proved followed by a sequence of line each of which contains a theorem and a reason why it is a theorem with the last line the theorem being proved. Subproofs may be indented.

Figure 1: **Hilbert Style Proof**

Theorem to be proved:

Steps		**Reasons**
1.		1.
	1.	2.
2.		3.

A |- B
Each step consists of a formula. The corresponding reason is either assumption, instance of a theorem, or an inference rule. The inference rules are those of natural deduction.

The point of a proof is to provide convincing evidence of the correctness of some statement. The following proof formats make clear the intent of the proof as it is read from beginning to end.

Figure : **Proof Formats**

Natural Deduction	**Hilbert Style Proof Format**	
P, P->Q ——— Q	Q 1 P 2 P -> Q	by **Modus Ponens** ... *explanation* ... *explanation*
A |- B ——— A -> B	A->B 1 ¬B ... i ¬A	by **Contrapositive** Assumption ... ... *explanation*
P, Q |- R ——— P/\Q->R	P/\Q->R 1 P 2 Q ... *i* R	by **Deduction** Assumption Assumption ... ... *explanation*

¬P |- Q/\¬Q --- P	P 1 ¬P ... *i* Q/\¬Q	by **Contradiction** Assumption ... ... *explanation*
P |- Q/\¬Q --- ¬P	¬P 1 P ... *i* Q/\¬Q	by **Contradiction** Assumption ... ... *explanation*
P\/Q, P->R, Q->R --- R	R 1 P\/Q 2 P->R 3 Q->R	by **Case analysis** ... *explanation* ... *explanation* ... *explanation*
P->Q, Q->P --- P<->Q	P<->Q 1 P->Q 2 Q->P	By **Mutual implication** ... *explanation* ... *explanation*
P(0), P(n) -> P(n+1) --- /\n.P	/\n.P 1 P(0) 2 P(n) ... *i* P(n+1)	By **Induction** Base step ... *explanation* Assumption (Inductive hypothesis) ... ... *explanation*

References

Last Modified - . Comments and content invited aabyan@wwc.edu

Horn Clause Logic

Connections:

- Related to: Normal forms, Prolog technology, Classical logic
- Prerequisties:
- Requisite for:

Horn clauses are used in the programming language Prolog.

Syntax

Terms -

Figure 1: **Terms**

Symbols

C = { c_0, c_1, c_2... }; the set of constants
X = { x_0, x_1, x_2... }; the set of variables
F = {f_0^0, f_0^1, ... , f_1^0, f_1^1, ... , f_2^0, f_2^1, ... , ...} the set of function symbols

Let

c be a syntactic variable for constants,
x be a syntactic variable for variables, and
f be a syntactic variable for function symbols.

T ::= c|x|f(T,...,T); the set of terms.

Horn clause -

Figure 2: **Horn Clauses**

Symbols and Formulas

P = { p_0^0, p_0^1, ... , p_1^0, p_1^1, ... , p_2^0, p_2^1, ... , ...} a set of predicate symbols

Let p be a syntactic variable for predicate symbols

A = **true**| p(T,...,T) ; a set of atomic formulas

Let a be ***a*** syntactic variable for an atomic formula

G ::= ***a*** | G/\G - the set of goals
D ::= ***a*** | G -> ***a*** | /\x.D - the set of positive Horn clauses

Any formula of classical first-order logic can be translated to a Horn clause formula.

- Put the formula into negation normal form.
- Skolemize (replace existential variables with Skolem constants or Skolem functions of universal variables (from the outside inward). Replace
  - Exists x. P(x) with P(c) where c is new
  - Forall x. ... Exists y. P(y) with Forall x. ... P(f_c(c_k)) where f_c and c_k are new
- Remove the quatifiers.
- Put the formula into conjuctive normal form.

Replace $C_1/\.../\C_n$ with {C_1, ... , C_n}. Each conjunct is of the form: $\neg A_1\/...\/\neg A_m\/B_1\/...\/B_n$ which is equivalent to: $A_1/\.../\A_m->B_1\/...\/B_n$

- If m=0 and n=1 then we have a Prolog fact.
- If m>0 and n=1 then we have a Prolog rule.
- If m>0 and n=0 then we have a Prolog query.

If n always is 1 then the logic is called Horn Clause Logic which is equivalent in computational power to the Universal Turing Machine.

Finally, replace each conjunct $A_1/\.../\A_m -> B_1\/...\/B_n$ with { $A_1/\.../\A_m-> B_1$, $A_1/\.../\A_m-> B_2$, ... $A_1/\.../\A_m-> B_n$ }.

Sequents

From the point of view of sequents,

Figure 3: **Sequent for Horn Clause Logic**

Rules	**[*true formulas* |- *false formulas*]**	
Axiom	[U, A |- A]	leaf node
Negation		
Rule A		
Rule B	$\dfrac{[U \vdash A \wedge B]}{[U \vdash A],\ [U \vdash B]}$	$\dfrac{[\ U, A \to B \vdash B\ ]}{[\ U, A \to B \vdash A]}$ MGU
Rule C		
Rule D		

An implementation is available.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Tableau Rules for Modal Logic

Connections

- Related to: Temporal logic
- Prerequisites: Analytic proof style, Analytic tableaux, Modal Logic
- Requisite for:

In addition to the tableau rules for extending a branch and creating a new branch, modal logic adds several rules. We use the following conventions:

- p, q denote atomic propositions
- P, Q, and R denote formulas
- X, Y, and Z denote sets of formulas
- X, Y stands for X u Y and X, P stands for X u {P}
- []X stands for { []P | P in X }
- <>Y stands for { [<>P | P in Y }
- <>{P_1,...,P_n} stands for {<>P_i|i = 1,...,n} i.e., <>Y
- Lit stands for a set of literal formulas - atomic propositions and negations of atomic propositions.

In addition, we assume that formulas are in negation normal form. For modal logic, we use a block with three sets of formulas:

General formulas; [] Formulas; <> Formulas

The initial block consists of the formula to be proved:

Formula; {}; {}

The tableau rules for the modal logic K are in Figure 1.

Figure 1: **Propositional Modal Logic Tableau Rules**

Alpha:

$$\frac{\text{S, Alpha; []X; <>Y}}{\text{S, alpha}_1\text{, alpha}_2\text{; []X; <>Y}}$$

Beta: $\dfrac{S, beta;\ []X; <>Y}{S, beta_1;\ []X; <>Y \mid S, beta_2;\ []X; <>Y}$

[]: $\dfrac{S, []A; []X; <>Y}{S; []X, []A; <>Y}$

<>: $\dfrac{S, <>A; []X; <>Y}{S; []X; <>Y, <>A}$

New Worlds: $\dfrac{Lit; []X; \{\}}{X}$ $\qquad$ $\dfrac{Lit; []X; <>\{P_1,...,P_n\}}{X, P_1; \{\}; \{\} \mid ... \mid X, P_n; \{\}; \{\}}$

The accessibility relation for the modal logic S4 is reflexive and transitive.

Figure 2: **Tableau rules for S4**
(acessibility is reflexive and transitive)

[]: $\dfrac{S, []A; []X; <>Y}{S, A; []X, []A; <>Y}$

<>: $\dfrac{S, <>A; []X; <>Y}{S, A; []X; <>Y \mid S; []X; <>Y, <>A}$

New World: $Lit; []X;\{\}$ $\qquad$ $\dfrac{Lit; []X; <>\{P_1,...,P_n\}}{X, P_1; []X; \{\} \mid ... \mid X, P_n; []X; \{\}}$

The reflexive nature of the accessibility function is seen in the []- and <>- rules while transitivity is seen in the new world rule. Cycles are possible in S4 with formulas such as []<>p

Temporal logics require that the accessibility relation be reflexive, transitive, and serial. The corresponding tableau rules are given in Figure 2. Notice that the <>-rule is now a braching rule.

Figure 3: **Tableau rules for Temporal Logic**
(S4 + serial)

[]: $\dfrac{S, []A; []X; <>Y}{S, A; []X, []A; <>Y}$

<>: $\dfrac{S, \langle\rangle A; []X; \langle\rangle Y}{S, A; []X; \langle\rangle Y \mid S; []X; \langle\rangle Y, \langle\rangle A}$

New World: $\dfrac{Lit; []X; \langle\rangle Y}{X, \langle\rangle Y; []X; \{\}}$

The serial nature of temporal logic is seen in the new world rule.

Figure 3: **Tableau rules for Linear Time Temporal Logic ([], <>, 0)**

[]: $\dfrac{S, []A; 0X}{S, A; 0X, 0[]A}$

<>: $\dfrac{S, \langle\rangle A; 0X}{S, A; 0X \mid S; 0X, 0\langle\rangle A}$

0: $\dfrac{S, 0A; 0X}{S; 0X, 0A}$

***U*:** $\dfrac{S, A\,U\,B; 0X}{S,B; 0X \mid S, A; 0X, 0(A\,U\,B)}$

New World: $\dfrac{Lit; 0X}{X}$

The transitivity and serial requirements in temporal logics add additional complexity to both theorem proving and model construction. In either case, an unsatisfiable formula may result in cycling through a sequence of states e.g. [] (p /\ <> ¬p). An implementation is available.

Uniform notation:

And	**alpha**	**$alpha_1$**	**$alpha_2$**
	[] A	A	0[] A
	¬ <>A	¬ A	¬ <>A
Or	**beta**	**$beta_1$**	**$beta_2$**
	<>A	A	0<>A
	¬ [] A	¬A	0¬ [] A
	A *U* B	B	A & 0(A *U* B)

Last Modified - . Comments and content invited aabyan@wwc.edu

Natural Deduction

Natural deduction is an approach to proof using rules that are designed to mirror human patterns of reasoning. There are no axioms, only inference rules. For each logical connective, there are two kinds of rules:

- Each **introduction rule** answers the question, 'underwhat conditions can the connective be introduced'.
- Each **elimination rule** answers the question, 'underwhat conditions can the connective be eliminated'.

Figure 1: **Natural Deduction Inference rules**

	Introduction		**Elimination**	
¬	$\dfrac{A \vdash B \wedge \neg B}{\neg A}$		$\dfrac{\neg A \vdash B \wedge \neg B}{A}$	
$\wedge$	$\dfrac{A,\ B}{A \wedge B}$		$\dfrac{A \wedge B}{A}$	$\dfrac{A \wedge B}{B}$
$\vee$	$\dfrac{A}{A \vee B}$	$\dfrac{B}{A \vee B}$	$\dfrac{A \vee B}{A}$	$\dfrac{A \vee B}{B}$
->	$\dfrac{A \vdash B}{A \rightarrow B}$		$\dfrac{A,\ A \rightarrow B}{B}$	
$\forall x.$	$\dfrac{F}{\forall x.F}$		$\dfrac{\forall x.F}{[F]^x_c}$	
$\exists x.$	$\dfrac{F}{\exists x.[F]^c_x}$		$\dfrac{\exists x.F}{[F]^x_c}$	

The nature of many proofs in natural deduction consists of picking apart a logical expression using the elimination rules to get at the constituent parts and then building up new expressions from the constituent parts using the introduction rules. Natural deduction inference rules are used in Hilbert style proofs and sequent systems.

References

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Normal Forms and Skolem Functions

Connections

- Related to: Substitution
- Prerequisites:
- Requisite for:

Normal Forms

Normal forms are based on the expressing formulas in terms of negation, conjunction, disjunction, and the quantifiers, {¬, /\, \/, /\x, \/x}. An implementation is available. See also Horn clause logic.

Negation normal form - NNF

In the negation normal form, negations are attached to atomic formulas. The procedure to convert a formula to negation normal form is to recursively replace formulas appearing on the left with formulas appearing on the right.

A->B	=> ¬A\/B
A<->B	=> A/\B \/ ¬A/\¬B
¬¬ F	=> F
¬(A\/B)	=> ¬A/\¬B
¬(A/\B)	=> ¬A\/¬B
¬(A->B)	=> A/\¬B
¬(A<->B)	=> ¬A/\B \/ A/\¬B
¬/\x.F	=> \/x.¬F
¬\/x.F	=> /\x.¬F
¬ [] F	=> <>¬F
¬<>F	=> []¬F

During the construction of the NNF, a count can be kept of the number of disjunctions in each subformula. This information can be used to rearrange the formula so that the subformula with the fewest number of disjunctions appears on the left.

The rewriting rules are in fact equivalences.

Conjunctive normal form - CNF

A formula in NNF is placed in the conjunctive normal form by recursively moving disjunctions inward and conjunctions outward using the following rewriting rules (which are in fact, equivalences):

A\/(B/\C) => (A\/B) /\ (A\/C)
(A/\B)\/C) => (A\/C) /\ (B\/C)

Disjunctive normal form - DNF

A formula in NNF is placed in the disjunctive normal form by recursively moving conjunctions inward and disjunctions outward using the following rewriting rules (which are in fact equivalences):

A/\(B\/C) => (A/\B) \/ (A/\C)
(A\/B)/\C => (A/\C) \/ (B/\C)

Prenex normal form - PNF

A formula is placed in prenix normal form by recursively moving quantifiers outward so that all quantifiers appear at the beginning of the formula.

Qx¬/\y.F = Qx\/y.¬F
Qx¬\/y.F = Qx/\y.¬F

Qx(/\y.P \/ Q) = Qx/\z. (P(z) \/ Q)
Qx(P \/ /\y.Q) = Qx/\z. (P \/ Q(z))
Qx(\/y.P \/ Q) = Qx\/z. (P(z) \/ Q)
Qx(P \/ \/y.Q) = Qx\/z. (P \/ Q(z))

Qx(/\y.P /\ Q) = Qx/\z. (P(z) /\ Q)
Qx(P /\ /\y.Q) = Qx/\z. (P /\ Q(z))
Qx(\/y.P /\ Q) = Qx\/z. (P(z) /\ Q)
Qx(P /\ \/y.Q) = Qx\/z. (P /\ Q(z))

Qx(/\y.P -> Q) = Qx\/z. (P(z) -> Q)
Qx(P/\ -> /\y.Q) = Qx/\z. (P -> Q(z))
Qx(\/y.P -> Q) = Qx/\z. (P(z) -> Q)
Qx(P/\ -> \/y.Q) = Qx\/z. (P -> Q(z))

where Qx is the list of quantifiers and variables at the beginning of the formula and z does not occur in P or Q or in x.

Skolem Normal Form - SNF

A formula in PNF in which all existential quantifiers precede all universal quantifiers is said to be in *Skolem normal form*.

Skolem Functions

A *Skolem constant* is a new constant that is subtituted for a variable when eliminating an existential quantifier from a formula. In the formula, exist(X, all(Y, F)), the choice of a value for X is *independent* of the choice of a value for Y since once the choice for a value for X is made, it must hold for all choices for Y. In this case, the variable X would be replaced by a Skolem constant c and the formula that results is: all(Y,F(c)).

When an existential quantifier is in the scope of a universal quantifier, the quantified variable must be replaced with a *Skolem function* of the universally quantified variables. While in the formula, all(Y, exists(X, F)), the choice of a value for X is *dependent* on the choice of a value for Y since the form asserts that for each Y there is an appropriate value for X. In this case, the variable X would be replaced with a Skolem function of Y and the formula that results is: all(Y,F(skf(i,Y))).

In either case the choice of a value for Y is independent of the choice of a value for X.

A good choice for Skolemizing a formula can shorten proofs. Some options include, replacing the existentially quantified variable with

- a unique constant,
- liberalized rule: from D we may infer D(c) providing the either
  - c is new or
  - the following 3 conditions all hold
    - c does not occur in D
    - c has not be previously introduced
    - no parameter previously introduced by the rule occurs in D
- a unique function of the free variables occuring in the proof,
- the formula itself.

Skolemization can be done once when a formula is placed into the NNF or whenever existential quantifiers are encountered during a proof.

Theorem: For every formula F in language L, there is a universal formula F' in language L' with function symbols that is statisfiable iff F is satisfiable.

Proof: Assume the formula is in Prenex normal form. The idea is to introduce a new function symbol, f, for each existentially quantified variable, x, which takes as arguments the universally quantified variables preceding x.

Clausal normal form

The clausal normal form is used in logic programming and many theorem proving systems. The procedure to put a formula into clausal form destroys the structure of the formula and often causes exponential blowup in the size of the resulting formula.

The procedure begins with any formula of classical first-order logic

1. Put the formula into negation normal form.
2. Skolemize (replace existential variables with Skolem constants or Skolem functions of universal variables (from the outside inward). Replace
    1. Replace Exists x. P(x) with P(c) where c is new
    2. Forall x. ... Exists y. P(y) with Forall x. ... P(f_c(c_k)) where f_c and c_k are new
3. Remove the quatifiers.
4. Put the formula into conjuctive normal form.

Replace $C_1/\.../\C_n$ with {C_1, ... , C_n}. Each conjunct is of the form: $\neg A_1\/...\/\neg A_m\/B_1\/...\/B_n$ which is equivalent to: $A_1/\.../\A_m$->$B_1\/...\/B_n$

- If m=0 and n=1 then we have a Prolog fact.
- If m>0 and n=1 then we have a Prolog rule.
- If m>0 and n=0 then we have a Prolog query.

If n always is 1 then the logic is called Horn Clause Logic which is equivalent in computational power to the Universal Turing Machine.

Finally, replace each conjunct $A_1/\.../\A_m$ -> $B_1\/...\/B_n$ with { $A_1/\.../\A_m$-> B_1, $A_1/\.../\A_m$-> B_2, ... $A_1/\.../\A_m$-> B_n }.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Prolog Technology for Theorem Proving

Connections:

- Related to:
- Prerequisites:
- Requisite for:

---

1. Variables are implemented using Prolog variables
2. Prolog's unification algorithm must be suplemented with an occurs check
3. Prolog's depth-first search must be replaced with a breadth-first seach using iterative deepening
4. Prolog's pattern maching
5. Substitution may be implemented using `copy_term`/2 or `assert`/`retract`
6. Meta techniques
7. Operator declaration

op(Precedence, Specification, Name)

Precedence = 0,..,1200

Type: specifies position and associativity, x and y represent arguments and f the operator.

Prefix operators: `fx fy`
Infix operators: `xfx xfy yfx yfy`
Postfix operators: `xf yf`

`x` - operators of lower precedence
`y` - operators of equal or greater precedence

The position of `y` indicates associativity `yfx` is left associative, `xfy` is right associative.

op(+*Precedence*, +*Type*, +*Name*)
Declare *Name* to be an operator of type *Type* with precedence *Precedence*. *Name* can also be a list of names, in which case all elements of the list are declared to be identical operators. *Precedence* is an integer between 0 and 1200. Precedence 0 removes the declaration. *Type* is one of: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `yfy`, `fy` or `fx`. The `` `f' `` indicates the position of the functor, while `x` and `y` indicate the position of the arguments. `` `y' `` should be interpreted as ``on this

position a term with precedence lower or equal to the precedence of the functor should occur". For `x' the precedence of the argument must be strictly lower. The precedence of a term is 0, unless its principal functor is an operator, in which case the precedence is the precedence of this operator. A term enclosed in brackets (. . .) has precedence 0.

The predefined operators for SWI Prolog are shown. Note that all operators can be redefined by the user.

Precedence	Type	Name
1200	xfx	**-->**, **:-**
1200	fx	**:-**, **?-**
1150	fx	**dynamic**, **multifile**, **module_transparent**, **discontiguous**, **volatile**, **initialization**
1100	xfy	**;**, **|**
1050	xfy	**->**
1000	xfy	**,**
954	xfy	****
900	fy	**\+**, **not**
900	fx	**~**
700	xfx	**<**, **=**, **=..**, **=@=**, **=:=**, **=<**, **==**, **=\=**, **>**, **>=**, **@<**, **@=<**, **@>**, **@>=**, **\=**, **\==**, **is**
600	xfy	**:**
500	yfx	**+**, **-**, **/**, **\/**, **xor**
500	fx	**+**, **-**, **?**, ****
400	yfx	*****, **/**, **//**, **<<**, **>>**, **mod**, **rem**
200	xfx	******
200	xfy	**^**

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Resolution

Connections

- Related to: Horn clause logic
- Prerequisites: Normal forms, Unification
- Requisite for:

Resolution is an inference rule which requires formulas to be in clausal normal form.

Figure 1: **Resolution inference rules**

Unit resolution	$\frac{A \vee B, \neg B}{A}$
Resolution	$\frac{A \vee B, \neg B \vee C}{A \vee C}$
Resolution	$\frac{P_1 \vee P_2 \vee \ldots \vee P_m, \neg P_1 \vee Q_2 \vee \ldots \vee Q_n}{P_2 \vee \ldots \vee P_m \vee Q_2 \vee \ldots \vee Q_n}$
Horn clause	$\frac{\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_m, Q_1 \& Q_2 \& \ldots \& Q_n \rightarrow P_1}{\neg P_2 \vee \ldots \vee \neg P_m \vee \neg Q_1 \vee \ldots \vee \neg Q_n}$

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Semantics

Figure 0: **Syntax and Semantics**

Language	Semantic Function	Model
L	$->_s$	***M***
l in ***L***	s(*l*) = *m*	*m* in ***M***

If ***M*** is a language and a subset of ***L***, then the semantics are called *reduction semantics*.
If ***M*** is a language, then the semantics are called *translation semantics*.
If ***M*** is a mathematical object, then the semantics are called *denotational semantics*.

Last Modified - . Comments and content invited aabyan@wwc.edu

Sequent Systems (Gentzen)

The goal of sequent systems is to determine the set of formulas **T** (*theorems*) that are reduceable to the single *axiom*:

[U, A |- V, A]

by means of *inference rules*. The task of determining whether or not some arbitrary formula ***f*** is a member of **T** is called *theorem proving*.

In terms of sets, the set of theorems **T** is a subset of formulas ***F*** which is a subset of the set of strings ***S**** of some language ***L*** (**T** c ***F*** c ***S****) .

The language ***L*** = (S, G) consists of

- S, a set of *symbols* (***S****, is the set of all strings of symbols in S) and
- G, a set of grammar rules (often called formation rules; ***F***, is the set of *formulas* defined by the grammar rules).

The set of theorems , **T** is constructed incrementally beginning with the axiom **A**. A formula is added to **T** if it can be derived from the formulas in **T** by the application of a *inference rule*. The derived formula is called a *theorem*. A sequence of applications of the inference rules is called a *proof*. The sets of formulas may be ordered as follows:

A c **T** c ***F*** c ***S****

If (**T**=***F***), then the axiom system is of little interest and in logic is condidered contradictory.

Sequent systems have many inference rules and one axiom and reason *backward* (or top-down) from the formula (theorem to be proved) to the axiom. Backward reasoning is also called *goal directed* reasoning. The advantage with backward reasoning is that it suggests directions to look in searching for a proof. The disadvantage is that proofs may be longer than those produced with other methods.

In contrast, the axiomatic method uses *forward* (or bttom-up) reasoning with often many axioms and few inference rules.

A **sequent** is a pair of sets of formulas separated by the turnstyle,

[*U* |- *V*];

alternative notations include [U --> V] and [U => V]. The first element is referred to as the *antecedent* of the sequent and the second element is called its *succedent*. A sequent corresponds to the assertion that if every formula in *U* holds, then some formula in *V* holds. Symbolically,

$$A_1/\.../\A_m \text{ -> } S_1\/...\/S_n.$$

In sequent systems a formula is a theorem if it can be reduced (in a *backwards* manner) by means of a finite number of the inference rules to an instance of the axiom. A proof consists of constructing a finite tree of sequents using inference rules based on the analytic properties of formulas and natural deduction rules. At the root of the tree is the sequent

> [*Assumptions, axioms, and previously proved theorems* |- *Theorem to be proved*].
> The tree is constructed by the application of the rules such as are found in Figure 2. The proof ends if each branch ends with the sequent at the leaf of the form
> [*U, A* |- *V, A*].

which is called an *initial sequent*. Particular sequent systems are charactorized by whether the antecedent and succedent are multisets, sets, sequences or single formulas and the choice of inference rules and initial sequents.

Figure 2: **Sequent Axiom and Inference rules for Classical First-Order Logic**

Axiom:	[U, X |- V, X]	initial sequent (leaf node)
Rules	**[set of *anticedent formulas* |- *set of succedent formulas*]**	
Negation	$\dfrac{[\ U, \neg F \vdash V\ ]}{[\ U \vdash V, F\ ]}$ $\dfrac{[\ U \vdash V, \neg F\ ]}{[\ U, F \vdash V\ ]}$	
Rule A	$\dfrac{[U, \textbf{alpha} \vdash V]}{[U, \textbf{alpha}_1, \textbf{alpha}_2 \vdash V]}$ $\dfrac{[\ U \vdash V, \textbf{beta}\ ]}{[\ U \vdash V, \textbf{beta}_1, \textbf{beta}_2\ ]}$	
Rule B	$\dfrac{[U \vdash V, \textbf{alpha}]}{[U\vdash V, \textbf{alpha}_1], [U\vdash V, \textbf{alpha}_2\ ]}$ $\dfrac{[\ U, \textbf{beta} \vdash V\ ]}{[\ U, \textbf{beta}_1 \vdash V]\ \ [\ U, \textbf{beta}_2 \vdash V\ ]}$	
Rule C	$\dfrac{[\ U, \text{gamma} \vdash V\ ]}{[\ U, \text{gamma}, \text{gamma(c)} \vdash V\ ]}$ $\dfrac{[\ U\vdash V, \text{delta}\ ]}{[\ U\vdash V, \text{delta}, \text{delta(c)}\ ]}$	any c in C
Rule D	$\dfrac{[\ U \vdash V, \text{gamma}\ ]}{[\ U \vdash V, \text{gamma(c)}\ ]}$ $\dfrac{[\ U, \text{delta} \vdash V]}{[\ U, \text{delta(c)} \vdash V]}$	some c in C new to the sequent

An implementation for classical propositional logic is available.
An implementation for classical first-order logic is available.

Proofs using theories (a theory is a set of formulas) are implemented in sequents by placing the theory on the left and the formula to be proved on the right, [Theory |- Formula].

Proof construction

The inference rules may be used to construct forward or backwards (goal oriented) proofs.

Forward proofs

To prove [A |- B], use the rules breakdown and reassemble the formulas on the left until [U,B |- B] is derived.

Goal oriented proofs

To prove [A |- B], work backwards from the goal B. Use left rules to break down formulas on the left and and right rules to break down the formulas on the right. Assumption and contradiction rules terminate branches of the tree. Some rules of thumb:

- In intuitionistic deduction, avoid using the beta right rules before beta left rules.
- Use delta left and gamma right before delta right and gamma left.

Example

Figure 3: **Proof of**

[(A/\B)=>C |- A=>(B=>C)]

[(A/\B)=>C, A |- (B=>C)]

[(A/\B)=>C, A, B |- C]

[C, A, B |- C], **closed**

[A, B |- A/\B, C]

[A, B |- A, C], **closed**

[A, B |- B, C] **closed**

- A sequent calculus for intuitionistic logic. A<=>B = A=>B & B=>A

Figure 4: **Sequent Axiom and Inference rules for Intuitionistic Logic**

Axioms	[U, X |- V, X]	[U, **false** |- U, A]

Rules	***Left Rules*** **[*true formulas* |- *false formulas*]**	**Right Rules** **[*true formulas* |- *false formulas*]**
Negation	[U, ¬F |- V] / [U, ¬F |- V, F]	[U |- V, ¬F] / [U, F |- V, **false**]
And	[U, A & B |- V] / [U, A, B |- V]	[U |- V, A & B] / [U |- V, A], [U |- V, B]
Or	[U, A v B |- V] / [U, A |- V], [U, B|- V]	[U |- V, A v B] / [U |- V, A, B]
=>	[U, A=>B |- V] / [U, B |- V], [U, A=>B |- V, A]	[U |- V, A=>B] / [U, A |- V, B]
/\x	[U, /\x.A |- V] / [U, A(c), /\x.A |- V]	[U|- V, /\x.A] / [U|- V, A(c)]*
\/x	[U, \/x.A |- V] / [U, A(c) |- V]*	[U|- V, \/x.A] / [U |- V, A(c)]

* c is new

Figure 4: **Sequent Axiom and Inference rules for Intuitionistic Logic**

Axiom	[U, X |- V, X]	

Rules	***Left Rules*** **[*true formulas* |- *false formulas*]**	**Right Rules** **[*true formulas* |- *false formulas*]**
Negation	[U, ¬F |- V] / [U |- V, F]	[U |- V, ¬F] / [U, F |- V]
And	[U, A & B |- V] / [U, A, B |- V]	[U |- V, A & B] / [U |- V, A], [U |- V, B]

Or	$\dfrac{[\ U, A \vee B \vdash V\ ]}{[\ U, A \vdash V\ ],\ [\ U, B \vdash V\ ]}$	$\dfrac{[\ U \vdash V, A \vee B\ ]}{[\ U \vdash V, A, B\ ]}$
=>	$\dfrac{[\ U, A \Rightarrow B \vdash V\ ]}{[\ U, B \vdash V\ ],\ [\ U \vdash V, A\ ]}$	$\dfrac{[\ U \vdash V, A \Rightarrow B\ ]}{[\ U, A \vdash V, B\ ]}$
/\x	$\dfrac{[\ U, \forall x.A \vdash V\ ]}{[\ U, A(c), \forall x.A \vdash V\ ]}$	$\dfrac{[\ U \vdash V, \forall x.A\ ]}{[\ U \vdash V, A(c)\ ]^*}$
\/x	$\dfrac{[\ U, \exists x.A \vdash V\ ]}{[\ U, A(c) \vdash V\ ]^*}$	$\dfrac{[\ U \vdash V, \exists x.A\ ]}{[\ U \vdash V, A(c)\ ]}$

* c is new

An implementation for intuitionistic first-order logic is available.

References

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Substitution - alternative notations

Substitution in logic is *textual* substition. It is the inspiration for passing parameters by *name* as found in the programming language Algol-60.

Substitute *e* for *x* in *F*
(replace *x* with *e* in *F*)

$[F]^x_e$

F[e/x] inspired by multiplication and cancellation in fractions

$[F]^e_x$

F[x:=e] inspired by the assignment operation

F[x<-e]

F[e->x]

F[x:e] inspired by definition

$S_e^x F|$

Simultaneous substitution is represented by generalizing the notations.

Last Modified - . Comments and content invited aabyan@wwc.edu

Syntax

The standard logical expressions are read as indicated in Figure 1.

Figure 1: **Syntax: Prefix**

Symbolic Language	**Natural Language**
f, _|_	false
t, ¯|¯	true
¬F, ~F	not F
/\AB, &AB	A and B
\/AB	A or B
->AB	if A then B
<->AB	A if and only if B
/\x.F	for all (each, every, any) x, F
\/x.F	for some (exists) x, F
[ ] F	Necessarily F
<> F	Possibly F
o F	Somehow F

Often, a minimal set of operators is chosen and the other operators are introduced as abbreviations.

- Minimal sets of operators
  - **f**, ->, [] or <>
  - ¬, ->, [] or <>
  - ¬, /\, [] or <>
  - ¬, \/, [] or <>
- Abbreviations
  - ¬F for F -> **f**
  - A -> B for ¬A \/ B - called conditional or implication.
  - A <-> B for (A -> B) /\ (B -> A) or (A /\ B) \/ (¬ A /\ ¬ B) - biconditional or equivalence.
  - \/AB = ->¬AB
  - /\AB = ¬->¬A¬ B
  - <->AB = /\->AB->BA
  - \/x.F = ¬/\x.¬F

- /\x.F = ¬\/x.¬F
- <> F = ¬[]¬F
- [] F = ¬<>¬F

The prefix notation has the advantage of being unambiguous while the *infix notation* has the advantage of being more readable but requires the use of grouping symbols or precedence rules to remove ambiguity.

F ::= P | ¬(F) | (F)/\(F) | (F)\/(F) | (F)->(F) | (F)<->(F) | [] (F) | <> (F)

The following conventions allow a reduction in the number of parentheses:

1. To improve readability we will sometimes use brackets ([,]) and braces ({,}) for grouping in addition to parentheses.
2. We drop the outermost parentheses.
3. If other parentheses are omitted, then the operators are ranked in precedence (from high to low) as follows: ¬, /\, \/.

Functions are often a part of the definition of the syntax of logic. Functions are a convenience as they can be replaced with additional predicates and a longer formula.

Constants, functions and terms.

Figure 2: **Constants, functions and Terms**

Symbols

C = { c_0, c_1, c_2... }; the set of constants
X = { x_0, x_1, x_2... }; the set of variables
F = {f_0^0, f_0^1, ... , f_1^0, f_1^1, ... , f_2^0, f_2^1, ... , ...} the set of function symbols

Let

c be a syntactic variable for constants,
x be a syntactic variable for variables, and
f be a syntactic variable for function symbols.

T ::= ***c***|***x***|***f*** (T,...,T); the set of terms.

Exercises

1. Use truth tables to vidate the following equivalences:
    1. A -> B = ¬A \/ B
    2. A <-> B = (A -> B) /\ (B -> A) = (A /\ B) \/ (¬ A /\ ¬ B)
    3. \/AB = ->~AB
    4. /\AB = ~->~A~ B
    5. <->AB = /\->AB->BA

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Temporal Logics

Connections

- Related to:
- Prerequisites: Formal Systems, Classical, Modal
- Requisite for: Tableau rules for modal logic

Temporal logics are designed to express temporal progression. It is customary to add the operator [] with the interpretation determined by the logic. A second operator <> is the dual of the first i.e. <>A = ¬[]¬A and []A = ¬<>¬A. Figure 1 illustrates some readings of the formulas []A and <>A.

Temporal logic plays an important role in the specification, derivation, and verification of programs as programs may be viewed as progressing through a sequence of states, a new state after each event in the system. They have a particularly useful role in the specification and verification of communication protocols and reactive systems.

Propositional temporal logics have the *finite model property* making them useful for the derivation of programs from formal specifications. The derived model resembles a *finite state machine* however, the model accepts infinite strings and belongs to the class of *w*-automata. Without the addition of additional temporal operators, temporal logic cannot express all regular expressions.

Syntax

Figure 2: **The Syntax**

Symbols and Formulas:

C = { _|_, -|-} The propositional constants.
L = { p_0, p_1, p_2, ...} The propositional letters.
P in C union L
F ::= P | ¬F | /\FF | \/FF | ->FF | **[]**F | **<>**F {The set of formulas}

Axioms and Inference Rules:

T = The tautologies are the axioms

$$\frac{A,\ A \text{-->} B}{B}$$ The inference rule, A & B are formulas

Additional information on syntax is available.

Semantics - Multiple Worlds (Saul Kripke)

Multiple structures can be used to model *modal* formulas.

Figure 3: **Multiple world structure** $U = (W, A)$

A set of relational structures: $W = \{w \mid w \text{ is a relational structure}\}$

An accessibility relation: A a subset of $W \times W$

set of constants in each world is monotonic in the accessibility relation however, the worlds may differ in the atomic formulas that hold for each world. ***U*** is a graph whose edges are labeled with literal formulas (the formulas required to be true by the valuation function). A model $M = (W, A, w, v)$ is a Kripke structure. A Kripke structure where |**W**| = 1 corresponds to traditional logics. A reachability relation that is symmetric (**Axy** implies **Ayx**) implies that the graph is nondirectional. A reachability relation that is transitive (**Axy** and **Ayz** implies **Axz**) can be used to model temporal phenomenon. A reachability relation that is reflexive (**Axx**), symmetric, and transitive, can be used to reason about finite state systems.

The propositional modal logics share with classical propositional logic the finite model property; if a collection of formulas is satisfiable, it is satisfiable in a finite graph.

There are many modal logics. The table that follows illustrates the approach to semantics for modal logics.

Figure 4: **Model -** $M \models F$
where $M = (U, w, v)$

$M \models p$	iff $v(p)$ in w
$M \models \neg A$	iff not $M \models A$
$M \models \text{->}AB$	iff $M \models \neg A$ or $M \models B$
$M \models \neg\text{->}AB$	iff $M \models A$ and $M \models \neg B$
$M \models \mathbf{[]}A$	iff $M' \models A$ for all u such that Awu and $M' = (U, u, v)$
$M \models \mathbf{<>}A$	iff $M' \models A$ for some u such that Awu and $M' = (U, u, v)$
$M \models /\backslash x.F$	iff $M \models [F]^x_c$ for all c in C

$\boldsymbol{M} \models \neg\forall x.F$	iff $\boldsymbol{M} \models [\neg F]^{x}_{c}$ for some c in C

A formula F is **valid** (a **tautology**), |= F, iff for all **w** in **W**, ***M***|= F i.e., F *is true in all possible worlds*.

A formula F is said to be **valid** (|=F) iff it is valid in all models ***M*** (***M*** |= F for all ***M***). A valid formula is called a **tautology**. Predicate Logic (or Predicate Calculus or First-Order Logic) is a generalization of Propositional Logic. Generalization requires the introduction of variables.

Linear time temporal logic is an example of a logic that uses multiple world semantics. Each time increment is represented by a world. The accessiblity relation is reflexive and transitive but not symmetric as we assume that time does not run backwards. For the formula **[]**A, A holds in the current world and in all future worlds and for the formula **<>**A, A holds in either the current world or some future world.

Program specifications in temporal logic:

- Safety properties: **[]**P
- Liveness properties: **<>**P
- Safe-livenes property: **[]**(A->**<>**B)
- The end of time: ¬**[]<>A**

Additional temporal operators include

- OP - next time
- P*U*Q - P until Q

Definition

- A sentence S of L is **valid**, |=S, if it is true in all structures for ***L***.
- A sentence S of L is a **logical consequence** of a set of sentences Ss of L (Ss |= S), if S is true in every structure in which all of the members of Ss are true.
- A set of sentences Ss, is **satisfiable** if there is a structure *A* in which all of the members of Ss are true. Such a structure is called a **model** of Ss. If Ss has no model, it is **unsatisfiable**.

Proofs in classical logic concern truth in a single state while proofs in modal logics may involve several states. Since a formula may refer to a state other than the one in which it appears, once the collection of states has been constructed, the states must be checked to determine that all such references are satisfied.

An implementation for propositional modal logic is available.
An implementation for first-order modal logic is available.

Proof Theory

In classical logic, the idea was to systematically search for a structure agreeing with the starting sentences. The result being that we get such a structure or each possible analysis leads to a contradiction. In modal logic, we try to build a frame agreeing with the sentences or see that all attempts lead to contradictions.

The Accessiblity Relation

Gore 1992 has a wonderful list of axioms, a naming scheme

Figure : **Model operators and the accessibility relation**

M **|= []**A	iff ***M'*** |= A for all ***u*** such that ***Awu*** and ***M'*** = (***U***, ***u***, *v*)
M **|= <>**A	iff ***M'*** |= A for some ***u*** such that ***Awu*** and ***M'*** = (***U***, ***u***, *v*)

Property	**Axiom**	**Tableau rule**
reflexive	**T:** []A => A	
symmetric	**B:** A => []**<>**A	
transitive	**4:** []A => [][]A	
serial	**D:** []A => **<>**A	

Temporal logic and the Next time operator

	Formula		**Recursive definition**
Always A	[]A	=	A /\ 0[]A
Eventually A	**<>**A	=	A \/ 0**<>**A
A Until B	A *U* B	=	B \/ (A /\ 0(A *U* B))

References

Gore Rajeev Prabharkar
Cut-free Sequent and Tableau Systems for Propositional Normal Modal Logics. (1992)
Smullyan, Raymond (1987)
Forever Undecided Alfred A. Knopf Inc.

Author: Anthony A. Aaby
Last Modified - .
Comments and content invited aabyan@wwc.edu

Truth Tables

The following truth tables use two values, 0 to represent false and 1 to represent true.

Negation

A	¬ A
0	1
1	0

Disjunction

A	B	A \/ B
0	0	0
0	1	1
1	0	1
1	1	1

Conjunction

A	B	A /\ B
0	0	0
0	1	0
1	0	0
1	1	1

Implication

A	B	A-> B
0	0	1
0	1	1
1	0	0
1	1	1

Biconditional

A	B	A <-> B
0	0	1
0	1	0
1	0	0
1	1	1

XOR

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

NOR

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0

NAND

A	B	A nand B
0	0	1
0	1	1
1	0	1
1	1	0

Exercises

1. Show that every truth function is generated by a statement form involving the connectives
    1. ¬, /\, and \/, or
    2. /\ and ¬, or
    3. \/ and ¬, or
    4. -> and ¬ or
    5. nor or
    6. nand.
2. Show that the NOR and NAND connectives are the only binary connectives adequate for the construction of all truth functions.
3. Show that each of the following pairs of connectives are not adequate to express all truth functions
    1. ->, \/
    2. ¬, <->
4. Construct three valued truth tables, undefined, false, and true.

5. Construct three valued truth tables, false, intermediate, and true.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Unification

Connections

- Related to: Horn clause logic
- Prerequisites:
- Requisite for:

MGU - most general unifier

Occurs check

Last Modified - . Comments and content invited aabyan@wwc.edu

---

The Logic of Fact, Fiction, Fantasy, & Physics

Anthony Aaby

This is a formalization of the ideas of fact, fiction, fantasy, scientific theory, paranormal ...

In what follows, we assume the language of many sorted infinite valued modal logic and a structure for interpretation of the formulas of the language.

Logic	material on logic
Reality	For the purposes of this paper, reality will be assumed to be a structure in the sense of mathematical logic.
Fact	The correspondence theory of truth sets up a correspondence between a language (consisting of symbols, formulas, and axioms) - a theory and a structure. The correspondence defines the semantics of the theory (which formulas are true and which are false). The true formulas are called facts. This is the standard construction of formal logic.
Fiction	Fiction may be understood as a language with symbols, formulas, and axioms but no correspondence with a structure. The type relationships are substitution instances.
Fantasy	Fantasy extends fiction with type relationships that do not occur in typical structures. For example, people may be able to fly by flapping their hands or breath unassisted under water.
Physics, Scientific theory, & explanations	Formalized scientific theories are theories in the sense of mathematical logic. The correspondence theory and nature (semantics of scientific theories) is one of approximation. Scientific measurements are often approximations. Thus the quality of a scientific theory is determined by conclusions remaining within the limits of experimental error. Numerical analysis is concerned with the determination of amount of error produced by numerical operations given that the initial values contain error.
Allegories & parables,	An allegory or parable is theory with a correspondence between it an a second theory.
analogy	if a.1 corresponds to b.1 then a.2 may correspond to b.2
simile	a is b
metaphor	a is like b

normal	naturally occurring
paranormal	not scientifically explainable phenomenon of a psychological or supernatural nature - real or imagined
natural	in accordance with or determined by nature
supernatural	beyond the visible observable universe

References

Logic

Aaby, Anthony
The Logical Foundations of Computer Science and Mathematics

Literature

Philosophy of science

Introduction to Compilers

A language translator is a program which translates programs from source language into an equivalent program in an object language.

*Keywords and phrases:*source-language, object-language, syntax-directed, compiler, assembler, linker, loader, parser, scanner, top-down, bottom-up, context-free grammar, regular expressions

Introduction

A *computer* constructed from actual physical devices is termed an *actual computer* or *hardware computer*. From the programming point of view, it is the instruction set of the hardware that defines a machine. An operating system is built on top of a machine to manage access to the machine and to provide additional services. The services provided by the operating system constitute another machine, a **virtual machine**.

A programming language provides a set of operations. Thus, for example, it is possible to speak of a Java computer or a Haskell computer. For the programmer, the programming language is the computer; the programming language defines a virtual computer. The virtual machine for Simple consists of a data area which contains the association between variables and values and the program which manipulates the data area.

Figure M.N: Simple's Virtual Machine and Runtime Environment

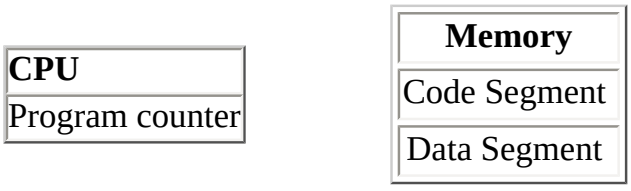


Figure M.N: C's Virtual machine

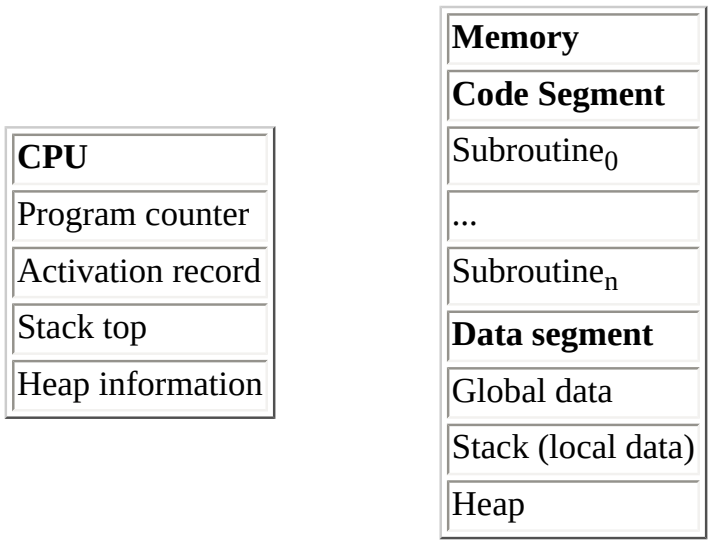
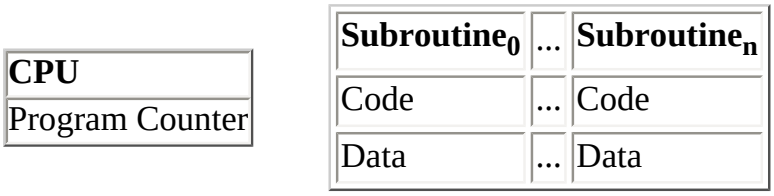


Figure M.N: Nonrecursive language with subroutines



Between the programmer's view of the program and the virtual machine provided by the operating system is another virtual machine. It consists of the data structures and algorithms necessary to support the execution of the program. This virtual machine is the run time system of the language. Its complexity may range in size from virtually nothing, as in the case of FORTRAN, to an extremely sophisticated system supporting memory management and inter process communication as in the case of a concurrent programming language like SR. The run time system for Simple as includes the processing unit capable of executing the code and a data area in which the values assigned to variables are accessed through an offset into the data area.

User programs constitute another class of virtual machines.

A language translator is a program which translates programs from source language into an equivalent program in an object language. The source language is usually a high-level programming language and the object language is usually the machine language of an actual computer. From the pragmatic point of view, the translator defines the semantics of the programming language, it transforms operations specified by the syntax into operations of the computational model---in this case, to some virtual machine. Context-free grammars are used in the construction of language translators. Since the translation is based on the syntax of the source language, the translation is said to be **syntax-directed**.

A **compiler** is a translator whose source language is a high-level language and whose object language is close to the machine language of an actual computer. The typical compiler consists of an analysis phase and a synthesis phase.

In contrast with compilers an **interpreter** is a program which simulates the execution of programs written in a source language. Interpreters may be used either at the source program level or an interpreter may be used it interpret an object code for an idealized machine. This is the case when a compiler generates code for an

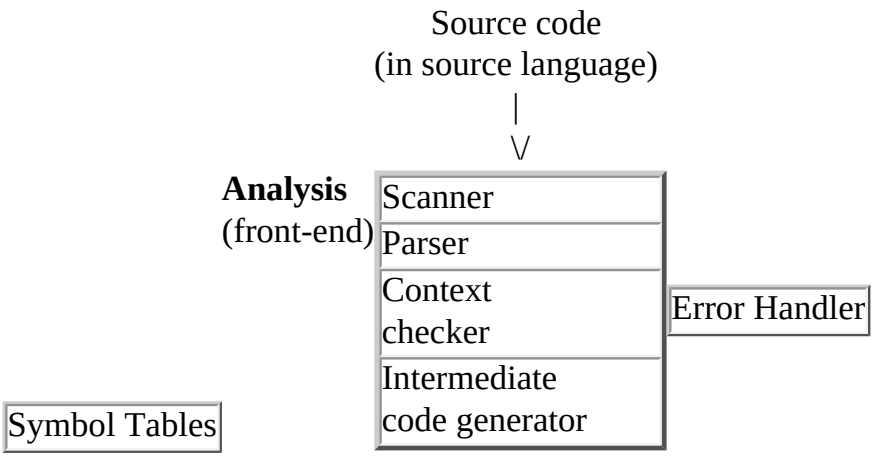
idealized machine whose architecture more closely resembles the source code.

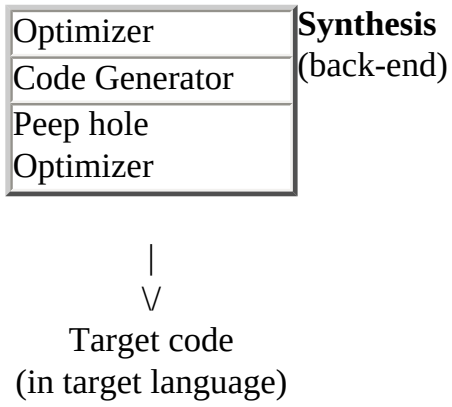
There are several other types of translators that are often used in conjunction with a compiler to facilitate the execution of programs. An **assembler** is a translator whose source language (an assembly language) represents a one-to-one transliteration of the object machine code. Some compilers generate assembly code which is then assembled into machine code by an assembler. A **loader** is a translator whose source and object languages are machine language. The source language programs contain tables of data specifying points in the program which must be modified if the program is to be executed. A **link editor** takes collections of executable programs and links them together for actual execution. A **preprocessor** is a translator whose source language is an extended form of some high-level language and whose object language is the standard form of the high-level language.

The typical compiler consists of several phases each of which passes its output to the next phase

- The *lexical phase* (scanner) groups characters into lexical units or tokens. The input to the lexical phase is a character stream. The output is a stream of tokens. Regular expressions are used to define the tokens recognized by a scanner (or lexical analyzer). The scanner is implemented as a finite state machine.
- The *parser* groups tokens into syntactical units. The output of the parser is a parse tree representation of the program. Context-free grammars are used to define the program structure recognized by a parser. The parser is implemented as a push-down automata.
- The *contextual analysis phase* analyzes the parse tree for context-sensitive information often called the *static semantics*. The output of the contextual analysis phase is an annotated parse tree. Attribute grammars are used to describe the static semantics of a program.
- The *optimizer* applies semantics preserving transformation to the annotated parse tree to simplify the structure of the tree and to facilitate the generation of more efficient code.
- The *code generator* transforms the simplified annotated parse tree into object code using rules which denote the semantics of the source language.
- The *peep-hole* optimizer examines the object code, a few instructions at a time, and attempts to do machine dependent code improvements.

Figure N.1: Traditional Compiler Structure





The Scanner

The **scanner** groups the input stream (of characters) into a stream of tokens (lexeme) and constructs a **symbol table** which is used later for contextual analysis. The lexemes include

- Key words,
- identifiers,
- operators,
- constants: numeric, character, special, and
- comments.

The **lexical phase** (scanner) groups characters into lexical units or tokens. The input to the lexical phase is a character stream. The output is a stream of tokens. Regular expressions are used to define the tokens recognized by a scanner (or lexical analyzer). The scanner is implemented as a finite state machine.

Lex and Flex are tools for generating scanners is C. Flex is a faster version of Lex.

The Parser

The **parser** groups tokens into syntactical units. The output of the parser is a parse tree representation of the program. Context-free grammars are used to define the program structure recognized by a parser. The parser is implemented as a push-down automata.

Yacc and Bison are tools for generating bottom-up parsers in C. Bison is a faster version of Yacc. Jack is a tool for generating scanners and top-down parsers in Java.

Symbol Tables and Error Handlers

In addition to a data stream passing through the phases of the compiler, additional information acquired during a phase may be needed by a later phase. The **symbol table** is used to store the names encountered in the source program and relavant attributes. The information in the symbol table is used by the semantic checker when applying the context-senitive rules and by the code generator. The **error handler** is used to report and recover from errors encountered in the source.

Contextual Checkers

Contextual checkers analyze the parse tree for context-sensitive information often called the *static semantics*. The output of the semantic analysis phase is an annotated parse tree. Attribute grammars are used to describe the static semantics of a program.

This phase is often combined with the paser. During the parse, information concerning variables and other objects is stored in a *symbol table*. The information is utilized to perform the context-sensitive checking.

Intermediate Code Generator

The data structure passed between the analysis and synthesis phases is called the **intermediate representation** (**IR**)of the program. A well designed intermediate representation facilitates the independence of the analysis and syntheses (front- and back-end) phases. Intermedate representations may be

- assembly language like or
- be an abstract syntax tree.

Code Optimizer

Restructuring the parse tree to reduce its size or to present an equivalent tree from which the code generator can produce more efficient code is called **optimization**.

It may be possible to restructure the parse tree to reduce its size or to present a parse to the code generator from which the code generator is able to produce more efficient code. Some optimizations that can be applied to the parse tree are illustrated using source code rather than the parse tree.

- *Constant folding*

```
 I := 4 + J - 5;  --> I := J - 1;
or
 I := 3; J := I + 2;  --> I := 3; J := 5
```

- *Loop-Constant code motion*

```
From:
       while (count < limit) do
          INPUT SALES;
          VALUE := SALES * ( MARK_UP + TAX );
          OUTPUT := VALUE;
          COUNT := COUNT + 1;
       end;  -->
to:
       TEMP :=  MARK_UP + TAX;
       while (COUNT < LIMIT) do
          INPUT SALES;
          VALUE := SALES * TEMP;
          OUTPUT := VALUE;
          COUNT := COUNT + 1;
```

```
end;
```

- *Induction variable elimination* Most program time is spent in the body of loops so loop optimization can result in significant performance improvement. Often the induction variable of a for loop is used only within the loop. In this case, the induction variable may be stored in a register rather than in memory. And when the induction variable of a for loop is referenced only as an array subscript, it may be initialized to the initial address of the array and incremented by only used for address calculation. In such cases, its initial value may be set

```
From:
       For I := 1 to 10 do
          A[I] := A[I] + E
to:
       For I := address of first element in A
             to address of last element in A
             increment by size of an element of A do
          A[I] := A[I] + E
```

- *Common subexpression elimination*

```
From:
      A := 6 * (B+C);
      D := 3 + 7 * (B+C);
      E := A * (B+C);
to:
      TEMP := B + C;
      A    := 6 * TEMP;
      D    := 3 * 7 * TEMP;
      E    := A * TEMP;
```

- *Strength reduction*

```
      2*x  --> x + x
      2*x  --> shift left x
```

- *Mathematical identities*

```
       a*b + a*c --> a*(b+c)
       a - b --> a + ( - b )
```

We do not illustrate an optimizer in the parser for Simp.

Code Generator

The **code generator** transforms the intermediate representation into object code using rules which denote the semantics of the source language. These rules are define a *translation semantics.*

The **code generator**'s task is to translate the intermediate representation to the *native code* of the target

machine. The native code may be an actual executable binary, assembly code or another high-level language. Producing low-level code requires familiarity with such machine level issues such as

- data handling
- machine instruction syntax
- variable allocation
- program layout
- registers
- instruction set

The code generator may be integrated with the parser.

As the source program is processed, it is converted to an internal form. The internal representation in the example is that of an implicit parse tree. Other internal forms may be used which resemble assembly code. The internal form is translated by the code generator into object code. Typically, the object code is a program for a virtual machine. The virtual machine chosen for Simp consists of three segments. A data segment, a code segment and an expression stack.

The data segment contains the values associated with the variables. Each variable is assigned to a location which holds the associated value. Thus, part of the activity of code generation is to associate an address with each variable. The code segment consists of a sequence of operations. Program constants are incorporated in the code segment since their values do not change. The expression stack is a stack which is used to hold intermediate values in the evaluation of expressions. The presence of the expression stack indicates that the virtual machine for Simp is a ``stack machine".

Declaration translation

Declarations define an environment. To reserve space for the data values, the `DATA` instruction is used.

```
integer x,y,z.          DATA 2
```

Statement translation

The assignment, if, while, read and write statements are translated as follows:

Assignment	x := expr		*code for expr*
			STORE X
Conditional	if C then		*code for C*
	S1		BR_FALSE L1
	else		*code for S1*
	S2		BR L2
	end	L1:	*code for S2*
		L2:	

While-do	while C do S	L1:	*code for C* BR_FALSE L2 *code for S* BR L1
		L2:	
Input	read X		IN_INT X
Output	write expr		*code for expr* OUT_INT

If the code is placed in an array, then the label addresses must be *back-patched* into the code when they become available.

Expression translation

Expressions are evaluated on an expression stack. Expressions are translated as follows:

constant LD_INT constant

variable LD variable

e1 op e2 code for e1
code for e2
code for op

Peephole Optimizer

Peephole optimizers scan small segments of the *target code* for standard replacement patterns of inefficient instruction sequences. The peephole optimizer produces machine dependent code improvements.

Figure N.1 contains a context-free grammar for a simple imperative programming language. It will be used to illustrate the concepts in this chapter.

Figure N.2: Context-free grammar for Simple

```
program ::= LET definitions IN command_sequence END

definitions ::= e | INTEGER id_seq IDENTIFIER .

id_seq ::= e | id_seq IDENTIFIER ,

command_sequence ::= e | command_sequence command ;

command := SKIP
```

```
          |  READ IDENTIFIER
          |  WRITE exp
          |  IDENTIFIER := exp
          |  IF exp THEN command_sequence ELSE command_sequence FI
          |  WHILE bool_exp DO command_sequence END

exp ::=  exp + term | exp - term  | term
term :: term * factor | term / factor | factor
factor ::= factor^primary  | primary
primary ::=  NUMBER | IDENT | ( exp )
bool_exp ::=  exp = exp | exp < exp | exp > exp
```

Systematic development of a recursive descent parser

A parser groups sequences of tokens into larger meaningful units described by a context-free grammar. The parser takes as input a stream of tokens where each token contains both the *class* and *spelling* of a token. The stream of tokens is processed sequentially and `currentToken` contains the token of immediate interest. The output of the parser is a syntax tree. The tree may or may not be built explicitly.

There are four steps in the systematic construction of a recursive descent parser.

1. Transform the grammar into proper form.
2. Determine the sets First[E] and Follow[N] for each right-hand side E and non-terminal N of the grammar.
3. Construct parsing procedures from the grammar.
4. Construct the parser.

Figure N.1 summarizes the grammar transformation rules.

Figure N.3: Grammar Transformation Rules

- Convert the grammar to EBNF
- Remove left-recursion: replace N ::= E | NF with N ::= E(F)*
- Left-factor the grammar: replace N ::= EFG | EF'G with N ::= E(F|F')G
- If N ::= E is not recursive, remove it and replace all occurrences of N in the grammar with E

First the grammar is converted to EBNF. The resulting grammar must have a single production rule for each non-terminal symbol. Next, rules containing left recursion are transformed to rules which do not contain left recursion. Left recursion occurs when the same non-terminal appears both at the head of the rule and as a left-most symbol on the right-hand side. The parser can enter an infinite loop if this transformation is not done. Mutual recursion must also be eliminated but it is more difficult. Next, the grammar is simplified by replacing

non-terminals with their defining body. This should be done bottom up, stopping when recursion is encountered. Finally, simplify the grammar by factoring the right-hand sides. This makes it easier for the parser to select the correct grammar rule.

The first and follow sets are used by the parser to select the applicable grammar rule. Figure N.2 summarizes the rules for computing the First and Follow sets.

Figure N.2: First[E] and Follow[N]

First[**e**]	= empty set	
First[**t**]	= {t}	t is a terminal
First[N]	= First[E]	where N ::= E
First[E F]	= First[E] union First[F]	if E generates **lambda**
	= First[E]	otherwise
First[E|F]	= First[E] union First[F]	
First[E*]	= First[E]	
Follow[N]	= {t}	in context Nt, t is terminal
	= First[F]	in context NF, F is non-terminal

The First[E] is the set of terminal symbols that can start a string generated by E. The Follow[N] is the set of terminal symbols that can appear in strings that follow those strings generated by N. The importance of the first and follow sets becomes apparent when the grammar rules are converted to parsing procedures.

Figure N.3 summarizes the rules for converting the EBNF grammar to a collection of parsing procedures.

Figure N.3: EBNF to Parsing Procedures

- For each grammar rule N::=E, construct a parsing procedure

```
parseN {
    parse E
}
```

- Refine *parse E*

If *parse E* is:	**then refine to:**
parse **lambda**	`skip`
parse t	`accept(t)` where t is a terminal
parse N	`parseN` where N is a non-terminal
parse E F	parse E; parse F

parse E|F	`if currentToken.class in First[E] then`
	parse E
	`else if currentToken.class in First[F] then`
	parse F
	`else` *report a syntactic error*

parse E*	`while currentToken.class in First[E] do` *parse E*

If *parse E* is *parse **lambda*** (recall **lambda** is the empty string), then *parse E* is the skip command. If *parse E* is *parse t* (where t is a terminal symbol), then *parse E* is `accept(t)`. If the current token is known to be **t**, then `acceptIt`. If *parse E* is *parse N* (where **N** is a non-terminal), then *parse E* is the call `parseN`. If *parse E* is *parse E F*, then *parse E* is `{`*`parse E; parse F`*`}`. If *parse E* is *parse E*|*F*, then *parse E* is

```
if currentToken.class in First[E] then
        parse E
else if currentToken.class in First[F] then
        parse F
else
        report a syntactic error
```

where `First[E]` and `First[F]` are disjoint. If *parse E* is *parse E**, then *parse E* is

```
while currentToken.class in First[E] do
        parse E
```

where `First[E]` is disjoint from `Follow[E*]`

The parser consists of:

- a global variable `currentToken`;
- auxiliary procedures
  - `scanToken` obtains the next token from the scanner
  - `accept(`*`tc`*`)` which obtains the next token from the scanner if the current token is of the class *tc* else returns a syntactic error. In some instances, the current token is known and then a simplified procedure `acceptIt` may be used. It obtains the next token from the scanner.
- the parsing procedures developed from the grammar;
- a driver `parse` that calls `parseS` (where `S` is the start symbol of the grammar) after having called the scanner to store the first input token in `currentToken`;

```
parse() {
   getChar;
   scanToken;
   parseS;
}
```

Systematic development of a table-driven parser

Given a grammar which satisfies the restrictions specified in the recursive descent parser construction, a table-driven parser may be constructed using the top-down parsing algorithm.

Systematic development of a scanner

A scanner groups sequences of charactors into tokens described by a regular grammar. The scanner takes as input a stream of charactors. The stream of characters is processed sequentially and `currentChar` contains the character of immediate interest. The characters defining a token are collected into a string and the class of the token is identified. The output of the scanner is a stream of tokens. Each token contains information concerning its class and spelling.

There are three steps in the systematic construction of a scanner.

1. Transform the regular expressions into an EBNF grammar.
2. Transcribe each EBNF production rule `N ::= E` to a scanning procedure `scanN`, whose body is determined by `E`.
3. Construct the scanner.

Figure N.M summarizes the rules for tranforming the regular expressions to and EBNF grammar.

Figure N.M: RE to EBNF

- Each regular expression RE_i defining a token class T_i is put into the EBNF form: T_i ::= RE_i.
- A regular expression Sep is constructed defining the symbols which sparate tokens.
- The EBNF production S ::= Sep*(T_0|...|T_n) is added to the grammar.

For each regular expression RE defining a token T, the EBNF rule T ::= RE. A regular expression sep* defining the strings that separate tokens is constructed. And the EBNF production S ::= Sep*(T_0|...|T_n) is defined.

Figure N.3: EBNF to Scanning Procedures

- For each grammar rule T_i::=E_i, construct a scanning procedure `scanT`$_i$ {*scan* E_i}.
- Refine *scan* E_i

***scan* E_i**	**Refinement**
scan **lambda**	`skip`
scan ch	takeIt(t) where ch is a character
scan N	`scanN` where N is a non-terminal
scan E F	scan E; scan F

```
scan E|F    if currentChar in First[E] then
            scan E
            else if currentChar in First[F] then
            scan F
            else report a syntactic error
scan E*     while currentChar in First[E] do
            scan E
```

The scanner is developed from an EBNF grammar (must be non-self embedding) as follows:

1. Objects
   - `currentChar` contains the current character.
   - `currentToken` contains the current token, its spelling and its class.
2. Convert the grammar to EBNF with a single production rule for each non-terminal symbol.
3. The scanner consists of the procedures developed in step (2) enhanced to record the token's class and spelling;
4. a procedure `scanToken` that scans 'separator*Token', and sets `currentToken.spelling` to the charactor string scanned and `currentToken.class` token.
5. the auxiliary procedures
   - `start` sets `currentToken.spelling` to the empty string.
   - `getChar` appends `currentChar` to `currentToken.spelling` and fetches the next character into `currentChar`.
   - `finish` sets `currentToken.class` to the identified class (used for simple disjoint classes)
   - `screen` sets `currentToken.class` to the identified class (used for complex classes that require additional analysis to determine class).

If `currentChar` is part of `currentToken` which is under construction, the procedure `takeIt` adds `currentChar` to `currentToken` and If `currentChar` is not part of `currentToken` which is under construction, the procedure `leaveIt` adds `currentChar` to `currentToken`.

Attribute Grammars and Contextual Constraints

Context-free grammars are not able to completely specify the structure of programming languages. For example, declaration of names before reference, number and type of parameters in procedures and functions, the correspondence between formal and actual parameters, name or structural equivalence, scope rules, and the distinction between identifiers and reserved words are all structural aspects of programming languages which cannot be specified using context-free grammars. These *context-sensitive* aspects of the grammar are often called the *static semantics* of the language. The term *dynamic semantics* is used to refer to semantics proper, that is, the relationship between the syntax and the computational model. Even in a simple language like Simp, context-free grammars are unable to specify that variables appearing in expressions must have an assigned value. Context-free descriptions of syntax are supplemented with natural language descriptions of the static semantics or are extended to become attribute grammars.

Attribute grammars are an extension of context-free grammars which permit the specification of context-sensitive properties of programming languages. Attribute grammars are actually much more powerful and are fully capable of specifying the semantics of programming languages as well.

For an example, the following partial syntax of an imperative programming language requires the declaration of variables before reference to the variables.

```
P ::= D B
D ::= V...
B ::= C ...
C ::= V := E $|$ ...
```

However, this context-free syntax does not indicate this restriction. The declarations define an environment in which the body of the program executes. Attribute grammars permit the explicit description of the environment and its interaction with the body of the program.

Since there is no generally accepted notation for attribute grammars, attribute grammars will be represented as context-free grammars which permit the parameterization of non-terminals and the addition of where statements which provide further restrictions on the parameters. Figure~\ref{ag:decl} is an attribute grammar for declarations.

Figure : An attribute grammar for declarations

```
P ::= D(SymbolTable) B(SymbolTable)
D(SymbolTable)  ::= ...V( insert( V in SymbolTable)...

B(SymbolTable) ::= C(SymbolTable)...
C(SymbolTable) ::= V := E(SymbolTable, Error(if V not in SymbolTable)
                   | ...
```

The parameters marked with $\downarrow$ are called inherited attributes and denote attributes which are passed down the parse tree while the parameters marked with $\uparrow$ are called synthesized attributes and denote attributes which are passed up the parse tree. Attribute grammars have considerable expressive power beyond there use to specify context sensitive portions of the syntax and may be used to specify:

- context sensitive rules
- evaluation of expressions
- translation

Historical Perspectives and Further Reading

For information on compiler construction using Lex and Yacc see\cite{SchFre85}. Pratt \cite{Pratt84} emphasizes virtual machines. ELI, PCCTS, FLEX/BISON, LEX/YACC, Amsterdam Compiler Kit, Jack

Exercises

1. (translation) Construct a translation semantics for
    a. Simple
    b. HTML to TeX/LaTeX
    c. TeX/LaTeX to HTML
2. Construct a scanner and a parser for expressions (use a grammar from chapter 2)
3. Construct an attribute grammar for expressions
4. Construct a calculator using the attribute grammar for expressions.
5. Construct a scanner for Simple
6. Construct a parser for Simple
7. Construct a code generator for Simple
8. Construct an interpreter for Simple
9. Construct an interpreter for BASIC.

The Parser

A parser is a program which determines if its input is syntactically valid and determines its structure. Parsers may be hand written or may be automatically generated by a parser generator from descriptions of valid syntactical structures. The descriptions are in the form of a *context-free grammar*. Parser generators may be used to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

Yacc is a program which given a context-free grammar, constructs a C program which will parse input according to the grammar rules. Yacc was developed by S. C. Johnson an others at AT\&T Bell Laboratories. Yacc provides for semantic stack manipulation and the specification of semantic routines. A input file for Yacc is of the form:

C and parser declarations
%%
Grammar rules and actions
%%
C subroutines

The first section of the Yacc file consists of a list of tokens (other than single characters) that are expected by the parser and the specification of the start symbol of the grammar. This section of the Yacc file may contain specification of the precedence and associativity of operators. This permits greater flexibility in the choice of a context-free grammar. Addition and subtraction are declared to be left associative and of lowest precedence while exponentiation is declared to be right associative and to have the highest precedence.

```
%start program
%token LET INTEGER IN
%token SKIP IF THEN ELSE END WHILE DO READ WRITE
%token NUMBER
%token IDENTIFIER
%left '-' '+'
%left '*' '/'
%right '^'
%%
Grammar rules and actions
%%
C subroutines
```

The second section of the Yacc file consists of the context-free grammar for the language. Productions

are separated by semicolons, the '::=' symbol of the BNF is replaced with ':', the empty production is left empty, non-terminals are written in all lower case, and the multicharacter terminal symbols in all upper case. Notice the simplification of the expression grammar due to the separation of precedence from the grammar.

```
C and parser declarations
%%
program : LET declarations IN commands END
;
declarations : /* empty */
| INTEGER id_seq IDENTIFIER '.'
;
id_seq : /* empty */
| id_seq IDENTIFIER ','
;
commands : /* empty */
| commands command ';'
;
command : SKIP
| READ IDENTIFIER
| WRITE exp
| IDENTIFIER ASSGNOP exp
| IF exp THEN commands ELSE commands FI
| WHILE exp DO commands END
;
exp : NUMBER
| IDENTIFIER
| exp '<' exp
| exp '=' exp
| exp '>' exp
| exp '+' exp
| exp '-' exp
| exp '*' exp
| exp '/' exp
| exp '^' exp
| '(' exp ')'
;
%%
C subroutines
```

The third section of the Yacc file consists of C code. There must be a `main()` routine which calls the function `yyparse()`. The function `yyparse()` is the driver routine for the parser. There must also be the function `yyerror()` which is used to report on errors during the parse. Simple examples of the function `main()` and `yyerror()` are:

```
C and parser declarations
```

```
%%
Grammar rules and actions
%%
main( int argc, char *argv[] )
{ extern FILE *yyin;
++argv; --argc;
yyin = fopen( argv[0], "r" );
yydebug = 1;
errors = 0;
yyparse (); }

yyerror (char *s) /* Called by yyparse on error */
{printf ("%s\n", s);}
```

The parser, as written, has no output however, the parse tree is implicitly constructed during the parse. As the parser executes, it builds an internal representation of the the structure of the program. The internal representation is based on the right hand side of the production rules. When a right hand side is recognized, it is *reduced* to the corresponding left hand side. Parsing is complete when the entire program has been reduced to the start symbol of the grammar.

Compiling the Yacc file with the command `yacc -vd` *file*`.y` (`bison -vd` *file*`.`y) causes the generation of two files *file*`.tab.h` and *file*`.tab.c`. The *file*`.tab.h` contains the list of tokens is included in the file which defines the scanner. The file *file*`.tab.c` defines the C function `yyparse()` which is the parser.

Yacc is distributed with the Unix operating system while Bison is a product of the Free Software Foundation, Inc.

For more information on using Yacc/Bison see the appendex, consult the manual pages for `bison`, the paper *Programming Utilities and Libraries LR Parsing* by A. V. Aho and S. C. Johnson, Computing Surveys, June, 1974 and the document *BISON the Yacc-compatible Parser Generator* by Charles Donnelly and Richard Stallman.

The Scanner

This lecture takes 2 class periods.

A scanner (lexical analyzer) is a program which recognizes patterns in text. Scanners may be hand written or may be automatically generated by a lexical analyzer generator from descriptions of the patterns to be recognized. The descripions are in the form of regular expressions.

Lex is a lexical analyzer generator developed by M. E. Lesk and E. Schmidt of AT&T Bell Laboratories. The input to Lex is a file containing tokens defined using regular expressions. Lex produces an entire scanner module that can be compiled and linked to other compiler modules.

Lex generates a function yylex() which is called to obtain the next token -- an integer denoting the token recognized. Lex calls the function yywrap() at the end of its input. Lex provides the global variable char * yytext which contains the characters of the current token and the global variable int yyleng which is the length of that string. Lex provides extensions to the basic regular expression operators. An input file for Lex is of the form:

```
C declarations, #includes and scanner macros
%%
Token definitions and actions
%%
C subroutines
```

The first section of the Lex file contains the C declaration to include the file (`simple.tab.h`) produced by Yacc/Bison which contains the definitions of the the multi-character tokens. The first section also contains Lex definitions used in the regular expressions. In this case, `DIGIT` is defined to be one of the symbols `0` through `9` and `ID` is defined to be a lower case letter followed by zero or more letters or digits.

```
%{
#include "Simple.tab.h" /* The tokens */
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
Token definitions and actions
%%
C subroutines
```

The second section of the Lex file gives the regular expressions for each token to be recognized and a corresponding action. Strings of one or more digits are recognized as an integer and thus the value `INT` is returned to the parser. The reserved words of the language are strings of lower case letters (upper-case may be used but must be treated differently). Blanks, tabs and newlines are ignored. All other single character symbols are returned as themselves (the scanner places all input in the string `yytext`).

```
C and scanner declarations
%%
":="      { return(ASSGNOP);  }
{DIGIT}+  { return(NUMBER);   }
do        { return(DO);       }
else      { return(ELSE);     }
end       { return(END);      }
fi        { return(FI);       }
if        { return(IF);       }
in        { return(IN);       }
integer   { return(INTEGER);  }
let       { return(LET);      }
read      { return(READ);     }
skip      { return(SKIP);     }
then      { return(THEN);     }
while     { return(WHILE);    }
write     { return(WRITE);    }
{ID}      { return(IDENTIFIER); }
[ \t\n]+ /* blank, tab, new line: eat up whitespace */
.         { return(yytext[0]); }
%%
C subroutines
```

The values associated with the tokens are the integer values that the scanner returns to the parser upon recognizing the token.

Figure M.N gives the format of some of the regular expressions that may be used to define the tokens.

Figure M.N: Lex/Flex Regular Expressions

. any character except newline
x match the character `x'
rs the regular expression r followed by the regular expression s; called ``concatenation"
r|s either an r or an s
(r) match an r; parentheses are used to provide grouping.
r* zero or more r's, where r is any regular expression

r+ one or more r's
[xyz] a ``character class''; in this case, the pattern matches either an 'x', a 'y', or a `z'.
[abj-oZ] a ``character class'' with a range in it; matches an `a', a `b', any letter from `j' through `o', or a `Z'.
{name} the expansion of the ``name'' definition.
X if X is an `a', `b', `f', `n', `r', `t', or `v', then the ANSI-C interpretation of \x.
"[+xyz]+\"+foo" the literal string: [xyz]"foo

There is a global variable `yylval` is accessible by both the scanner and the parser and is used to store additional information about the token.

The third section of the file is empty in this example but may contain C code associated with the actions.

Compiling the Lex file with the command `lex` *file*`.lex` (`flex` *file*`.lex`) results in the production of the file `lex.yy.c` which defines the C function `yylex()`. One each invocation, the function `yylex()` scans the input file an returns the next token.

Lex is distributed with the Unix operating system while Flex is a product of the Free Software Foundation, Inc.

For more information on using Lex/Flex consult the manual pages **lex**, **flex** and **flexdoc**, and see the paper *LEX --Lexical Analyzer Generato*r by M. E. Lesk and E. Schmidt.

Symbol Tables

Lex and Yacc files can be extended to handle the context sensitive information. For example, suppose we want to require that, in Simple, we require that variables be declared before they are referenced. Therefore the parser must be able to compare variable references with the variable declarations.

One way to accomplish this is to construct a list of the variables during the parse of the declaration section and then check variable references against the those on the list. Such a list is called a *symbol table*. Symbol tables may be implemented using lists, trees, and hash-tables.

We modify the Lex file to assign the global variable `yylval` to the identifier string since the information will be needed by the attribute grammar.

The Symbol Table Module

To hold the information required by the attribute grammar we construct a symbol table. A symbol table contains the environmental information concerning the attributes of various programming language constructs. In particular, the type and scope information for each variable. The symbol table will be developed as a module to be included in the yacc/bison file.

The symbol table for Simple consists of a linked list of identifiers, initially empty. Here is the declaration of a node, initialization of the list to empty,

```
struct symrec
{
  char *name;             /* name of symbol          */
  struct symrec *next;    /* link field              */
};
typedef struct symrec symrec;
symrec *sym_table = (symrec *)0;
symrec *putsym ();
symrec *getsym ();
```

and two operations: putsym to put an identifier into the table, and `getsym` which returns a pointer to the symbol table entry corresponding to an identifier.

```
symrec * putsym ( char *sym_name )
{
  symrec *ptr;
  ptr = (symrec *) malloc (sizeof(symrec));
  ptr$->$name = (char *) malloc (strlen(sym_name)+1);
  strcpy (ptr$->$name,sym_name);
  ptr$->$next = (struct symrec *)sym_table;
  sym_table = ptr;
  return ptr;
}

symrec * getsym ( char *sym_name )
{
  symrec *ptr;
  for (ptr = sym_table; ptr != (symrec *) 0;
       ptr = (symrec *)ptr$->$next)
    if (strcmp (ptr$->$name,sym_name) == 0)
      return ptr;
  return 0;
}
```

The Parser Modifications

The Yacc/Bison file is modified to include the symbol table and with routines to perform the installation of an indentifier in the symbol table and to perform context checking.

```
%{
#include $<$stdlib.h$>$       /* For malloc in symbol table */
#include $<$string.h$>$       /* For strcmp in symbol table */
#include $<$stdio.h$>$        /* For error messages         */
#include "ST.h"           /* The Symbol Table Module    */
#define YYDEBUG 1         /* For debugging              */
install ( char *sym_name )
{  symrec *s;
   s = getsym (sym_name);
   if (s == 0)
        s = putsym (sym_name);
   else { errors++;
          printf( "%s is already defined\n", sym_name );
   }
}
context_check( char *sym_name )
{ if ( getsym( sym_name ) == 0 )
     printf( "%s is an undeclared identifier\n", sym_name );
```

```
}
%}
```
Parser declarations
```
%%
```
Grammar rules and actions
```
%%
```
C subroutines

Since the scanner (the Lex file) will be returning identifiers, a semantic record (static semantics) is required to hold the value and `IDENT` is associated with that semantic record.

C declarations
```
%union {                    /* SEMANTIC RECORD            */
char    *id;                /* For returning identifiers  */
}
%token INT SKIP IF THEN ELSE FI WHILE DO END
%token <id> IDENT /* Simple identifier      */
%left '-' '+'
%left '*' '/'
%right '^'
%%
```
Grammar rules and actions
```
%%
```
C subroutines

The context free-grammar is modified to include calls to the install and context checking functions. $*n* is a variable internal to Yacc which refers to the semantic record corresponding the the n-th symbol on the right hand side of a production.

C and parser declarations
```
%%
...
declarations : /* empty */
   | INTEGER id_seq IDENTIFIER '.' { install( $3 ); }
;
id_seq : /* empty */
   | id_seq IDENTIFIER ','  { install( $2 );   }
;
command : SKIP
   | READ IDENTIFIER   { context_check( $2 ); }
   | IDENT ASSGNOP exp { context_check( $2 ); }
...
exp : INT
   | IDENT             { context_check( $1 ); }
...
```

%%
C subroutines

In this implementation the parse tree is implicitly annotated with the information concerning whether a variable is assigned to a value before it is referenced in an expression. The annotations to the parse tree are collected into the symbol table.

The Scanner Modifications

The scanner must be modified to return the literal string associated each identifier (the semantic value of the token). The semantic value is returned in the global variable `yylval`. `yylval`'s type is a union made from the `%union` declaration in the parser file. The semantic value must be stored in the proper member of the union. Since the union declaration is:

```
%union { char *id;
}
```

The semantic value is copied from the global variable `yytext` (which contains the input text) to `yylval.id`. Since the function `strdup` is used (from the library `string.h`) the library must be included. The resulting scanner file is:

```
%{
#include <string.h>     /* for strdup                          */
#include "Simple.tab.h" /* for token definitions and yylval */
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
":="     { return(ASSGNOP);   }
{DIGIT}+ { return(NUMBER);    }
do       { return(DO);        }
else     { return(ELSE);      }
end      { return(END);       }
fi       { return(FI);        }
if       { return(IF);        }
in       { return(IN);        }
integer  { return(INTEGER);   }
let      { return(LET);       }
read     { return(READ);      }
skip     { return(SKIP);      }
then     { return(THEN);      }
while    { return(WHILE);     }
write    { return(WRITE);     }
{ID}     { yylval.id = (char *) strdup(yytext);
```

```
            return(IDENTIFIER);}
[ \t\n]+ /* eat up whitespace */
.        { return(yytext[0]);}
%%
```

Intermediate Representation

Most compilers convert the source code to an intermedate representation during this phase. In this example, the intermediate representation is a parse tree. The parse tree is held in the stack but it could be made explicit. Other popular choices for intermediate representation include abstract parse trees, three-address code, also known as quadruples, and post fix code. In this example we have chosen to bypass the generation of an intermediate representation and go directly to code generation. The principles illustrated in the section on code generation also apply to the generation of intermediate code.

Optimization

This lecture takes 1.5 class periods.

Optimization

It may be possible to restructure the parse tree to reduce its size or to present a parse to the code generator from which the code generator is able to produce more efficient code. Some optimizations that can be applied to the parse tree are illustrated using source code rather than the parse tree.

- Constant folding:

```
 I := 4 + J - 5;  --> I := J - 1;
or
 I := 3; J := I + 2;  --> I := 3; J := 5
```

- Loop-Constant code motion:

```
From:
       while (count < limit) do
          INPUT SALES;
          VALUE := SALES * ( MARK_UP + TAX );
          OUTPUT := VALUE;
          COUNT := COUNT + 1;
       end;  -->
to:
       TEMP :=  MARK_UP + TAX;
       while (COUNT < LIMIT) do
          INPUT SALES;
          VALUE := SALES * TEMP;
          OUTPUT := VALUE;
          COUNT := COUNT + 1;
       end;
```

- Induction variable elimination:
  Most program time is spent in the body of loops so loop optimization can result in significant performance improvement. Often the induction variable of a for loop is used only within the loop. In this case, the induction variable may be stored in a register rather than in memory. And when the induction variable of a for loop is referenced only as an array subscript, it may

be initialized to the initial address of the array and incremented by only used for address calculation. In such cases, its initial value may be set

```
From:
       For I := 1 to 10 do
          A[I] := A[I] + E
to:
       For I := address of first element in A
             to address of last element in A
             increment by size of an element of A do
          A[I] := A[I] + E
```

- Common subexpression elimination:

```
From:
      A := 6 * (B+C);
      D := 3 + 7 * (B+C);
      E := A * (B+C);
to:
      TEMP := B + C;
      A    := 6 * TEMP;
      D    := 3 * 7 * TEMP;
      E    := A * TEMP;
```

- Strength reduction:

```
     2*x  --> x + x
     2*x  --> shift left x
```

- Mathematical identities:

```
      a*b + a*c --> a*(b+c)
      a - b --> a + ( - b )
```

We do not illustrate an optimizer in the parser for Simpile.

Peephole Optimization

Following code generation there are further optimizations that are possible. The code is scanned a few instructions at a time (the peephole) looking for combinations of instructions that may be replaced by more efficient combinations. Typical optimizations performed by a peephole optimizer include copy propagation across register loads and stores, strength reduction in arithmetic operators and memory access, and branch chaining.

We do not illustrate a peephole optimizer for Simp.

```
x := x + 1          ld x        ld x
                    inc         inc
                    store x     dup
y := x + 3          ld x
                    ld 3        ld 3
                    add         add
                    store y     store y
x := x + z          ld x
                    ld z        ld z
                    add         add
                    store x     store x
```

Further Reading

For information on compiler construction using Lex and Yacc see\cite{SchFre85}. Pratt \cite{Pratt84} emphasizes virtual machines.

Virtual Machines

A *computer* constructed from actual physical devices is termed an *actual computer* or *hardware computer*. From the programming point of view, it is the instruction set of the hardware that defines a machine. An operating system is built on top of a machine to manage access to the machine and to provide additional services. The services provided by the operating system constitute another machine, a *virtual machine*.

A programming language provides a set of operations. Thus, for example, it is possible to speak of a Pascal computer or a Scheme computer. For the programmer, the programming language is the computer; the programming language defines a virtual computer. The virtual machine for Simple consists of a data area which contains the association between variables and values and the program which manipulates the data area.

Between the programmer's view of the program and the virtual machine provided by the operating system is another virtual machine. It consists of the data structures and algorithms necessary to support the execution of the program. This virtual machine is the run time system of the language. Its complexity may range in size from virtually nothing, as in the case of FORTRAN, to an extremely sophisticated system supporting memory management and inter process communication as in the case of a concurrent programming language like SR. The run time system for Simple as includes the processing unit capable of executing the code and a data area in which the values assigned to variables are accessed through an offset into the data area.

User programs constitute another class of virtual machines.

A Stack Machine

The S-machine* is a stack machine organized to simplify the implementation of block structured languages. It provides dynamic storage allocation through a stack of activation records. The activation records are linked to provide support for static scoping and they contain the context information to support procedures.

Machine Organization

The S-machine consists of two stores, a program store, **C** (organized as an array and is read only), and a data store, **S** (organized as a stack). There are four registers, an instruction register, **IR**, which contains the instruction which is being interpreted, the stack top register, **T,** which contains the address of the top element of the stack, the program address register, **PC**, which contains the address of the next instruction to be fetched for interpretation, and the current activation record register, **AR**, which contains the base address of the activation record of the procedure which is being interpreted. Each location of **C** is capable of holding an instruction. Each location of **S** is capable of holding an address or an integer. Each instruction consists of three fields an operation code and two parameters.

Storage	*Registers*	*Instruction*
C - code **S** - stack	**IR** - instruction register **T** - stack top pointer **PC** - program counter **AR** - activation record pointer	OpCode, arg_1, arg_2

Instruction Set

S-codes are the machine language of the S-machine. S-codes occupy four bytes each. The first byte is the operation code (op-code). There are nine basic S-code instructions, each with a different op-code. The second byte of the S-code instruction contains either 0 or a lexical level offset, or a condition code for the conditional jump instruction. The last two bytes taken as a 16-bit integer form an operand which is a literal value, or a variable offset from a base in the stack, or a S-code instruction location, or an operation number, or a special routine number, depending on the op-code.

The action of each instruction is described using a mixture of English language description and mathematical formalism. The mathematical formalism is used to note changes in values that occur to the registers and the stack of the S-machine. Data access and storage instructions require an offset within the activation record and the level difference between the referencing level and the definition level. Procedure calls require a code address and the level difference between the referencing level and the definition level.

Instruction	**Operands**	**Comments**
LIT	0,N	Load literal value N onto stack: T := T+ 1; S(T) := N
OPR	0,0	Return (from subroutine call)
	0,1	Negate: S(T) := -1*S(T)
	0,2	Add: S(T-1) := S(T-1) + S(T); T := T-1
	0,3	Subtract: S(T-1) := S(T-1) - S(T); T := T-1
	0,4	Multiply: S(T-1) := S(T-1) * S(T); T := T-1
	0,5	Divide: S(T-1) := S(T-1) / S(T); T := T-1
	0,6	undefined
	0,7	Mod: S(T-1) := S(T-1) mod S(T); T := T-1
	0,8	Equal: S(T-1) := if S(T-1) = S(T) then 1 else 0; T:= T-1
	0,9	Not equal: S(T-1) := if S(T-1) <> S(T) then 1 else 0; T:= T-1
	0,10	Less than: S(T-1) := if S(T-1) < S(T) then 1 else 0; T:= T-1
	0,11	Greater than or equal: S(T-1) := if S(T-1) >= S(T) then 1 else 0; T:= T-1
	0,12	Greater than: S(T-1) := if S(T-1) > S(T) then 1 else 0; T:= T-1
	0,13	Less than or equal: S(T-1) := if S(T-1) <= S(T) then 1 else 0; T:= T-1
	0,14	Or: S(T-1) := if(S(T-1) + S(T) > 1 then 1 else 0; T := T-1
	0,15	And: S(T-1) := S(T-1)*S(T); T := T-1
	0,16	Not: S(T) := if S(T) = 0 then 1 else 0
	0,19	Increment: S(T) := S(T)+1
	0,20	Decrement: S(T) := S(T)-1

	0,21	Copy: S(T+1):= S(T); T := T+1
Data Access Operations		
LOD	L,N	Load value of variable at level offset L, base offset N in stack onto top of stack T:= T + 1; S(T):= S(f(L,AR)+N)+3
LOD	255,0	Load byte from memory address which is on top of stack onto top of stack: S(T) := S(S(T))
LODX	L,D	Indexed load: S(T) := S(f(L,AR)+D+S(T)
STO	L,N	Store value on top of stack into variable location at level offset L, base offset N in stack: S(f(L,AR)+N+3):= S(T); T:= T - 1
STO	255,0	Store: S(S(T-1)) := S(T); T:=T-2
STOX	L,D	Indexed store: POP index, POP A, store A at (base of level offset L)+D+index
Control Operations		
CAL	L, N	call PROC or FUNC at S-code location N declared at level offset L: S(T+1):= f(ld,AR); {Static Link} S(T+2):= AR; {Dynamic Link} S(T+3):= P; {Return Address} AR:= T+1; {Activation Record} PC:= N; {Program Counter} T:= T+3 {Stack Top}
JMP	0, N	JUMP: P := N;
JPC	C, N	JUMP: if S(T) = C then P:= N; T:= T-1
CSP	0, 0	CHARACTER Input: T := T+1, S(T) := input;
	0, 1	CHARACTER Output: Output := S(T); T := T-1;
	0, 2	INTEGER Input: T := T+1; S(T) := input
	0, 3	INTEGER Output: Output := S(T); T := T-1
	0, 8	STRING Output: L := S(T); T := T-1; FOR I := 1 to L DO BEGIN Output := S(T); T := T-1 END

Where the static level difference between the current procedure and the called procedure is ld.
os is the offset within the activation record, ld is the static level difference between the current activation record and the activation record in which the value is to be stored and
f(ld,a) = if i=0 then a else f(i-1,S(a))

Operation

The registers and the stack of the S-machine are initialized as follows:

```
P := 0; {Program Counter}
AR := 0; {Activation Record}
T := 2; {Stack Top}
S[0] := 0; {Static Link}
S[1] := 0; {Static Dynamic Link}
S[2] := 0; {Return Address}
```

The machine repeatedly fetches the instruction at the address in the register P, increments the register P and executes the instruction until the register P contains a zero.

```
execution-loop: I := C(P);
                P := P+1;
                interpret(I);
                if P > 0 -> execution-loop
```

The Stack Machine Module

The implementation of the stack machine is straight forward. The instruction set and the structure of an instruction are defined as follows:

```
/* OPERATIONS: Internal Representation */
enum code_ops { HALT, STORE, JMP_FALSE, GOTO, DATA, LD_INT,
                LD_VAR, READ_INT, WRITE_INT, LT, EQ, GT,
                ADD, SUB, MULT, DIV, PWR };

/* OPERATIONS: External Representation */
char *op_name[] = {"halt", "store", "jmp_false", "goto",
```

```
                        "data", "ld_int", "ld_var", "in_int",
                        "out_int", "lt", "eq", "gt",
                        "add", "sub", "mult", "div", "pwr" };

    struct instruction { enum code_ops op; int arg; };
```

Memory is separtated into two segments, a code segment and a run-time data and expression stack.

```
    struct instruction code[999];
    int stack[999];
```

The definitions of the registers, the program counter {\tt pc}, the instruction register {\tt ir}, the activation record pointer {\tt ar} (which points to be begining of the current activation record), and the pointer to the top of the stack {\tt top}, are straight forward.

```
    int pc = 0;
    tructure instruction ir;
    int ar = 0;
    int top = 0;
```

The fetch-execute cycle repeats until a halt instruction is encountered.

```
    void fetch_execute_cycle()
    { do { /* Fetch */ ir = code[pc++];
           /* Decode & Execute */
           switch (ir.op) {
              case HALT      : printf( "halt\n" ); break;

              case READ_INT  : printf( "Input: " );
                               scanf( "%ld", &stack[ar+ir.arg] );     break;
              case WRITE_INT : printf( "Output: %d\n", stack[top--] );  break;
              case STORE     : stack[ir.arg] = stack[top--];            break;
              case JMP_FALSE : if ( stack[top--] == 0 ) pc = ir.arg;
                                                                        break;
              case GOTO      : pc = ir.arg;                             break;
              case DATA      : top = top + ir.arg;                      break;
              case LD_INT    : stack[++top] = ir.arg;                   break;
              case LD_VAR    : stack[++top] = stack[ar+ir.arg];         break;
              case LT        : if ( stack[top-1] < stack[top] )
                                    stack[--top] = 1;
                               else stack[--top] = 0;                   break;
              case EQ   : if ( stack[top-1] == stack[top] ) stack[--top] = 1; else stack[-
    -top] = 0;        break;
              case GT   : if ( stack[top-1] > stack[top] )  stack[--top] = 1; else stack[-
    -top] = 0; top--; break;
              case ADD  : stack[top-1] = stack[top-1] + stack[top]; top--; break;
              case SUB  : stack[top-1] = stack[top-1] - stack[top]; top--; break;
              case MULT : stack[top-1] = stack[top-1] * stack[top]; top--; break;
              case DIV  : stack[top-1] = stack[top-1] / stack[top]; top--; break;
              case PWR  : stack[top-1] = stack[top-1] * stack[top]; top--; break;
              default   : printf( "%sInternal Error: Memory Dump\n" ); break; } }
      while (ir.op != HALT); }
```

---

*This is an adaptation of: Niklaus Wirth, *Algorithms + Data Structures = Programs* Prentice-Hall, Englewood Cliffs, N.J., 1976.

See also Wilhelm and Maurer, *Compiler Design* Addison-Wesley 1995 pp. 7-60.

Code Generation

This lecture takes 1.5 class periods.

Introduction

The primary objective of the **code generator** is to convert atoms or syntax trees to instructions.

Architecture
: instruction-set opertions, instruction formats, addressing modes, data formats, CPU registers, I/O instructions, etc (conventional machine language)

front end
: machine independent (lexical and syntactical analysis)

back end
: code generation and optimization

maintainability
: separation of concerns

portability

Machine Architectures

- Zero address architecture (stack machine)
- One address architecture (accumulator machine)
- Two address architecture
- Three address architecture (register machine)

As the source program is processed, it is converted to an internal form. The internal representation in the example is that of an implicit parse tree. Other internal forms may be used which resemble assembly code. The internal form is translated by the code generator into object code. Typically, the object code is a program for a virtual machine. The virtual machine chosen for Simple consists of three segments. A data segment, a code segment and an expression stack.

The data segment contains the values associated with the variables. Each variable is assigned to a location which holds the associated value. Thus, part of the activity of code generation is to associate an address with each variable. The code segment consists of a sequence of operations. Program constants are incorporated in the code segment since their values do not change. The expression stack is a stack which is used to hold intermediate values in the evaluation of expressions. The presence of the expression stack indicates that the virtual machine for Simple is a ``stack machine".

Converting Atoms to Instructions

Each atom class converts to an instruction sequence

Single Pass vs. Multiple Passes

Single pass
: code generation and address resolution integrated with the parser utilizing a fixup table.

Multiple pass
: scan atoms determining label addresses, second scan generates code incorporating label addresses.

Register Allocation

Register allocation is the process of assigning a purpose to a particular register.

Declaration translation

Declarations define an environment. To reserve space for the data values, the {\tt DATA} instruction is used.

```
integer x,y,z.          DATA 2
```

Statement translation

The assignment, if, while, read and write statements are translated as follows:

```
x := expr             code for expr
                      STORE X

if cond then          code for cond
   S1                 BR_FALSE L1
else                  code for S1
   S2                 BR L2
end               L1: code for S2
                  L2:

while cond do     L1: code for cond
   S                  BR_FALSE L2
end                   code for S
                      BR L1
                  L2:

read X                IN_INT X

write expr            code for expr
                      OUT_INT
```

If the code is placed in an array, then the label addresses must be {\em back-patched} into the code when they become available.

Expression translation

Expressions are evaluated on an expression stack. Expressions are translated as follows:

```
constant              LD_INT constant

variable              LD variable

e1 op e2              code for e1
                      code for e2
                      code for op
```

Function translation

Function definition - *prologue, body, epilogue*

- pseudo-instructions to announce the beginning of a function
- label definition for function name
- instructions to adjust the stack pointer
- instructions to save environment
- store instructions to save callee registers including return address register
- Function Body
- instruction to return result
- load instructions to restore callee-saved registers
- an instruction to reset the stack pointer
- a return instruction
- pseudo-instructions as needed to announce the end of the function

The Code Generator Module

The data segment begins with an offset of zero and space is reserved, in the data segment, by calling the function data_location which returns the address of the reserved location.

```
int data_offset = 0;
int data_location() { return data_offset++; }

}
```

The code segment begins with an offset of zero. Space is reserved, in the code segment, by calling the function `reserve_loc` which returns the address of the reserved location. The function `gen_label` returns the value of the code offset.

```
int code_offset = 0;
int reserve_loc()
{
   return code_offset++;
}
int gen_label()
{
   return code_offset;
}
```

The functions `reserve_loc` and `gen_label` are used for backpatching code.

The functions `gen_code` and `back_patch` are used to generate code. `gen_code` generates code at the current offset while `back_patch` is used to generate code at some previously reserved address.

```
void gen_code( enum code_ops operation, int arg )
{ code[code_offset].op    = operation;
  code[code_offset++].arg = arg;
}
void back_patch( int addr,  enum code_ops operation, int arg  )
{
```

```
  code[addr].op  = operation;
  code[addr].arg = arg;
}

}
```

The Symbol Table Modifications

The symbol table record is extended to contain the offset from the base address of the data segment (the storage area which is to contain the values associated with each variable) and the `putsym` function is extended to place the offset into the record associated with the variable.

```
struct symrec
{
  char *name;                   /* name of symbol      */
  int offset;                   /* data offset         */
  struct symrec *next;          /* link field          */
};
...
symrec * putsym (char *sym_name)
{
  symrec *ptr;
  ptr = (symrec *) malloc (sizeof(symrec));
  ptr->name = (char *) malloc (strlen(sym_name)+1);
  strcpy (ptr->name,sym_name);
  ptr->offset = data_location();
  ptr->next = (struct symrec *)sym_table;
  sym_table = ptr;
  return ptr;
}
...

}
```

The Parser Modifications

As an example of code generation, we extend our Lex and Yacc files for Simple to generate code for a stack machine. First, we must extend the Yacc and Lex files to pass the values of constants from the scanner to the parser. The definition of the semantic record in the Yacc file is modified that the constant may be returned as part of the semantic record. and to hold two label identifiers since two labels will be required for the if and while commands. The token type of `IF` and `WHILE` is `<lbls>` to provide label storage for backpatching. The function `newlblrec` generates the space to hold the labels used in generating code for the If and While statements. The `context_check` routine is extended to generate code.

```
%{#include <stdio.h>      /* For I/O                                 */
#include <stdlib.h>       /* For malloc here and in symbol table     */
#include <string.h>       /* For strcmp in symbol table              */
#include "ST.h"           /* Symbol Table                            */
#include "SM.h"           /* Stack Machine                           */
#include "CG.h"           /* Code Generator                          */
#define  YYDEBUG 1        /* For Debugging                           */
```

```
int  errors;              /* Error Count-incremented in CG, ckd here */
 struct  lbs              /* For labels: if and while                */
 {
  int for_goto;
  int for_jmp_false;
  };
struct lbs * newlblrec()  /* Allocate space for the labels           */
{
  return  (struct lbs *) malloc(sizeof(struct lbs));
}
install ( char *sym_name )
{
   symrec *s;
   s = getsym (sym_name);
   if (s == 0)
        s = putsym (sym_name);
   else { errors++;
          printf( "%s is already defined\n", sym_name );
   }
}
context_check( enum code_ops operation, char *sym_name )
{ symrec *identifier;
  identifier = getsym( sym_name );
  if ( identifier == 0 )
       { errors++;
         printf( "%s", sym_name );
         printf( "%s\n", " is an undeclared identifier"  );
       }
  else gen_code( operation, identifier->offset );
}
%}
%union semrec             /* The Semantic Records         */
{
 int     intval;          /* Integer values               */
 char    *id;             /* Identifiers                  */
 struct  lbs *lbls        /* For backpatching             */
}
%start program
%token <intval>  NUMBER         /* Simple integer                */
%token <id>      IDENTIFIER     /* Simple identifier             */
%token <lbls>    IF WHILE       /* For backpatching labels       */
%token SKIP THEN ELSE FI DO END
%token INTEGER READ WRITE LET IN
%token ASSGNOP
%left '-' '+'
%left '*' '/'
%right '^'
%%
/* Grammar Rules and Actions */
%%
/* C subroutines */

}
```

The parser is extended to generate and assembly code. The code implementing the if and while commands must contain the correct jump addresses. In this example, the jump destinations are labels. Since the destinations are not known until the entire command is processed, *back-patching* of the destination information is required. In this example, the label identifier is generated when it is known that an address is required. The label is placed into the code when its position is known. An alternative solution is to store the code in an array and back-patch actual addresses.

The actions associated with code generation for a stack-machine based architecture are added to the grammar section. The code generated for the declaration section must reserve space for the variables.

```
/* C and Parser declarations */
%%
program : LET
             declarations
          IN           { gen_code( DATA, sym_table->offset );           }
             commands
          END          { gen_code( HALT, 0 ); YYACCEPT;                 }
;
declarations : /* empty */
   | INTEGER id_seq IDENTIFIER '.' { install( $3 );                      }
;
id_seq : /* empty */
   | id_seq IDENTIFIER ','  { install( $2 );                             }
;

}
```

The `IF` and `WHILE` commands require backpatching.

```
commands : /* empty */
   | commands command ';'
;
command : SKIP
   | READ IDENTIFIER   { context_check( READ_INT, $2 );                 }
   | WRITE exp         { gen_code( WRITE_INT, 0 );                      }
   | IDENTIFIER ASSGNOP exp { context_check( STORE, $1 );               }

   | IF exp            { $1 = (struct lbs *) newlblrec();
                         $1->for_jmp_false = reserve_loc();             }
     THEN commands     { $1->for_goto = reserve_loc();                  }
     ELSE              { back_patch( $1->for_jmp_false,
                                     JMP_FALSE,
                                     gen_label() );                     }
          commands
     FI                { back_patch( $1->for_goto, GOTO, gen_label() );  }

   | WHILE             { $1 = (struct lbs *) newlblrec();
                         $1->for_goto = gen_label();                    }
        exp            { $1->for_jmp_false = reserve_loc();             }
     DO
        commands
     END               { gen_code( GOTO, $1->for_goto );
                         back_patch( $1->for_jmp_false,
                                     JMP_FALSE,
```

```
                                        gen_label() );                  }
;

}
```

The code generated for expressions is straight forward.

```
exp : NUMBER           { gen_code( LD_INT, $1 );                        }
   | IDENTIFIER        { context_check( LD_VAR,  $1 );                  }
   | exp '<' exp       { gen_code( LT,   0 );                           }
   | exp '=' exp       { gen_code( EQ,   0 );                           }
   | exp '>' exp       { gen_code( GT,   0 );                           }
   | exp '+' exp       { gen_code( ADD,  0 );                           }
   | exp '-' exp       { gen_code( SUB,  0 );                           }
   | exp '*' exp       { gen_code( MULT, 0 );                           }
   | exp '/' exp       { gen_code( DIV,  0 );                           }
   | exp '^' exp       { gen_code( PWR,  0 );                           }
   | '(' exp ')'
;
%%
/* C subroutines */

}
```

The Scanner Modifications

Then the Lex file is extended to place the value of the constant into the semantic record.

```
%{
#include <string.h>     /* for strdup                        */
#include "simple.tab.h" /* for token definitions and yylval */
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
{DIGIT}+ { yylval.intval = atoi( yytext );
           return(INT);      }
...
{ID}     { yylval.id = (char *) strdup(yytext);
           return(IDENT);    }
[ \t\n]+ /* eat up whitespace */
.        { return(yytext[0]);}
%%

}
```

An Example

To illustrate the code generation capabilities of the compiler, the following are a program in Simple and the resulting stack code generated by the compiler.

A program in Simple

```
let
   integer n,x,n.
in
   read n;
   if n < 10 then x := 1;  else skip; fi;
   while n < 10 do  x := 5*x; n := n+1; end;
   skip;
   write n;
   write x;
end
```

The stack code

```
 0: data         1
 1: in_int       0
 2: ld_var       0
 3: ld_int      10
 4: lt           0
 5: jmp_false    9
 6: ld_int       1
 7: store        1
 8: goto         9
 9: ld_var       0
10: ld_int      10
11: lt           0
12: jmp_false   22
13: ld_int       5
14: ld_var       1
15: mult         0
16: store        1
17: ld_var       0
18: ld_int       1
19: add          0
20: store        0
21: goto         9
22: ld_var       0
23: out_int      0
24: ld_var       1
25: out_int      0
26: halt         0
```

Exercises

The exercises which follow vary in difficulty. In each case, determine what modifications must be made to the grammar, the symbol table and to the stack machine code.

1. Re-implement the symbol table as a binary search tree.
2. Re-implement the symbol table as a hash table.
3. Re-implement the symbol table, the code generator and the stack machine as C++ classes.
4. Extend the Micro Compiler with the extensions listed below. The extensions require the modification of the scanner to handle the new tokens and modifications to the parser to handle the extended grammar.
    1. Declarations: Change the semantic processing of identifier references to require previous declaration.
    2. Real literals and variables: Extend the symbol-table routines to store a type attribute with each identifier. Extend the semantic routines that generate code to consider the types of literals and variables they receive as parameters.
    3. Multiplication and division: Make appropriate changes to the semantic routines to handle code generation based on the new tokens.
    4. **if** and **while** statements: Semantic routines must generate the proper tests and jumps.
    5. Parameterless procedures: The symbol table must be extended to handle nested scopes and the semantic routines must be extended to generate code to manage control transfer at each point of call and at the beginning and end of each procedure body.

    Optional additions include:
    1. An interpreter for the code produced by the compiler
    2. Substitution of a table-driven parser for the recursive descent parser in the Micro compiler.
5. Extend the Micro-II compiler. A self-contained description of Macro is included in the cs360/compiler_tools directory. In brief, the following extensions are required.
    1. Scanner extensions to handle the new tokens, use of parser generator to produce new tables(20 points).
    2. Declarations of integer and real variables(10 points).
    3. Integer literals, expressions involving integers, I/O for integers, and output for strings(10 points).
    4. The **loop** and **exit** statements and addition of the **else**} and **elsif** parts to the **if** statement (20 points).
    5. Recursive procedures with parameters(8 points for simple procedures, 8 points for recursion, 12 points for parameters).
    6. Record declarations and field references(8 points).
    7. Array declarations and element references(12 points).
    8. Package declarations and qualified name references(12 points).

    The total number of points is 120.

6. The compiler is to be completely written from scratch. The list below assigns points to each of the features of the language, with a basic subset required of all students identified first. All of the other features are optional.

   Basic Subset(130 points)

   1. (100 points)
      1. Integer, Real, Boolean types (5 points)
      2. Basic expressions involving Integer, Real and Boolean types (+, -, *, /, **not**, **and**, **or**, **abs**, **mod,** **, <, <=, >, >=, =, /=) (30 points).
      3. Input/Output
         1. Input of Integer, Real, Boolean scalars(5 points).
         2. Output of String literals and Integer, Real and Boolean expressions(excluding formatting)(5 points).
      4. Block structure (including declaration of local variables and constants) (20 points).
      5. Assignment statement (10 points).
      6. **if**, **loop**, and **exit** statements (10, 5, 10 points respectively)
   2. (30 points) Procedures and scalar-valued functions of no arguments (including nesting and non-local variables).

   Optional Features(336 points possible)

   1. **loop** statements (15 points total)
      1. **in** and **in reverse** forms (10 points)
      2. **while** form (5 points)
   2. Arrays (30 points total)
      1. One-dimensional, compile-time bounds, including First and Last attributes (10 points)
      2. Multi-dimensional, compile-time bounds, including First and Last attributes (5-points)
      3. Elaboration-time bounds (9 points)
      4. Subscript checking (3 points)
      5. Record base type (3 points)
   3. Boolean short-circuit operators (**and then**, **or else**) (12 points)
   4. Strings (23 points total)
      1. Basic string operations (string variables, string assigns, all string operators (&, Substr, etc), I/O of strings) (10 points)
      2. Unbounded-length strings (5 points)
      3. Full garbage collection of unbounded-length strings (8 points)
   5. Records (15 points total)
      1. Basic features (10 points)
      2. Fields that are compile-time bounded arrays (2 points)
      3. Fields that are elaboration-time sized (both arrays and records) (3 points)
   6. Procedures and functions(53 points total)
      1. Scalar parameters (15 points)
      2. Array arguments and array-valued functions (compiler-time bounds) (7 points)
      3. Array arguments and array-valued functions (elaboration-time bounds) (5 points)
      4. Record arguments and record-value functions (4 points)
      5. Conformant array parameters (i.e. array declarations of the form {**type array** (T

range <>) of T2}) (8 points)
6. Array-valued functions (elaboration-sized bounds) (3 points)
7. Array-valued functions (conformant bounds) (4 points)
8. Forward definition of procedures and functions (3 points)
9. String arguments and string-valued functions (4 points)

7. **case**}statement (20 points total)
    1. Jump code (10 points)
    2. If-then-else code (4 points)
    3. Search-table code (6 points)
8. Constrained **subtypes** (including First and Last attributes) (10 points total)
    1. Run-time range checks (7 points)
    2. Compile-time range checks (3 points)
9. Folding of scalar constant expressions (8 points)
10. Initialized variables (10 points total).
    1. Compile-time values, global (without run-time code) (3 points)
    2. Compile-time values, local (2 points)
    3. Elaboration-time values (2 points)
    4. Record fields (3 points)
11. Formatted writes (3 points).
12. Enumerations (18 points total).
    1. Declaration of enumeration types; variables, assignment, and comparison operations (9 points)
    2. Input and Output of enumeration values (5 points)
    3. Succ, Pred, Char, and Val attributes (4 points)
13. Arithmetic type conversion (3 points).
14. Qualified names (from blocks and subprograms) (3 points).
15. Pragmata (2 points).
16. Overloading (25 points total)
    1. Subprogram identifier (18 points)
    2. Operators (7 points)
17. Packages (55 points total).
    1. Combined packages (containing both declaration and body parts); qualified access to visible part (20 points)
    2. Split packages (with distinct declaration and body parts) (5 points)
    3. Private types (10 points)
    4. Separate compilation of package bodies (20 points)
18. Use statements (11 points)
19. Exceptions (including **exception** declarations, **raise** statements, exception handlers, predefined exceptions) (20 points).

Extra credit project extensions:

- Language extensions -- array I/O, external procedures, sets, procedures as arguments, extended data types.
- Program optimizations -- eliminating redundant operations, storing frequently used variables or expressions in registers, optimizing Boolean expressions, constant-folding.
- High-quality compile-time and run-time diagnostincs -- ``Syntax error: operator expected", or ``Subscript out of range in line 21; illegal value: 137". Some form of

syntactic error repair might be included.

Lex and Flex

From the Flex man page

In order for Lex/Flex to recognize patterns in text, the pattern must be described by a *regular expression*. The input to Lex/Flex is a machine readable set of regular expressions. The input is in the form of pairs of regular expressions and C code, called rules. Lex/Flex generates as output a C source file, `lex.yy.c`, which defines a routine `yylex()`. This file is compiled and linked with the `-lfl` library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Lex/Flex Examples

The following Lex/Flex input specifies a scanner which whenever it encounters the string ``username'' will replace it with the user's login name:

```
%%
username    printf( "%s", getlogin() );
```

By default, any text not matched by a Lex/Flex scanner is copied to the output, so the net effect of this scanner is to copy its input file to its output with each occurrence of ``username'' expanded. In this input, there is just one rule. ``username'' is the *pattern* and the ``printf'' is the *action*. The ``%%'' marks the beginning of the rules.

Here's another simple example:

```
    int num_lines = 0, num_chars = 0;

%%
\n    ++num_lines; ++num_chars;
.     ++num_chars;

%%
main()
    {
    yylex();
    printf( "# of lines = %d, # of chars = %d\n",
            num_lines, num_chars );
    }
```

This scanner counts the number of characters and the number of lines in its input (it produces no output other than the final report on the counts). The first line declares two globals, `num_lines` and `num_chars`, which are accessible both inside `yylex()` and in the `main()` routine declared after the second "%%". There are two rules, one which matches a newline ("\n") and increments both the line count and the character count, and one which matches any character other than a newline (indicated by the ``." regular expression).

A somewhat more complicated example:

```
/* scanner for a toy Pascal-like language */

%{
/* need this for the call to atof() below */
#include <math.h>
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+    {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }

{DIGIT}+"."{DIGIT}*        {
            printf( "A float: %s (%g)\n", yytext,
                    atof( yytext ) );
            }

if|then|begin|end|procedure|function        {
            printf( "A keyword: %s\n", yytext );
            }

{ID}        printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"   printf( "An operator: %s\n", yytext );

"{"[\^{$\;$}}\n]*"}"     /* eat up one-line comments */

[ \t\n]+          /* eat up whitespace */

.           printf( "Unrecognized character: %s\n", yytext );

%%

main( argc, argv )
```

```
int argc;
char **argv;
{
++argv, --argc;  /* skip over program name */
if ( argc > 0 )
        yyin = fopen( argv[0], "r" );
else
        yyin = stdin;

yylex();
}
```

This is the beginnings of a simple scanner for a language like Pascal. It identifies different types of *tokens* and reports on what it has seen.

The details of this example will be explained in the following sections.

The Lex/Flex Input File

The Lex/Flex input file consists of three sections, separated by a line with just %% in it:

```
definitions
%%
rules
%%
user code
```

The Declarations Section

The *definitions* section contains declarations of simple *name* definitions to simplify the scanner specification.

Name definitions have the form:

```
name definition
```

The ``name'' is a word beginning with a letter or an underscore (`_') followed by zero or more letters, digits, `_', or `-' (dash). The definition is taken to begin at the first non-white-space character following the name and continuing to the end of the line. The definition can subsequently be referred to using ``{name}'', which will expand to ``(definition)''. For example,

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

defines ``DIGIT'' to be a regular expression which matches a single digit, and ``ID'' to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to

```
{DIGIT}+"."{DIGIT}*
```

is identical to

```
([0-9])+"."([0-9])*
```

and matches one-or-more digits followed by a `.' followed by zero-or-more digits.

The Rules Section

The *rules* section of the Lex/Flex input contains a series of rules of the form:

```
pattern action
```

where the pattern must be unindented and the action must begin on the same line.

See below for a further description of patterns and actions.

Finally, the user code section is simply copied to lex.yy.c verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second %% in the input file may be skipped, too.

In the definitions and rules sections, any \underline{indented} text or text enclosed in %{ and %} is copied verbatim to the output (with the %{}'s removed). The %{}'s must appear unindented on lines by themselves.

In the rules section, any indented or %{} text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or %{} text in the rule section is still copied to the output, but its meaning is not well-defined and it may well cause compile-time errors.

In the definitions section, an unindented comment (i.e., a line beginning with ``/*'') is also copied verbatim to the output up to the next ``*/''.

Lex/Flex Patterns

The patterns in the input are written using an extended set of regular expressions. These are:

x match the character `x'

. any character except newline
[xyz] a ``character class''; in this case, the pattern matches either an 'x', a 'y', or a `z'
[abj-oZ] a ``character class'' with a range in it; matches an `a', a `b', any letter from `j' through `o', or a `Z'
[^A-Z] a ``negated character class'', i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.
[^A-Z\n] any character EXCEPT an uppercase letter ora newline
r* zero or more r's, where r is any regular expression
r+ one or more r's
r? zero or one r's (that is, ``an optional r'')
r{2,5} anywhere from two to five r's
r{2,} two or more r's
r{4} exactly 4 r's
{name} the expansion of the ``name'' definition (see above)
`"[xyz]\"foo"` the literal string: `[xyz]"foo`
\X if X is an `a', `b', `f', `n', `r', `t', or `v', then the ANSI-C interpretation of \x. Otherwise, a literal `X' (used to escape operators such as `*')
\0 a NUL character (ASCII code 0)
\123 the character with octal value 123
\x2a the character with hexadecimal value 2a
(r) match an r; parentheses are used to override precedence (see below)
rs the regular expression r followed by the regular expression s; called ``concatenation''
r|s either an r or an s
r/s an r but only if it is followed by an s. The s is not part of the matched text. This type of pattern is called as ``trailing context''.
^r an r, but only at the beginning of a line
r$ an r, but only at the end of a line. Equivalent to ``r/\n''.
<s>r an r, but only in start condition s
<s1,s2,s3>r an r in any of start conditions s1, s2, s2
...

The regular expressions listed above are grouped according to precedence, from highest precedence at the top to lowest at the bottom. Those grouped together have equal precedence. For example,

```
foo|bar*
```

is the same as

```
(foo)|(ba(r*))
```

since the `*' operator has higher precedence than concatenation, and concatenation higher than alternation (`|'). This pattern therefore matches either the string ``foo'' or the string ``ba'' followed by zero-or-more r's. To match ``foo'' or zero-or-more ``bar''s, use:

```
foo|(bar)*
```

and to match zero-or-more ``foo"'s-or-``bar"'s:

```
(foo|bar)*
```

A note on patterns: A negated character class such as the example [^ A-Z] above *will match a newline* unless "\n" (or an equivalent escape sequence) is one of the characters explicitly present in the negated character class (e.g., [^ A-Z\n]). This is unlike how many other regular expression tools treat negated character classes, but unfortunately the inconsistency is historically entrenched. Matching newlines means that a pattern like [^"]* can match an entire input (overflowing the scanner's input buffer) unless there's another quote in the input.

How the Input is Matched

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the Lex/Flex input file is chosen.

Once the match is determined, the text corresponding to the match (called the token) is made available in the global character pointer yytext, and its length in the global integer yyleng. The action corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.

If no match is found, then the default rule is executed: the next character in the input is considered matched and copied to the standard output. Thus, the simplest legal Lex/Flex input is:

```
%%
```

which generates a scanner that simply copies its input (one character at a time) to its output.

Lex/Flex Actions

Each pattern in a rule has a corresponding action, which can be any arbitrary C statement. The pattern ends at the first non-escaped whitespace character; the remainder of the line is its action. If the action is empty, then when the pattern is matched the input token is simply discarded. For example, here is the specification for a program which deletes all occurrences of ``zap me" from its input:

```
%%
"zap me"
```

(It will copy all other characters in the input to the output since they will be matched by the default rule.)

Here is a program which compresses multiple blanks and tabs down to a single blank, and throws away whitespace found at the end of a line:

```
%%
[ \t]+        putchar( ' ' );
[ \t]+$       /* ignore this token */
```

If the action contains a `{', then the action spans till the balancing `}' is found, and the action may cross multiple lines. Lex/Flex knows about C strings and comments and won't be fooled by braces found within them, but also allows actions to begin with %{ and will consider the action to be all the text up to the next %} (regardless of ordinary braces inside the action).

Actions can include arbitrary C code, including return statements to return a value to whatever routine called yylex(). Each time yylex() is called it continues processing tokens from where it last left off until it either reaches the end of the file or executes a return. Once it reaches an end-of-file, however, then any subsequent call to yylex() will simply immediately return.

Actions are not allowed to modify yytext or yyleng.

The Code Section

The code section contains the definitions of the routines called by the action part of a rule. This section also contains the definition of the function main if the scanner is a stand-alone program.

The Generated Scanner

The output of Lex/Flex is the file lex.yy.c, which contains the scanning routine yylex(), a number of tables used by it for matching tokens, and a number of auxiliary routines and macros. By default, yylex() is declared as follows:

```
int yylex()
    {
    ... various definitions and the actions in here ...
    }
```

(If your environment supports function prototypes, then it will be ``int yylex( void )".) This definition may be changed by redefining the ``YY_DECL" macro. For example, you could use:

```
#undef YY_DECL
#define YY_DECL float lexscan( a, b ) float a, b;
```

to give the scanning routine the name lexscan, returning a float, and taking two floats as arguments. Note that if you give arguments to the scanning routine using a K&R-style/non-prototyped function declaration, you must terminate the definition with a semi-colon (;).

Whenever yylex() is called, it scans tokens from the global input file yyin (which defaults to stdin). It continues until it either reaches an end-of-file (at which point it returns the value 0) or one of its actions executes a return statement. In the former case, when called again the scanner will immediately return unless yyrestart() is called to point yyin at the new input file. (yyrestart() takes one argument, a FILE * pointer.) In the latter case (i.e., when an action executes a return), the scanner may then be called again and it will resume scanning where it left off.

Interfacing with Yacc/Bison

One of the main uses of Lex/Flex is as a companion to the Yacc/Bison parser-generator. Yacc/Bison parsers expect to call a routine named yylex() to find the next input token. The routine is supposed to return the type of the next token as well as putting any associated value in the global yylval. To use Lex/Flex with Yacc/Bison, one specifies the -d option to Yacc/Bison to instruct it to generate the file y.tab.h containing definitions of all the %tokens appearing in the Yacc/Bison input. This file is then included in the Lex/Flex scanner. For example, if one of the tokens is ``TOK_NUMBER", part of the scanner might look like:

```
%{
#include "y.tab.h"
%}

%%

[0-9]+        yylval = atoi( yytext ); return TOK_NUMBER;
```

Yacc/Bison

In order for Yacc/Bison to parse a language, the language must be described by a *context-free grammar*. The most common formal system for presenting such rules for humans to read is *Backus-Naur Form* or ``BNF", which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Yacc/Bison is essentially machine-readable BNF.

Not all context-free languages can be handled by Yacc/Bison, only those that are LALR(1). In brief, this means that it must be possibly to tell how to parse any portion of an input string with just a single token of look-ahead. Strictly speaking, that is a description of an LR(1) grammar, and LALR(1) involves additional restrictions that are hard to explain simply; but it is rare in actual practice to find an LR(1) grammar that fails to be LALR(1).

An Overview

A formal grammar selects tokens only by their classifications: for example, if a rule mentions the terminal symbol `integer constant', it means that *any* integer constant is grammatically valid in that position. The precise value of the constant is irrelevant to how to parse the input: if `x+4` is grammatical then `x+1` or `x+3989` is equally grammatical.

But the precise value is very important for what the input means once it is parsed. A compiler is useless if it fails to distinguish between 4, 1 and 3989 as constants in the program! Therefore, each token has both a token type and a *semantic value*.

The token type is a terminal symbol defined in the grammar, such as `INTEGER`, `IDENTIFIER` or `','`. It tells everything you need to know to decide where the token may validly appear and how to group it with other tokens. The grammar rules know nothing about tokens except their types.

The semantic value has all the rest of the information about the meaning of the token, such as the value of an integer, or the name of an identifier. (A token such as `','` which is just punctuation doesn't need to have any semantic value.)

For example, an input token might be classified as token type `INTEGER` and have the semantic value 4. Another input token might have the same token type `INTEGER` but value 3989. When a grammar rule says that `INTEGER` is allowed, either of these tokens is acceptable because each is an `INTEGER`. When the parser accepts the token, it keeps track of the token's semantic value.

Each grouping can also have a semantic value as well as its nonterminal symbol. For example, in a calculator, an expression typically has a semantic value that is a number. In a compiler for a programming language, an expression typically has a semantic value that is a tree structure describing

the meaning of the expression.

As Yacc/Bison reads tokens, it pushes them onto a stack along with their semantic values. The stack is called the *parser stack*. Pushing a token is traditionally called *shifting*.

But the stack does not always have an element for each token read. When the last *n* tokens and groupings shifted match the components of a grammar rule, they can be combined according to that rule. This is called *reduction*. Those tokens and groupings are replaced on the stack by a single grouping whose symbol is the result (left hand side) of that rule. Running the rule's action is part of the process of reduction, because this is what computes the semantic value of the resulting grouping.

The Yacc/Bison parser reads a sequence of tokens as its input, and groups the tokens using the grammar rules. If the input is valid, the end result is that the entire token sequence reduces to a single grouping whose symbol is the grammar's start symbol. If we use a grammar for C, the entire input must be a `sequence of definitions and declarations'. If not, the parser reports a syntax error.

The parser tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.

This kind of parser is known in the literature as a bottom-up parser.

The function `yyparse` is implemented using a finite-state machine. The values pushed on the parser stack are not simply token type codes; they represent the entire sequence of terminal and nonterminal symbols at or near the top of the stack. The current state collects all the information about previous input which is relevant to deciding what to do next.

Each time a look-ahead token is read, the current parser state together with the type of look-ahead token are looked up in a table. This table entry can say, ``Shift the look-ahead token.'' In this case, it also specifies the new parser state, which is pushed onto the top of the parser stack. Or it can say, ``Reduce using rule number *n*.'' This means that a certain of tokens or groupings are taken off the top of the stack, and replaced by one grouping. In other words, that number of states are popped from the stack, and one new state is pushed.

There is one other alternative: the table can say that the look-ahead token is erroneous in the current state. This causes error processing to begin.

A Yacc/Bison Example

The following is a Yacc/Bison input file which defines a reverse polish notation calculator. The file created by Yacc/Bison simulates the calculator. The details of the example are explained in later sections.

```
/* Reverse polish notation calculator. */
%{
#define YYSTYPE double
```

```
#include <math.h>
%}
%token NUM
%% /* Grammar rules and actions follow */
input : /* empty */
   | input line
;
line : '\n'
   | exp '\n'  { printf ("\t%.10g\n", $1); }
;
exp : NUM             { $$ = $1;         }
   | exp exp '+'     { $$ = $1 + $2;    }
   | exp exp '-'     { $$ = $1 - $2;    }
   | exp exp '*'     { $$ = $1 * $2;    }
   | exp exp '/'     { $$ = $1 / $2;    }
   /* Exponentiation */
   | exp exp '^'     { $$ = pow ($1, $2); }
   /* Unary minus    */
  | exp 'n'         { $$ = -$1;        }
;
%%
/* Lexical analyzer returns a double floating point
   number on the stack and the token NUM, or the ASCII
   character read if not a number.  Skips all blanks
   and tabs, returns 0 for EOF. */
#include <ctype.h>
yylex ()
{ int c;
  /* skip white space  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* process numbers   */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* return end-of-file  */
  if (c == EOF)
    return 0;
  /* return single chars */
  return c;
}
main ()     /* The ``Main'' function to make this stand-alone  */
{
  yyparse ();
```

```
}
#include <stdio.h>
yyerror (s)  /* Called by yyparse on error */
     char *s;
{
  printf ("%s\n", s);
}
```

The Yacc/Bison Input File

Yacc/Bison takes as input a context-free grammar specification and produces a C-language function that recognizes correct instances of the grammar. The input file for the Yacc/Bison utility is a *Yacc/Bison grammar file*. The Yacc/Bison grammar input file conventionally has a name ending in `.y`.

A Yacc/Bison grammar file has four main sections, shown here with the appropriate delimiters:

```
%{
C declarations
%}
Yacc/Bison declarations
%%
Grammar rules
%%
Additional C code
```

Comments enclosed in `/* ... */` may appear in any of the sections. The %%, `%{` and %} are punctuation that appears in every Yacc/Bison grammar file to separate the sections.

The C declarations may define types and variables used in the actions. You can also use preprocessor commands to define macros used there, and use `\#include` to include header files that do any of these things.

The Yacc/Bison declarations declare the names of the terminal and nonterminal symbols, and may also describe operator precedence and the data types of semantic values of various symbols.

The grammar rules define how to construct each nonterminal symbol from its parts.

The additional C code can contain any C code you want to use. Often the definition of the lexical analyzer `yylex` goes here, plus subroutines called by the actions in the grammar rules. In a simple program, all the rest of the program can go here.

The Declarations Section

The C Declarations Section

The *C declarations* section contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules. These are copied to the beginning of the parser file so that they precede the definition of `yylex`. You can use `#include` to get the declarations from a header file. If you don't need any C declarations, you may omit the `%{` and `%}` delimiters that bracket this section.

The Yacc/Bison Declarations Section

The *Yacc/Bison declarations* section defines symbols of the grammar. *Symbols* in Yacc/Bison grammars represent the grammatical classifications of the language.

Definitions are provided for the terminal and nonterminal symbols, to specify the precedence and associativity of the operators, and the data types of semantic values.

The first rule in the file also specifies the start symbol, by default. If you want some other symbol to be the start symbol, you must declare it explicitly.

Symbol names can contain letters, digits (not at the beginning), underscores and periods. Periods make sense only in nonterminals.

A *terminal symbol* (also known as a *token type*) represents a class of syntactically equivalent tokens. You use the symbol in grammar rules to mean that a token in that class is allowed. The symbol is represented in the Yacc/Bison parser by a numeric code, and the `yylex` function returns a token type code to indicate what kind of token has been read. You don't need to know what the code value is; you can use the symbol to stand for it. By convention, it should be all upper case. All token type names (but not single-character literal tokens such as `'+'` and `'*'`) must be declared.

There are two ways of writing terminal symbols in the grammar:

- A *named token type* is written with an identifier, it should be all upper case such as, `INTEGER`, `IDENTIFIER`, `IF` or `RETURN`. A terminal symbol that stands for a particular keyword in the language should be named after that keyword converted to upper case. Each such name must be defined with a Yacc/Bison declaration such as

  ```
  %token INTEGER IDENTIFIER
  ```

  The terminal symbol `error` is reserved for error recovery. In particular, `yylex` should never return this value.

- A *character token type* (or *literal token*) is written in the grammar using the same syntax used in C for character constants; for example, `'+'` is a character token type. A character token type doesn't need to be declared unless you need to specify its semantic value data type, associativity, or precedence.

  By convention, a character token type is used only to represent a token that consists of that

particular character. Thus, the token type `'+'` is used to represent the character `+` as a token. Nothing enforces this convention, but if you depart from it, your program will confuse other readers.

All the usual escape sequences used in character literals in C can be used in Yacc/Bison as well, but you must not use the null character as a character literal because its ASCII code, zero, is the code `yylex` returns for end-of-input.

How you choose to write a terminal symbol has no effect on its grammatical meaning. That depends only on where it appears in rules and on when the parser function returns that symbol.

The value returned by `yylex` is always one of the terminal symbols (or 0 for end-of-input). Whichever way you write the token type in the grammar rules, you write it the same way in the definition of `yylex`. The numeric code for a character token type is simply the ASCII code for the character, so `yylex` can use the identical character constant to generate the requisite code. Each named token type becomes a C macro in the parser file, so `yylex` can use the name to stand for the code. (This is why periods don't make sense in terminal symbols.)

If `yylex` is defined in a separate file, you need to arrange for the token-type macro definitions to be available there. Use the `-d` option when you run Yacc/Bison, so that it will write these macro definitions into a separate header file *`name`*`.tab.h` which you can include in the other source files that need it.

A *nonterminal symbol* stands for a class of syntactically equivalent groupings. The symbol name is used in writing grammar rules. By convention, it should be all lower case, such as `expr`, `stmt` or `declaration`. Nonterminal symbols must be declared if you need to specify which data type to use for the semantic value.

Token Type Names

The basic way to declare a token type name (terminal symbol) is as follows:

```
%token name
```

Yacc/Bison will convert this into a `#define` directive in the parser, so that the function `yylex` (if it is in this file) can use the name *name* to stand for this token type's code.

Alternatively you can use `%left`, `%right`, or `%nonassoc` instead of `%token`, if you wish to specify precedence.

You can explicitly specify the numeric code for a token type by appending an integer value in the field immediately following the token name:

```
%token NUM 300
```

It is generally best, however, to let Yacc/Bison choose the numeric codes for all token types. Yacc/Bison will automatically select codes that don't conflict with each other or with ASCII characters.

In the event that the stack type is a union, you must augment the `%token` or other token declaration to include the data type alternative delimited by angle-brackets. For example:

```
%union {              /* define stack type */
  double val;
  symrec *tptr;
}
%token  NUM      /* define token NUM and its type */
```

Operator Precedence

Use the `%left`, `%right` or `%nonassoc` declaration to declare a token and specify its precedence and associativity, all at once. These are called *precedence declarations*.

The syntax of a precedence declaration is the same as that of `%token`: either

```
%left symbols ...
```

or

```
%left <type> symbols ...
```

And indeed any of these declarations serves the purposes of `%token`. But in addition, they specify the associativity and relative precedence for all the *symbols*:

- The associativity of an operator *op* determines how repeated uses of the operator nest: whether *x op y op z* is parsed by grouping *x* with *y* first or by grouping *y* with *z* first. `%left` specifies left-associativity (grouping *x* with *y* first) and `%right` specifies right-associativity (grouping *y* with *z* first). `%nonassoc` specifies no associativity, which means that *x op y op z* is considered a syntax error.
- The precedence of an operator determines how it nests with other operators. All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity. When two tokens declared in different precedence declarations associate, the one declared later has the higher precedence and is grouped first.

The Collection of Value Types

The `%union` declaration specifies the entire collection of possible data types for semantic values. The keyword `%union` is followed by a pair of braces containing the same thing that goes inside a `union` in C. For example:

```
%union {
```

```
  double val;
  symrec *tptr;
}
```

This says that the two alternative types are `double` and `symrec *`. They are given names `val` and `tptr`; these names are used in the `%token` and `%type` declarations to pick one of the types for a terminal or nonterminal symbol. Note that, unlike making a `union` declaration in C, you do not write a semicolon after the closing brace.

Yacc/Bison Declaration Summary

Here is a summary of all Yacc/Bison declarations: \begin{description}

%union] Declare the collection of data types that semantic values may have.
%token] Declare a terminal symbol (token type name) with no precedence or associativity specified.
%right] Declare a terminal symbol (token type name) that is right-associative.
%left] Declare a terminal symbol (token type name) that is left-associative.
%nonassoc] Declare a terminal symbol (token type name) that is nonassociative (using it in a way that would be associative is a syntax error).
%type $<$*non-terminal*$>$] Declare the type of semantic values for a nonterminal symbol. When you use `%union` to specify multiple value types, you must declare the value type of each nonterminal symbol for which values are used. This is done with a `%type` declaration. Here *nonterminal* is the name of a nonterminal symbol, and *type* is the name given in the `%union` to the alternative that you want. You can give any number of nonterminal symbols in the same `%type` declaration, if they have the same value type. Use spaces to separate the symbol names.
%start $<$*non-terminal*$>$] Specify the grammar's start symbol. Yacc/Bison assumes by default that the start symbol for the grammar is the first nonterminal specified in the grammar specification section. The programmer may override this restriction with the `%start` declaration. \end{description}

The Grammar Rules Section

The *grammar rules* section contains one or more Yacc/Bison grammar rules, and nothing else.

There must always be at least one grammar rule, and the first %% (which precedes the grammar rules) may never be omitted even if it is the first thing in the file.

A Yacc/Bison grammar rule has the following general form:

```
result : components ...;
```

where *result* is the nonterminal symbol that this rule describes and *components* are various terminal and nonterminal symbols that are put together by this rule. For example,

```
exp : exp '+' exp ;
```

says that two groupings of type `exp`, with a `+` token in between, can be combined into a larger grouping of type `exp`.

Whitespace in rules is significant only to separate symbols. You can add extra whitespace as you wish.

Scattered among the components can be *actions* that determine the semantics of the rule. An action looks like this:

```
{C statements}
```

Usually there is only one action and it follows the components.

Multiple rules for the same *result* can be written separately or can be joined with the vertical-bar character `|` as follows:

```
result : rule1-components ...
       | rule2-components ...
       ...
       ;
```

They are still considered distinct rules even when joined in this way.

If *components* in a rule is empty, it means that *result* can match the empty string. For example, here is how to define a comma-separated sequence of zero or more `exp` groupings:

```
expseq : /* empty */
       | expseq1
       ;

expseq1 : exp
       | expseq1 ',' exp
       ;
```

It is customary to write a comment `/* empty */` in each rule with no components.

A rule is called *recursive* when its *result* nonterminal appears also on its right hand side. Nearly all Yacc/Bison grammars need to use recursion, because that is the only way to define a sequence of any number of somethings. Consider this recursive definition of a comma-separated sequence of one or more expressions:

```
expseq1 : exp
       | expseq1 ',' exp
       ;
```

Since the recursive use of `expseq1` is the leftmost symbol in the right hand side, we call this *left*

recursion. By contrast, here the same construct is defined using *right recursion*:

```
expseq1 : exp
        | exp ',' expseq1
        ;
```

Any kind of sequence can be defined using either left recursion or right recursion, but you should always use left recursion, because it can parse a sequence of any number of elements with bounded stack space. Right recursion uses up space on the Yacc/Bison stack in proportion to the number of elements in the sequence, because all the elements must be shifted onto the stack before the rule can be applied even once.

Indirect or *mutual* recursion occurs when the result of the rule does not appear directly on its right hand side, but does appear in rules for other nonterminals which do appear on its right hand side. For example:

```
expr : primary
        | primary '+' primary
        ;

primary : constant
        | '(' expr ')'
        ;
```

defines two mutually-recursive nonterminals, since each refers to the other.

Semantic Actions

In order to be useful, a program must do more than parse input; it must also produce some output based on the input. In a Yacc/Bison grammar, a grammar rule can have an *action* made up of C statements. Each time the parser recognizes a match for that rule, the action is executed.

Most of the time, the purpose of an action is to compute the semantic value of the whole construct from the semantic values of its parts. For example, suppose we have a rule which says an expression can be the sum of two expressions. When the parser recognizes such a sum, each of the subexpressions has a semantic value which describes how it was built up. The action for this rule should create a similar sort of value for the newly recognized larger expression.

For example, here is a rule that says an expression can be the sum of two subexpressions:

```
expr : expr '+' expr   { $$ = $1 + $3; }
        ;
```

The action says how to produce the semantic value of the sum expression from the values of the two subexpressions.

Defining Language Semantics

The grammar rules for a language determine only the syntax. The semantics are determined by the semantic values associated with various tokens and groupings, and by the actions taken when various groupings are recognized.

For example, the calculator calculates properly because the value associated with each expression is the proper number; it adds properly because the action for the grouping {`x + y`} is to add the numbers associated with *x* and *y*.

Data Types of Semantic Values

In a simple program it may be sufficient to use the same data type for the semantic values of all language constructs. Yacc/Bison's default is to use type `int` for all semantic values. To specify some other type, define `YYSTYPE` as a macro, like this:

```
#define YYSTYPE double
```

This macro definition must go in the C declarations section of the grammar file.

More Than One Value Type

In most programs, you will need different data types for different kinds of tokens and groupings. For example, a numeric constant may need type `int` or `long`, while a string constant needs type `char *`, and an identifier might need a pointer to an entry in the symbol table.

To use more than one data type for semantic values in one parser, Yacc/Bison requires you to do two things:

- Specify the entire collection of possible data types, with the `%union` Yacc/Bison declaration.
- Choose one of those types for each symbol (terminal or nonterminal) for which semantic values are used. This is done for tokens with the `%token` Yacc/Bison declaration and for groupings with the `%type` Yacc/Bison declaration.

An action accompanies a syntactic rule and contains C code to be executed each time an instance of that rule is recognized. The task of most actions is to compute a semantic value for the grouping built by the rule from the semantic values associated with tokens or smaller groupings.

An action consists of C statements surrounded by braces, much like a compound statement in C. It can be placed at any position in the rule; it is executed at that position. Most rules have just one action at the end of the rule, following all the components. Actions in the middle of a rule are tricky and used only for special purposes.

The C code in an action can refer to the semantic values of the components matched by the rule with the construct $*n*, which stands for the value of the *n*th component. The semantic value for the grouping

being constructed is $$. (Yacc/Bison translates both of these constructs into array element references when it copies the actions into the parser file.)

Here is a typical example:

```
exp :  ...
       | exp '+' exp
          { $$ = $1 + $3; }
```

This rule constructs an exp from two smaller exp groupings connected by a plus-sign token. In the action, $1 and $3 refer to the semantic values of the two component exp groupings, which are the first and third symbols on the right hand side of the rule. The sum is stored into $$ so that it becomes the semantic value of the addition-expression just recognized by the rule. If there were a useful semantic value associated with the + token, it could be referred to as $2.

$*n* with *n* zero or negative is allowed for reference to tokens and groupings on the stack *before* those that match the current rule. This is a very risky practice, and to use it reliably you must be certain of the context in which the rule is applied. Here is a case in which you can use this reliably:

```
foo : expr bar '+' expr  {  ... }
       | expr bar '-' expr  {  ... }
       ;

bar : /* empty */
       { previous_expr = $0; }
       ;
```

As long as bar is used only in the fashion shown here, \$0 always refers to the expr which precedes bar in the definition of foo.

Data Types of Values in Actions

If you have chosen a single data type for semantic values, the \$\$ and \$*n* constructs always have that data type.

If you have used %union to specify a variety of data types, then you must declare a choice among these types for each terminal or nonterminal symbol that can have a semantic value. Then each time you use $$ or $*n*, its data type is determined by which symbol it refers to in the rule. In this example,efill

```
exp :  ...
       | exp '+' exp
          { $$ = $1 + $3; }
```

$1 and $3 refer to instances of exp, so they all have the data type declared for the nonterminal symbol exp. If $2 were used, it would have the data type declared for the terminal symbol '+', whatever that might be.

Alternatively, you can specify the data type when you refer to the value, by inserting `<`*`type`*`>` after the `$` at the beginning of the reference. For example, if you have defined types as shown here:

```
%union {
  int itype;
  double dtype;
}
```

then you can write `$<itype>1` to refer to the first subunit of the rule as an integer, or `$<dtype>1` to refer to it as a double.

Actions in Mid-Rule

Occasionally it is useful to put an action in the middle of a rule. These actions are written just like usual end-of-rule actions, but they are executed before the parser even recognizes the following components.

A mid-rule action may refer to the components preceding it using $*n*, but it may not refer to subsequent components because it is run before they are parsed.

The mid-rule action itself counts as one of the components of the rule. This makes a difference when there is another action later in the same rule (and usually there is another at the end): you have to count the actions along with the symbols when working out which number *n* to use in $*n*.

The mid-rule action can also have a semantic value. This can be set within that action by an assignment to `$$`, and can referred to by later actions using $*n*. Since there is no symbol to name the action, there is no way to declare a data type for the value in advance, so you must use the `$< ...>` construct to specify a data type each time you refer to this value.

Here is an example from a hypothetical compiler, handling a `let` statement that looks like `let (`*`variable`*`) `*`statement`* and serves to create a variable named *variable* temporarily for the duration of *statement*. To parse this construct, we must put *variable* into the symbol table while *statement* is parsed, then remove it afterward. Here is how it is done:

```
stmt : LET '(' var ')'
                { $<context>$ = push_context ();
                  declare_variable (\$3); }
       stmt     { $$ = $6;
                  pop_context ($<context>5); }
```

As soon as `let (`*`variable`*`)` has been recognized, the first action is run. It saves a copy of the current semantic context (the list of accessible variables) as its semantic value, using alternative `context` in the data-type union. Then it calls `declare_variable` to add the new variable to that list. Once the first action is finished, the embedded statement `stmt` can be parsed. Note that the mid-rule action is component number 5, so the `stmt` is component number 6.

After the embedded statement is parsed, its semantic value becomes the value of the entire `let`-statement. Then the semantic value from the earlier action is used to restore the prior list of variables. This removes the temporary `let`-variable from the list so that it won't appear to exist while the rest of the program is parsed.

Taking action before a rule is completely recognized often leads to conflicts since the parser must commit to a parse in order to execute the action. For example, the following two rules, without mid-rule actions, can coexist in a working parser because the parser can shift the open-brace token and look at what follows before deciding whether there is a declaration or not:

```
compound : '{' declarations statements '}'
         | '{' statements '}'
         ;
```

But when we add a mid-rule action as follows, the rules become nonfunctional:

```
compound : { prepare_for_local_variables (); }
           '{' declarations statements '}'
         | '{' statements '}'
         ;
```

Now the parser is forced to decide whether to run the mid-rule action when it has read no farther than the open-brace. In other words, it must commit to using one rule or the other, without sufficient information to do it correctly. (The open-brace token is what is called the *look-ahead* token at this time, since the parser is still deciding what to do about it.

You might think that you could correct the problem by putting identical actions into the two rules, like this:

```
compound : { prepare_for_local_variables (); }
           '{' declarations statements '}'
         | { prepare_for_local_variables (); }
           '{' statements '}'
         ;
```

But this does not help, because Yacc/Bison does not realize that the two actions are identical. (Yacc/Bison never tries to understand the C code in an action.)

If the grammar is such that a declaration can be distinguished from a statement by the first token (which is true in C), then one solution which does work is to put the action after the open-brace, like this:

```
compound : '{' { prepare_for_local_variables (); }
           declarations statements '}'
         | '{' statements '}'
         ;
```

Now the first token of the following declaration or statement, which would in any case tell Yacc/Bison which rule to use, can still do so.

Another solution is to bury the action inside a nonterminal symbol which serves as a subroutine:

```
subroutine : /* empty */
             { prepare_for_local_variables (); }
           ;

compound : subroutine
             '{' declarations statements '}'
           | subroutine
             '{' statements '}'
           ;
```

Now Yacc/Bison can execute the action in the rule for `subroutine` without deciding which rule for `compound` it will eventually use. Note that the action is now at the end of its rule. Any mid-rule action can be converted to an end-of-rule action in this way, and this is what Yacc/Bison actually does to implement mid-rule actions.

The Additional C Code Section

The *additional C code* section is copied verbatim to the end of the parser file, just as the *C declarations* section is copied to the beginning. This is the most convenient place to put anything that you want to have in the parser file but which need not come before the definition of `yylex`. For example, the definitions of `yylex` and `yyerror` often go here.

If the last section is empty, you may omit the %% that separates it from the grammar rules.

The Yacc/Bison parser itself contains many static variables whose names start with `yy` and many macros whose names start with YY. It is a good idea to avoid using any such names (except those documented in this manual) in the additional C code section of the grammar file.

It is not usually acceptable to have a program terminate on a parse error. For example, a compiler should recover sufficiently to parse the rest of the input file and check it for errors.

Yacc/Bison Output: the Parser File

When you run Yacc/Bison, you give it a Yacc/Bison grammar file as input. The output is a C source file that parses the language described by the grammar. This file is called a *Yacc/Bison parser*. Keep in mind that the Yacc/Bison utility and the Yacc/Bison parser are two distinct programs: the Yacc/Bison utility is a program whose output is the Yacc/Bison parser that becomes part of your program.

The job of the Yacc/Bison parser is to group tokens into groupings according to the grammar rules---for

example, to build identifiers and operators into expressions. As it does this, it runs the actions for the grammar rules it uses.

The tokens come from a function called the *lexical analyzer* that you must supply in some fashion (such as by writing it in C or using Lex/Flex). The Yacc/Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what is ``inside'' the tokens (though their semantic values may reflect this). Typically the lexical analyzer makes the tokens by parsing characters of text, but Yacc/Bison does not depend on this.

The Yacc/Bison parser file is C code which defines a function named `yyparse` which implements that grammar. This function does not make a complete C program: you must supply some additional functions. One is the lexical analyzer. Another is an error-reporting function which the parser calls to report an error. In addition, a complete C program must start with a function called `main`; you have to provide this, and arrange for it to call `yyparse` or the parser will never run.

Aside from the token type names and the symbols in the actions you write, all variable and function names used in the Yacc/Bison parser file begin with `yy` or `YY`. This includes interface functions such as the lexical analyzer function `yylex`, the error reporting function `yyerror` and the parser function `yyparse` itself. This also includes numerous identifiers used for internal purposes. Therefore, you should avoid using C identifiers starting with `yy` or `YY` in the Yacc/Bison grammar file except for the ones defined in this manual.

Parser C-Language Interface

The Yacc/Bison parser is actually a C function named `yyparse`. Here we describe the interface conventions of `yyparse` and the other functions that it needs to use.

Keep in mind that the parser uses many C identifiers starting with `yy` and `YY` for internal purposes. If you use such an identifier (aside from those in this manual) in an action or in additional C code in the grammar file, you are likely to run into trouble.

The Parser Function `yyparse`

You call the function `yyparse` to cause parsing to occur. This function reads tokens, executes actions, and ultimately returns when it encounters end-of-input or an unrecoverable syntax error. You can also write an action which directs `yyparse` to return immediately without reading further.

The value returned by `yyparse` is 0 if parsing was successful (return is due to end-of-input).

The value is 1 if parsing failed (return is due to a syntax error).

In an action, you can cause immediate return from `yyparse` by using these macros:
\begin{description}

- YYACCEPT Return immediately with value 0 (to report success).
- YYABORT Return immediately with value 1 (to report failure). \end{description} }

The Lexical Analyzer Function `yylex`

The *lexical analyzer* function, `yylex`, recognizes tokens from the input stream and returns them to the parser. Yacc/Bison does not create this function automatically; you must write it so that `yyparse` can call it. The function is sometimes referred to as a lexical scanner.

In simple programs, `yylex` is often defined at the end of the Yacc/Bison grammar file. If `yylex` is defined in a separate source file, you need to arrange for the token-type macro definitions to be available there. To do this, use the `-d` option when you run Yacc/Bison, so that it will write these macro definitions into a separate header file *`name`*`.tab.h` which you can include in the other source files that need it.

Calling Convention for `yylex`

The value that `yylex` returns must be the numeric code for the type of token it has just found, or 0 for end-of-input.

When a token is referred to in the grammar rules by a name, that name in the parser file becomes a C macro whose definition is the proper numeric code for that token type. So `yylex` can use the name to indicate that type.

When a token is referred to in the grammar rules by a character literal, the numeric code for that character is also the code for the token type. So `yylex` can simply return that character code. The null character must not be used this way, because its code is zero and that is what signifies end-of-input.

Here is an example showing these things:

```
yylex()
{
   ...
  if (c == EOF)     /* Detect end of file. */
    return 0;
   ...
  if (c == '+' || c == '-')
    return c;      /* Assume token type for `+' is '+'. */
   ...
  return INT;      /* Return the type of the token. */
   ...
}
```

This interface has been designed so that the output from the `lex` utility can be used without change as the definition of `yylex`.

Semantic Values of Tokens

In an ordinary (nonreentrant) parser, the semantic value of the token must be stored into the global variable `yylval`. When you are using just one data type for semantic values, `yylval` has that type. Thus, if the type is `int` (the default), you might write this in `yylex`:

```
...
yylval = value;  /* Put value onto Yacc/Bison stack. */
return INT;      /* Return the type of the token. */
...
```

When you are using multiple data types, `yylval`'s type is a union made from the `%union` declaration. So when you store a token's value, you must use the proper member of the union. If the `%union` declaration looks like this:

```
%union {
  int intval;
  double val;
  symrec *tptr;
}
```

then the code in `yylex` might look like this:

```
...
yylval.intval = value; /* Put value onto Yacc/Bison stack. */
return INT;          /* Return the type of the token. */
...
```

Textual Positions of Tokens

If you are using the @*n*-feature in actions to keep track of the textual locations of tokens and groupings, then you must provide this information in `yylex`. The function `yyparse` expects to find the textual location of a token just parsed in the global variable `yylloc`. So `yylex` must store the proper data in that variable. The value of `yylloc` is a structure and you need only initialize the members that are going to be used by the actions. The four members are called `first_line`, `first_column`, `last_line` and `last_column`. Note that the use of this feature makes the parser noticeably slower.

The data type of `yylloc` has the name `YYLTYPE`.

The Error Reporting Function `yyerror`

The Yacc/Bison parser detects a *parse error* or *syntax error* whenever it reads a token which cannot satisfy any syntax rule. A action in the grammar can also explicitly proclaim an error, using the macro `YYERROR`.

The Yacc/Bison parser expects to report the error by calling an error reporting function named `yyerror`, which you must supply. It is called by `yyparse` whenever a syntax error is found, and it receives one argument. For a parse error, the string is always `"parse error"`.

The following definition suffices in simple programs:

```
yyerror (s)
     char *s;
{
  fprintf (stderr, "%s\", s);
}
```

After `yyerror` returns to `yyparse`, the latter will attempt error recovery if you have written suitable error recovery grammar rules. If recovery is impossible, `yyparse` will immediately return 1.

Debugging Your Parser

Shift/Reduce Conflicts

Suppose we are parsing a language which has if-then and if-then-else statements, with a pair of rules like this:

```
if_stmt : IF expr THEN stmt
| IF expr THEN stmt ELSE stmt
;
```

(Here we assume that `IF`, `THEN` and `ELSE` are terminal symbols for specific keyword tokens.)

When the `ELSE` token is read and becomes the look-ahead token, the contents of the stack (assuming the input is valid) are just right for reduction by the first rule. But it is also legitimate to shift the `ELSE`, because that would lead to eventual reduction by the second rule.

This situation, where either a shift or a reduction would be valid, is called a *shift/reduce conflict*. Yacc/Bison is designed to resolve these conflicts by choosing to shift, unless otherwise directed by operator precedence declarations. To see the reason for this, let's contrast it with the other alternative.

Since the parser prefers to shift the `ELSE`, the result is to attach the else-clause to the innermost if-statement, making these two inputs equivalent:

```
if x then if y then win(); else lose;

if x then do; if y then win(); else lose; end;
```

But if the parser chose to reduce when possible rather than shift, the result would be to attach the else-

clause to the outermost if-statement. The conflict exists because the grammar as written is ambiguous: either parsing of the simple nested if-statement is legitimate. The established convention is that these ambiguities are resolved by attaching the else-clause to the innermost if-statement; this is what Yacc/Bison accomplishes by choosing to shift rather than reduce. This particular ambiguity is called the ``dangling `else`'' ambiguity.

Operator Precedence

Another situation where shift/reduce conflicts appear is in arithmetic expressions. Here shifting is not always the preferred resolution; the Yacc/Bison declarations for operator precedence allow you to specify when to shift and when to reduce.

Consider the following ambiguous grammar fragment (ambiguous because the input {`1 - 2 * 3`} can be parsed in two different ways):

```
expr : expr '-' expr
       | expr '*' expr
       | expr '<' expr
       | '(' expr ')'
        ...
;
```

Suppose the parser has seen the tokens `1`, `-` and `2`; should it reduce them via the rule for the addition operator? It depends on the next token. Of course, if the next token is `)`, we must reduce; shifting is invalid because no single rule can reduce the token sequence {`- 2 )`} or anything starting with that. But if the next token is `*` or `<`, we have a choice: either shifting or reduction would allow the parse to complete, but with different results.

What about input such as {`1 - 2 - 5`}; should this be {`(1 - 2) - 5`} or should it be {`1 - (2 - 5)`}? For most operators we prefer the former, which is called *left association*. The latter alternative, *right association*, is desirable for assignment operators. The choice of left or right association is a matter of whether the parser chooses to shift or reduce when the stack contains {`1 - 2`} and the look-ahead token is `-`: shifting makes right-associativity.

Specifying Operator Precedence

Yacc/Bison allows you to specify these choices with the operator precedence declarations. Each such declaration contains a list of tokens, which are operators whose precedence and associativity is being declared. The `%left` declaration makes all those operators left-associative and the `%right` declaration makes them right-associative. A third alternative is `%nonassoc`, which declares that it is a syntax error to find the same operator twice ``in a row''.

The relative precedence of different operators is controlled by the order in which they are declared. The first `%left` or `%right` declaration in the file declares the operators whose precedence is lowest, the next such declaration declares the operators whose precedence is a little higher, and so on.

Precedence Examples

In our example, we would want the following declarations:

```
%left '<'
%left '-'
%left '*'
```

In a more complete example, which supports other operators as well, we would declare them in groups of equal precedence. For example, `'+'` is declared with `'-'`:

```
%left '<' '>' '=' NE LE GE
%left '+' '-'
%left '*' '/'
```

(Here `NE` and so on stand for the operators for ``not equal'' and so on. We assume that these tokens are more than one character long and therefore are represented by names, not character literals.)

Often the precedence of an operator depends on the context. For example, a minus sign typically has a very high precedence as a unary operator, and a somewhat lower precedence (lower than multiplication) as a binary operator.

The Yacc/Bison precedence declarations, `%left`, `%right` and `%nonassoc`, can only be used once for a given token; so a token has only one precedence declared in this way. For context-dependent precedence, you need to use an additional mechanism: the `%prec` modifier for rules.

The `%prec` modifier declares the precedence of a particular rule by specifying a terminal symbol whose predecence should be used for that rule. It's not necessary for that symbol to appear otherwise in the rule. The modifier's syntax is:

```
%prec terminal-symbol
```

and it is written after the components of the rule. Its effect is to assign the rule the precedence of *terminal-symbol*, overriding the precedence that would be deduced for it in the ordinary way. The altered rule precedence then affects how conflicts involving that rule are resolved.

Here is how `%prec` solves the problem of unary minus. First, declare a precedence for a fictitious terminal symbol named `UMINUS`. There are no tokens of this type, but the symbol serves to stand for its precedence:

```
  ...
%left '+' '-'
%left '*'
%left UMINUS
```

Now the precedence of `UMINUS` can be used in specific rules:

```
exp :  ...
       | exp '-' exp
       ...
       | '-' exp %prec UMINUS
```

Reduce/Reduce Conflicts

A reduce/reduce conflict occurs if there are two or more rules that apply to the same sequence of input. This usually indicates a serious error in the grammar.

Yacc/Bison resolves a reduce/reduce conflict by choosing to use the rule that appears first in the grammar, but it is very risky to rely on this. Every reduce/reduce conflict must be studied and usually eliminated.

Error Recovery

You can define how to recover from a syntax error by writing rules to recognize the special token `error`. This is a terminal symbol that is always defined (you need not declare it) and reserved for error handling. The Yacc/Bison parser generates an `error` token whenever a syntax error happens; if you have provided a rule to recognize this token in the current context, the parse can continue. For example:

```
stmnts : /* empty string */
       | stmnts '\'
       | stmnts exp '\'
       | stmnts error '\'
```

The fourth rule in this example says that an error followed by a newline makes a valid addition to any `stmnts`.

What happens if a syntax error occurs in the middle of an `exp`? The error recovery rule, interpreted strictly, applies to the precise sequence of a `stmnts`, an `error` and a newline. If an error occurs in the middle of an `exp`, there will probably be some additional tokens and subexpressions on the stack after the last `stmnts`, and there will be tokens to read before the next newline. So the rule is not applicable in the ordinary way.

But Yacc/Bison can force the situation to fit the rule, by discarding part of the semantic context and part of the input. First it discards states and objects from the stack until it gets back to a state in which the `error` token is acceptable. (This means that the subexpressions already parsed are discarded, back to the last complete `stmnts`.) At this point the `error` token can be shifted. Then, if the old look-ahead token is not acceptable to be shifted next, the parser reads tokens and discards them until it finds a token which is acceptable. In this example, Yacc/Bison reads and discards input until the next newline so that the fourth rule can apply.

The choice of error rules in the grammar is a choice of strategies for error recovery. A simple and useful strategy is simply to skip the rest of the current input line or current statement if an error is detected:

```
stmnt : error ';'  /* on error, skip until ';' is read */
```

It is also useful to recover to the matching close-delimiter of an opening-delimiter that has already been parsed. Otherwise the close-delimiter will probably appear to be unmatched, and generate another, spurious error message:

```
primary : '(' expr ')'
        | '(' error ')'
         ...
        ;
```

Error recovery strategies are necessarily guesses. When they guess wrong, one syntax error often leads to another. To prevent an outpouring of error messages, the parser will output no error message for another syntax error that happens shortly after the first; only after three consecutive input tokens have been successfully shifted will error messages resume.

Further Debugging

If a Yacc/Bison grammar compiles properly but doesn't do what you want when it runs, the `yydebug` parser-trace feature can help you figure out why.

To enable compilation of trace facilities, you must define the macro `YYDEBUG` when you compile the parser. You could use `-DYYDEBUG=1` as a compiler option or you could put `\#define YYDEBUG 1` in the C declarations section of the grammar file. Alternatively, use the `-t` option when you run Yacc/Bison. We always define `YYDEBUG` so that debugging is always possible.

The trace facility uses `stderr`, so you must add `#include <stdio.h>` to the C declarations section unless it is already there.

Once you have compiled the program with trace facilities, the way to request a trace is to store a nonzero value in the variable `yydebug`. You can do this by making the C code do it (in `main`).

Each step taken by the parser when `yydebug` is nonzero produces a line or two of trace information, written on `stderr`. The trace messages tell you these things:

- Each time the parser calls `yylex`, what kind of token was read.
- Each time a token is shifted, the depth and complete contents of the state stack.
- Each time a rule is reduced, which rule it is, and the complete contents of the state stack afterward.

To make sense of this information, it helps to refer to the listing file produced by the Yacc/Bison `-v` option. This file shows the meaning of each state in terms of positions in various rules, and also what

each state will do with each possible input token. As you read the successive trace messages, you can see that the parser is functioning according to its specification in the listing file. Eventually you will arrive at the place where something undesirable happens, and you will see which parts of the grammar are to blame.

Stages in Using Yacc/Bison

The actual language-design process using Yacc/Bison, from grammar specification to a working compiler or interpreter, has these parts:

1. Formally specify the grammar in a form recognized by Yacc/Bison. For each grammatical rule in the language, describe the action that is to be taken when an instance of that rule is recognized. The action is described by a sequence of C statements.
2. Write a lexical analyzer to process input and pass tokens to the parser. The lexical analyzer may be written by hand in C. It could also be produced using Lex.
3. Write a controlling function that calls the Yacc/Bison-produced parser.
4. Write error-reporting routines.

To turn this source code as written into a runnable program, you must follow these steps:

1. Run Yacc/Bison on the grammar to produce the parser. The usual way to invoke Yacc/Bison is as follows:

   `bison` *`infile`*

   Here *infile* is the grammar file name, which usually ends in `.y`. The parser file's name is made by replacing the `.y` with `.tab.c`. Thus, the `bison foo.y` filename yields {foo.tab.c.

2. Compile the code output by Yacc/Bison, as well as any other source files.
3. Link the object files to produce the finished product.

Multiparadigm Programming Language 0.9

The goal of this work is

- to design an abstract grammar for those elements that programming languages have in common in particular, for abstraction, generalization, and modules and
- to integrate the grammar with abstract grammars for a variety of programming paradigms.

This work is supports ideas developing in *Introduction to Programming Languages* where abstraction, generalization and computational models are used as unifying concepts for understanding programming languages. The goal in that document is to provide a top-down description of the language design process - idea, abstract sysntax, semantics, concrete syntax, formal semantics, and implementation

Design Description: Multiparadigm Programming Language 0.9

WORK IN PROGRESS

The goal of this work is

- to design an abstract grammar for those elements that programming languages have in common in particular, for abstraction, generalization, and modules and
- to integrate the grammar with abstract grammars for a variety of programming paradigms.

This work is supports ideas developing in *Introduction to Programming Languages* where abstraction, generalization and computational models are used as unifying concepts for understanding programming languages.

See the grammar

Program

A program is a collection of modules. One of which must be named main. There is a module that is imported into all other modules by default. The program address space ....

Modules & Blocks

Modules are used to construct libraries, ADTs, classes, interfaces, and implementations. A module is the compilation unit. A module which contains only type abstractions is a specification or interface module. In a module, the only external names visible are those that are explicitly imported or declared in the module. The only internal names visible outside the module are those that are explicitly exported.

The **extends** is used to list modules whose entire definition is made a part the the module.

The **implements** is used to list modules whose...

The **from-import** construct is used to list names imported from other modules. The imported names can be used as if they were declared locally.

The **interface** construct is syntactic sugar for ...

Declarations ... In the context of modules,

- **import**
- **export** designates a name that is visible wherever the module is visible
- **private** designates a name the is local to the module
- **protected** designates a name that is local to the module and all modules that import the module
- **static** designates a name that is common to all objects of the module
- **entry** designates an exported name that is an entry point of a monitor
- **initial** designates an abstract that is executed when the module is initially activated.
- **final** designates an abstract that is executed when the module is terminated

Blocks In a block, the names that are visible are those declared in the block or in any inclosing block. A block may be implemented using module syntax:

> **module import** enclosing modules **export** to interior modules .. **initial** *abstract* **end**

but module syntax requires that qualified names be used.

Abstraction, Generalization & Specialization

Abstraction provides names for abstracts that can be used wherever the abstract may be used.

Generalization provides a way to parameterize abstracts.

Specialization application of a generalization (or its name) to an argument.

Parameters are declarations. The modifier suggests how the corresponding argument will be handled.

- in: values imported into the abstract
- out: values exported from the abstract
- in-out: values imported to and exported from the abstract

Arguments define what can be passed to an abstract and returned as a result.

- strict
- non-strict

Functional Programming

A functional program is an expression. The expressions include

- constants
- variables
- function application and
- function abstraction

The grammar provides no syntactic sugar, it is the lambda calculus.

Constants

Constants include numbers, the boolean values, nil, the arithmetic and relational operators, and other predefined function symbols.

Variables

Variable are identifiers. If the variable is the name of an abstract, then its value is the abstract otherwise its value is undefined.

Function Application

Function application takes the form

$$(\ expression_1 \ expression_2 \ )$$

The result is the reduction of the application to normal form. Reduction to normal form is function evaluation which if $expression_1$ is a generic then the quantifier is removed from the expression and $expression_2$ is substituted, in the body of $expression_1$, for the quantified variable. If the resulting expression is reducible, then it is reduced.

Function Abstraction

A function abstraction is in normal form and stands for its self.

Imperative Programming

An imperative program is a command. The commands include

- Skip Command
- Application Command
- Assignment Command
- Parallel Command

- Sequential Command
- Choice Command
- Iterative Command
- Abstraction
- Invocation

Skip Command

The skip command has the form

```
skip
```

It does nothing.

Application Command

The application command has the form

name(*actual parameters*)

The action performed by an application command is determined by its definition.

Assignment Command

The assignment command has the form:

$identifer_0, \ldots, identifier_n := expression_0, \ldots, expression_n$

For n>=0. The effect is as if the expressions are evaluated and assigned in parallel with the i^{th} $identifier_i$ assigned the value of the i^{th} $expression_i$. The identifier and expression must be type compatible (matching types).

Parallel Command

The parallel command is of the form:

{|| $command_0, \ldots, command_n$ }

The programmer may make no assumptions about the degree of parallelism or relative speeds with which the commands execute.

Sequential Command

The sequential command is of the form:

```
{; command_0,..., command_n }
```

The programmer may assume that the commands execute in sequence from left to right with each command terminating before the next begins.

Choice Command

The choice command is nondeterministic and is of the form:

```
{? guard_0 --> command_0,
   ...,
   guard_n --> command_n
}
```

The programmer may assume that if no guard evaluates to true, that the command terminates and that if some guard is true, that exactly one of the commands corresponding to a guard that evaluates to true is executed.

Iterative Command

The iterative command is nondeterministic and is of the form:

```
{* guard_0 --> command_0,
   ...,
   guard_n --> command_n
}
```

The programmer may assume that while some guard is true, exactly one of the commands corresponding to a guard that evaluates to true is executed and that if no guard evaluates to true, that the command terminates. The guards are reevaluated after the execution of a command.

Abstraction

Inline abstractions are restricted to

Invocation

Invocations are restricted to direct recursion within an abstraction,

Logic Programming

The logic programming grammar is pure Prolog.

Communication and Events

Monitor The monitor construct is for communication and synchronization in a shared memory system.

Send-Receive The send and receive constructs are for communication and synchronization in message passing systems.

Exceptions

Threads

Type Expressions, Literals and Constants

The goal is to provide a useful range of primitive and structured types and type definition facilities. One novel feature is the inclusion of a **null** (empty) type and a **nil** (undefined) value.

Types are either primitive or structured and may be referenced by name. In addition to the usual primitive types, a null type (empty) is provided. Singleton types (types with only one element) are called atoms (the type name and element are the same).

- **atom**: a type with exactly one value; implementation - type name and only element are the same - **x isa x**

The structured types are the sum, product, function (array) and class (in the form of a module). The sum type provides an optional tag which may be used when type information is needed. The sum type is used to define enumerated and subrange types.

- **sum type**: (+)
- **enumerated types**: constructed from sum types using primitive values and atoms (+ sun, mon, tue, wed, thir, fri, sat)
- **range types**: i..j = (+ i..j)

The product type provides for optional field names. Access to the individual fields is by field name or by pattern matching. The product type when combined with abstraction and generalization, permits the definition of recursive and polymorphic types. Product types may be defined using modules.

- **recursive types**: listT is (+ empty, (* item, listT))
- **Polymorphic Type**: BTreePT is \T,node.(+ empty, (*n is a T, BTreePT T, BTreePT T))
    - Integer Type Binary Tree : BTreeIntegerT is BTreePT **integer**
    - Variable of an integer type binary tree: MyTree isa BTreeIntegerT

Arrays are provided via a mutable function type

- **array types**: map (*0..n) --> (*a_0..,a_n)

Missing are set and file types.

Implementation

The compiler will be in written in Java and generate a Java program as object code to insure portability.

Grammar: Multiparadigm Programming Language 0.9

WORK IN PROGRESS

The goal of this work is

- to design an abstract grammar for those elements that programming languages have in common in particular, for abstraction, generalization, and modules and
- to integrate the grammar with abstract grammars for a variety of programming paradigms.

This work is supports ideas developing in *Introduction to Programming Languages* where abstraction, generalization and computational models are used as unifying concepts for understanding programming languages.

Notation

Figure M.N: **Notation**

Notation	Means
N ::= R	Occurrence of N may be replaced with R
A B	A followed by B
A | B	A or B
(A)	Grouping
[A]	A is optional
[A]*	Possibly empty sequence of A's
[A]+	One or more As
bold	Literal symbols
italic	Nonterminals
standard font	Symbols of the grammar definition language

The Grammar

Figure N.M: **Unified Paradigm Grammar**

Program:		
program	::=	[*module*]$^+$ (**?-** *predicate* [**,** *predicate*]* | **!** *command* | *#expression*)
Modules & blocks:		
module	::=	**module** [(**extends** *moduleName*$^+$| **implements** *interfaceName*)] *declaration** | **interface** *import** *declaration** |
declaration	::=	| [(**export** | **public**) | **private** | **protected** | **static**] (*abstraction* | *clause*) | (**initial** | **final**) *abstract*
import	::=	**import**(*moduleName* [**as** *localAlias*])$^+$ | **from** *moduleName* **import** *name*$^+$
block	::=	**block** *declaration** *abstract*
Abstraction, Generalization, & Specialization:		
abstraction	::=	*name* [**is** *abstract*]
abstract	::=	*expression* | *command* | [**a**] *type* [**,** *constant*] | *module* | *generalization*
generalization	::=	\ *parameter* **.** *abstract*
parameter	::=	*declaration*
specialization	::=	*generalization argument*
argument	::=	[[**non**] **strict**] *abstract*
Object Oriented Programming		
class	::=	**class**[(**extends** *className*$^+$ | **implements** *interfaceName*)] *declaration**
object	::=	*name***is**[**a**] *module*
Functional Programming: lambda calculus (reduction of an expression to a normal form)		
expression	::=	*constant* | *variableName* | **(** *expression expression* **)** | *expressionGeneralization* | *expressonBlock* | *expressionModule*
Imperative Programming: binding sequences		

command	::=	**skip** | *event* | *identifier*$^+$ **:=** *expression*$^+$ | **{;** *command** **}** | **{||** *command** **}** | **{?** *guardedCommand** **}** | **{*** *guardedCommand** **}** | *commandInvocation* | *commandBlock* | *commandModule*
guardedCommand	::=	*guard*--> *command*
guard	::=	(*event* | *booleanExpression*)[**,** *booleanExpression*)]* | **else**
Logic Programming: pure Prolog (deduction)		
logicProgram	::=	*theory query*
theory	::=	*clause*$^+$
clause	::=	[*predicate* **:-**] [*predicate* **,**]* *predicate* **.**
predicate	::=	*atom* | *atom***(** *term* [**,** *term*]* **)**
term	::=	*number* | *atom*[**(** *term* [**,** *term*]* **)**] | *variable*
query	::=	**?-** *predicate* [**,** *predicate*]* **.**
Concurrent Programming**Threads:** class or ?		
Communication and Event Primitives:I/O, keyboard, mouse, etc		
monitor	::=	**monitor** *declaration**
event	::=	*send* | *receive*
send	::=	**send***message***to***processIdentifier* | p**!**e | **output** *expression*
receive	::=	**receive***message***from** *processIdentifier* | p**?**x | **input** *variable*
message	::=	<**info,***a,b*>
Exceptions:class, expression, command, predicate		
throw	::=	(**throw** | **raise** | **error**) *exceptionName* [**(** *value* **)**]
exception	::=	**try** *abstract* (**catch** | **except**) *handlers***finally** *handler*
handlers	::=	[| *handler*]$^+$
handler	::=	*exceptionName*[**(** *value* **)**] [**,** *exceptionName* [**(** *value* **)**]]* => *command*
Type Expressions:		
type	::=	*primitiveType* | *structuredType* | *typeName*
primitiveType	::=	**null** | *atom* | **Boolean** | **Number** | **Character** | **String**
structuredType	::=	*sum* | *product* | *function* | *module* **exception** [**(***type***)**]
sum	::=	**(+** (*atom* | [*name***:**]*type*) [**,** (*atom* | [*name***:**]*type*)]* **)** | **(+** *I***..***J***)** | **(+** *I***,***J***..***K***)** *A* **is module end** *B* **is module extends** *A* **end** ... *N* **is module extends** *N-1* **end**

product	::=	**(*** [*name* **isa**]*type* [**,** [*name* **isa**]*type*]* **)** | **module** (**export** *name* **isa** *type* **.**)$^{+}$ **end**
function	::=	[**map**] *sum* **-->** *type*
Element Construction: literals and values		
constant	::=	*atomic* | *structured*
atomic	::=	**nil** | _|_ | *atom* | *boolean* | *number* | *character* | *string*
structured	::=	**(*** *constant* [**,** *constant*]* **)** | **map** *sum* **-->** *sum*

Syntactic Sugar - conventions & idioms for simplying programming

---

Multiparadigm Programming Language: Semantics 0.9

WORK IN PROGRESS
The goal of this work is

- to design an abstract grammar for those elements that programming languages have in common in particular, for abstraction, generalization, and modules and
- to integrate the grammar with abstract grammars for a variety of programming paradigms.

This work is supports ideas developing in *Introduction to Programming Languages* where abstraction, generalization and computational models are used as unifying concepts for understanding programming languages.

Imperative Programming

Axiomatic Semantics

Skip

$$\{P\}\textbf{skip}\{P\}$$

Assignment

$$\{P[x_0 \textbf{ is } e_0 \textbf{ ,...,} x_n \textbf{ is } e_n]\}x_0 \textbf{ ,...,} x_n \textbf{ := } e_0 \textbf{ ,...,} e_n\{P\}$$

Linear Sequence

$$\frac{P=Q_0,\ \{Q_i\}\ command_{i+1}\{Q_{i+1}\},\ Q_n=R;\ \text{for } i=0..n}{\{P\}\textbf{\{;}\ command_1\ \textbf{,...,}\ command_n\ \textbf{\}}\{R\}}$$

Selection

$$\frac{\{P \wedge guard_i\}\ \ command_i\{Q\};\ \text{for } i=1..n}{\{P\}\ \textbf{\{?}\ guard_1 \text{ --> } command_1\ \textbf{,}\ \ldots\ \textbf{,}\ guard_n \text{ --> } command_n\ \ \textbf{\}}\ \{Q\}}$$

Iteration

$$\frac{\{I \wedge guard_i\}\ command_i\ \{I\}}{\{I\}\ \textbf{\{*}\ guard_1 \text{ --> } command_1\ \textbf{,}\ \ldots\ \textbf{,}\ guard_n \text{ --> } command_n\ \textbf{\}}\ \{I, \wedge i \text{ not } guard_i\}} \text{ for } i=1..n$$

Parallel

$$\frac{\{P\}\ command_i\ \{Q_i\}}{\{P\}\ \textbf{\{||}\ command_1\ \textbf{,...,}\ command_n\ \textbf{\}}\ \{\wedge Q_i\}} \text{ for } i=1..n$$

Exceptions

HTML Style Guide

For consistency and later HTML to LaTeX conversion

- Chapter headings:
  - proceeded by hr
  - centered using H1
  - Followed by text in em which introduces the topic
  - Followed by keywords and phrases introduced with em but in normal type
  - Followed by p hr p
- Section headings: H2 ...
- times is ampersand times
- ö is ampersand ouml
- & gt=, and & lt=
- Top ¯|¯, bottom _|_.
- Symbols: |-->, -->, ->, |-, ==>, in for membership , union
- Greek alphabet various, usually spelled but may be in b, i, or em
- lambda: \ & lambda-
- Code, Equations & Formulas
  - inline: tt ... tt
  - single line: p center tt ... tt center p
  - multi-line: blockquote pre ... pre blockquote
- Font style
  - em -- for keywords defined in the running text
  - b -- for headings in figures, tables and definitions
  - tt -- for code
- Figures and tables: hr p center Figure/Table/Definition M.N b Description b p The Figure/Table/Definition p hr
- Definitions & Principles: p hr h4 Definition I.J h4 blockquote ... blockquote hr p
- Figures: blockquote center Figure N: b ... b center *the Figure* blockquote
  where the figure may be in pre ... pre
- Aside: blockquote b Aside. b content blockquote
- Terminology: blockquote b Terminiology. b content blockquote
- Principle: blockquote b Principle -name- b content blockquote
- Tables: p center ...caption ... caption ... center p
- Sections:
  - Falicies and Pitfalls
  - Concluding Remarks
  - Historical Perspectives and Further Reading:
    - historical remarks,
    - alternatives to the primiary presentation,
    - books and papers

- Exercises
  - [time/difficulty] (section) Problem statement

To Do

- Definitions
- Index
- Code
- Lecture Notes
    - Chapters 1-N
- Lab Manual
    - Labs 1-N
- Problem sets
    - Chapters 1-N
- Rewrite
    - Preface
    - Intro Chapter (models of computation, Definition style)
    - Semantics Chapter
    - Domains Chapter
        - add type inference rules
- In process
    - Logic Programming
- Finished

Miscellaneous Items

Area: Pragmatics

Software engineering

- Problem domain
- PITS vs PITL esp. overhead hello world program; csh vs Java
- Correctness
- Psychological

Implementation

- Architecture
- Compiler technology

Application

- RPG/Visual Basic
- Efficiency--space & time

Implementation

- Syntax
- Virtual Machine to vonNeuman arch
- Data/ADT/Objects
- Code
- Scope

Area: Imperative Programming

Productive Use of Failure

1. Iterators and Generators in Icon (ML, Prolog...)

2. Backtracking in Prolog

In Prolog ... the form ... generator(P) ... fail . Backtracking produces the successive elements of the generator.

```
% Generators

% Natural Numbers

nat(0).
nat(N) :- nat(M), N is M + 1.

% Infinite sequence from I

inf(I,I).
inf(I,N) :- I1 is I+1, inf(I1,N).

% An Alternate definition of natural numbers (more efficient)
alt_nat(N) :- inf(0,N).

% Sequence of Squares

square(N) :- alt_nat(X), N is X*X.

% Infinite Arithmetic Series

inf(I,J,I) :- I =< J.
inf(I,J,N) :- I < J, I1 is J + (J-I), inf(J,I1,N).

inf(I,J,I) :- I > J.
inf(I,J,N) :- I > J, I1 is J + (J-I), inf(J,I1,N).

% Finite Arithmetic Sequences

% Numbers between I and J increment by 1

between(I,J,I) :- I =< J.
between(I,J,N) :- I < J, I1 is I+1, between(I1,J,N).

between(I,J,I) :- I > J.
between(I,J,N) :- I > J, I1 is I-1, between(I1,J,N).

% Numbers between I and K increment by (J-I)

between(I,J,K,I) :- I =< K.
between(I,J,K,N) :- I < K, J1 is J + (J-I), between(J,J1,K,N).

between(I,J,K,I) :- I > K.
between(I,J,K,N) :- I > K, J1 is J + (J-I), between(J,J1,K,N).

% Infinite List -- Arithmetic Series the Prefixes

inflist(N,[N]).
inflist(N,[N|L]) :- N1 is N+1, inflist(N1,L).
```

```
% Primes -- using the sieve

prime(N) :- primes(PL), last(PL,N).

% List of Primes

primes(PL) :- inflist(2,L2), sieve(L2,PL).
sieve([],[]).
sieve([P|L],[P|IDL]) :- sieveP(P,L,PL), sieve(PL,IDL).

sieveP(P,[],[]).
sieveP(P,[N|L],[N|IDL]) :- N mod P  >  0, sieveP(P,L,IDL).
sieveP(P,[N|L],   IDL)  :- N mod P =:= 0, L  [], sieveP(P,L,IDL).

last([N],N).
last([H|T],N) :- last(T,N).

% Primes -- using the sieve (no list)

sprime(N) :- inflist(2,L2), ssieve(L2,N).

ssieve([P],P).
ssieve([P|L],NP) :- L  [], sieveP(P,L,PL), ssieve(PL,NP).

% B-Tree Generator -- Inorder traversal (Order important)
traverse(btree(Item,LB,RB),I) :- traverse(LB,I).
traverse(btree(Item,LB,RB),Item).
traverse(btree(Item,LB,RB),I) :- traverse(RB,I).
```

Area: Domains and Types

Primitive Domains

Among the primitive types provided by programming languages are

- Truth-value = { *false*, *true*}
- Integer = {..., -2, -1, 0, +1, +2, ...}
- Real = { $D^*.D^*$ | D in {0,...,9}}
- Character = {..., a, b, ..., z, ...}

- Vector = { A | A[i] = v for i in I and v in D }
- Record = { (v_1, ..., v_n) | n in I, v_i in D_i}
- Sequential File = { [H|T] | H in D, T in Seq. File }

These are the types whose values are are usually represented as bit patterns in the computer.

> **Aside.** Programming language definitions do not place restrictions on the primitive types. However hardware limitations and variation have considerable influence on actual programs so that, integers are an implementation defined range of whole numbers, reals are an implementation defined subset of the rational numbers and characters are an implementation defined set of characters.

Several languages permit the user to define additional primitive types. These primitive types are called *enumeration* types.

An elementary data object is always manipulated as a unit.

Integer

Specification: finite set of discrete values; arithmetic, relational and assignment operations
Implementation: usually hardware but extended precision is implemented in software

Floating-point reals

Specification: finite set of discrete values; arithmetic, relational, assignment operations, trigonometric, logarithmic and exponent operations
Implementation: usually hardware; exponentiation is often implemented in software

Other numeric

natural, fixed point, complex, and rational numbers

Enumerations

Specification: ordered list of distinct values; relational, successor, predecessor and assignment operations
Implementation: subrange of non-negative integers

Boolean

Specification: two discrete values; and, or, not and assignment operations
Implementation: usually single bit in a byte (zero = false, anything else = true)

Character

Specification: ASCII or other character set; relational and assignment operations
Implementation: usually hardware but extended precision is implemented in software

Compound Domains

Structured data types: A structured data object is constructed as an aggregate of other data objects.

Vector

Specification: fixed number of components of the same type; indexing to access components, create, destroy, and other operations.
Implementation: contiguous storage locations for components

Arrays

Specification: fixed number of components of the same type; indexing to access components, create, destroy, and other operations.
Implementation: contiguous storage locations for components; row major vs column major

Records

Specification: fixed number of components of (possibly) different type; access to components, create, destroy, and other operations.
Implementation: contiguous storage locations for components. r

Pointers

Specification:
Implementation: responsibility of the OS.

Dynamic storage allocation

Sets

Specification:
Implementation: responsibility of the OS.

Files

Specification:
Implementation: responsibility of the OS.

Abstract Types

An abstract type is a type which is defined by its operations rather than its values.

The primitive data types provided in programming languages are abstract types. The representation of integer, real, boolean and character types are hidden from the programmer. The programmer is provided with a set of operations and a high-level representation. The programmer only becomes aware of the lower level when an error such as an arithmetic overflow occurs.

An *abstract data type* consists of a type name and operations for creating and manipulating objects of the type. A key idea is that of the separation of the implementation from the type definition. The actual format of the data is

hidden (information hiding) from the user and the user gains access to the data only through the type operations.

There are two advantages to defining an abstract type as a set of operations. First, the separation of operations from the representation results in data independence. Second, the operations can be defined in a rigorous mathematical manner. As indicated in Chapter Semantics, algebraic definitions provide appropriate method for defining an abstract type. The formal specification of an abstract type can be separated into two parts. A syntactic specification with gives the signature of the operations and a semantic part in which axioms describe the properties of the operations.

In order to be fully abstract, the user of the abstract type must not be permitted access to the representation of values of the type. This is the case with the primitive types. For example, integers might be represented in two's complement binary numbers but there is no way of finding out the representation without going outside the language. Thus, a key concept of abstract types is the hiding of the representation of the values of the type. This means that the representation information must be local to the type definition.

Modula-3's approach is typical. An abstract type is defined using Modules -- a definition module (called an interface), and an implementation module (called a module).

Since the representation of the values of the type is hidden, abstract types must be provided with *constructor* and *destructor* operations. A constructor operation composes a value of the type from values from some other type or types while a destructor operation extracts a constituent value from an abstract type. For example, an abstract type for rational numbers might represent rational numbers as pairs of integers. This means that the definition of the abstract type would include an operation which given a pair of integers returns a rational number (whose representation as an ordered pair is hidden) which corresponds to the the quotient of the two numbers. The rational additive and multiplicative identities corresponding to zero and one would be provided also.

Figure~\ref{complex:adtspec}

Figure 3.N: **Complex numbers abstract type**

```
DEFINITION MODULE} ComplexNumbers;
TYPE Complex;

PROCEDURE MakeComplex ( firstNum, secondNum : Real ) : Complex;
PROCEDURE AddComplex ( firstNum, secondNum : Complex ) : Complex;
PROCEDURE MultiplyComplex ( firstNum, secondNum : Complex ) : Complex;
...
END ComplexNumbers.
```

---

is a definition definition module of an abstract type for complex numbers using Modula-2 and~\ref{complex:adtimp}

Figure 3.N: **Complex numbers implementation**

```
IMPLEMENTATION MODULE ComplexNumbers;
TYPE
```

```
    Complex = POINTER TO} ComplexData;
    ComplexData = RECORD
                     RealPart, ImPart : REAL};
                  END;

PROCEDURE MakeComplex ( firstNum, secondNum : Real ) : Complex;
   VAR result : Complex;

BEGIN
   new( result );
   result^.RealPart := firstNum;
   result^.ImPart := secondNum
   return result
END NewComplex;

PROCEDURE} AddComplex ( firstNum, secondNum : Complex ) : Complex;
   VAR result : Complex;

BEGIN
   new( result );
   result^.RealPart := firstNum^.RealPart +   secondNum^.RealPart;
   result^.ImPart := firstNum^.ImPart + secondNum^.ImPart
   return result
END AddComplex;
...
BEGIN
...
END ComplexNumbers.
```

---

is the corresponding implementation module.

Terminology:

The terms *abstract data type* and *ADT* are also used to denote what we call an *abstract type*.

Generic Types

Given an abstract type stack, the stack items would be restricted to be a specific type. This means that an abstract type definition is required for stacks which differ only in the element type contained in the stack. Since the code required by the stack operations is virtually identical, a programmer should be able to write the code just once and share the code among the different types of stacks. Generic types or *generics* are a mechanism to provide for sharing the code. The sharing provided by generics is through permitting the parameterization of type definitions. Figure~\ref{stack:generic} contains a Modula-2 definition module for a generic stack.

Figure 3.N: **A Generic Stack**

```
DEFINITION MODULE GenericStack;
TYPE stack(ElementType);
PROCEDURE Push ( Element:ElementType; Var Stack : stack(ElementType) );
...
END GenericStack
```

The definition differs from that of an abstract type in that the type name is parameterized with the element type. At compile time, code appropriate to the parameter is generated.

Type Checking type free languages, data type parameterization (polymorphism)

The problem of writing generic sorting routines points out some difficulties with traditional programming languages. A sorting procedure must be able to detect the boundaries between the items it is sorting and it must be able to compare the items to determine the proper ordering among the items. The first problem is solved by parameterizing the sort routine with the type of the items and the second is solved by either parameterizing the sort routine with a compare function or by overloading the relational operators to permit more general comparisons.

Generic packages in Ada is a cheap way to obtain polymorphism. Generic packages are not compiled at compile time, rather they are compiled whenever they are parameterized with a type. So that if a programmer desires to sort a array of integers and an array of reals, the compiler will generate two different sort routines and the appropriate routine is selected at run-time.

Area: Lambda Calculus

From Mathematics

Variables

- Each occurrence of a mathematical variable refers to the same value ($x^2 = 3x + 5$ vs x := x+1)
- A mathematical variable n may represent a fixed but otherwise arbitrary number throughout the discussion. It is *free* in the given context.
- A mathematical variable may run through a set of values -- $\int_0^1 e^x dx$ or $\forall x[(x+1)(x-1)=x^2-1]$ -- the variable x is *bound* in these formulas by a special symbol.
- Bound variable are not always clearly indicated in mathematics -- mathematicians some times write
  $f = x^3 -3x^2 + 3x -1$ when they should write
  $f(x) = x^3 -3x^2 + 3x -1$ or in lambda notation
  $f = \lambda x.x^3 -3x^2 + 3x -1$
  To show that the variable x is bound. The lambda calculus always uses the symbol \lambda to bind variables.

Functions

- Most programming languages do not have function variables.
- The original concept of a function: *finite description of a computational procedure*
- Modern concept of a function: *a set of ordered pairs, the second element is unique*

Domains, Types, & Higher-order functions

- The set of [D-->R] type functions is called the *function space* R^D
- Typed lambda calculus
- functionals -- take functions as arguments and return functions as results

Polymorphic functions and Currying

- A function is polymorphic if the type of (at least one of) its arguments may vary from call to call.
- A function which can take an arbitrary number of arguments is called polyadic -- implmented as functions of one argument - a list
- Currying -- + is of type [N×N-->N] can be rewritten as of type [N-->[N-->N]]

Area: Functional Programming

Functional Forms

Functional languages treat functions as first-class values: they can be passed as parameters, returned as results, built into composite values, and so on. Functions whose parameters are all non-functional are called *first-order*. A function that has a functional parameter is called a *higer-order* function or *functional*. Functional composition is an example of a functional.

double (x) = x * x
quad = double \circ double
twice (f) = f \circ f
odd = not \circ even

Functionals are often used to construct reusable code and to obtain much of the power of imperative programming within the functional paradigm.

An important functional results from *partial application* For example, suppose there is the function {\tt power} defined as follows:

power (n, b) = if n = 0 then 1 else b*power(n-1, b)

As written {\tt power} raises a base {\tt b} to a positive integer power {\tt n}. The expression {\tt power(2)} is a function of one argument which when applied to a value returns the square of the value.

composition
f \circ g (x) = f(g(x))
dispatching
f \& g (x) = (f(x), g(x))
parallel
currying
apply
apply(f,a) = f(a)
iterate
iterate(f,n) (a) = f(f(...(f(a))...))

(\x.\y.((* x) y) 3 = \y.((* 3) y)

Program Transformation

Since functional programs consist of function definitions and expression evaluations they are suitable for program transformation and formal proof just like any other mathematical system. It is the principle of referential transparency that makes this possible. The basic proof rule is that: *identifiers may be replaced by their values*. For example,

```
f 0 = 1
f n+1 = (n+1)*(f n)

fp 0 fn = fn
fp n+1 in = fp n (n+1)*in

f n = fp n 1

f 0 = fp 0 1  by definition of {\sf f} and {\sf fp}

assume f n = fp n 1
show f n+1 = fp n+1 1
f n+1 = (n+1)*f n
= (n+1)*fp n 1
and
k*fp m n = fp m k*n since
1*fp m n = fp m 1*n
and
(k+1)*fp m n = k*fp m n + fp m n = fp m k*n + fp m n
```

fold

unfold

The Lambda Calculus

Aside. which says that an occurrence of x in B can be replaced with e. All bound identifiers in B are renamed so as not to clash with the free identifiers in E.

The operational semantics of the lambda calculus define various operations on lambda expressions which enable lambda expressions to be *reduced* (evaluated) to a *normal form* (a form in which no further reductions are possible).

Thus, the operational semantics of the lambda calculus are based on the concept of substitution. A lambda abstraction denotes a function, to apply a lambda abstraction to an argument we use what is called beta-reduction. The following formula is a formalization of beta-reduction.

(lambda x.B M) <--> B[x:M]

The notation B[x:M] means the replacement of free occurrences of x in B with M.

One problem which arises with beta-reduction is the following. Suppose we apply beta-reduction as follows.

```
( lambda x.B  M) <--> B[x:M]
```

where y is a free variable in M but y occurs bound in B. Then upon the substitution of M for x, y becomes bound. To prevent this from occurring, we need to do the following.

> To prevent free variables from becoming bound requires the replacement of free variables with new free variable name, a name which does not occur in B.

This type of replacement is called alpha-reduction. The following formula is a formalization of alpha-reduction,

Alpha Reduction: lambda x.B <--> lambda y.B[x:y]

where y is not free in B.

Figure N.2: **The Alpha, Beta and Eta Reduction Rules**

Alpha-reduction:	If y does not occur in B
	\x.B --> \y.B[x:y]
Beta-reduction:	(\x.B) e --> B[x:e]
Eta-reduction:	\x.E x --> E

Syntax

Figure .: Two lambda expressions, P and Q, are **identical**, in symbols P \equiv Q, if and only if Q is an exact (symbol by symbol) copy of P.

Note that function application associates to the right.

Figure .: The set of free variables of an \lambda-expression E, denoted by **phi**(E), is defined as follows:

1. **phi**(x) = {x} for any variable x
2. **phi**(\lambda x.P) = **phi**(P) -{x}
3. **phi**((P)Q) = **phi**(P) union **phi**(Q)

Renaming, \alpha-congruence, and substitution

Figure .: Two lambda expressions are considered essentially the same if they differ only in the names of their bound variables.

Figure .: The relation M==> N (read M \beta-**reduces** to N) is defined as follows:

1. M ==> N if M \cong N
2. M ==> N if M --> N is an instance of the \beta-rule
3. If M ==> N for some M and N, then for any \lambda-expression E, both (M)N ==> (N)E and (E)M ==> (E)N
4. If M ==> N for some M and N, then for any variable x, \lambda x.M ==> \lambda x.N also holds.
5. If M ==> E and E ==> N then also M ==> N
6. M ==> N only as specified above

Figure .: M is \beta-**convertable** (or simply **equal**) to N, in symbols M=N, iff M\congN, or M==>N, or N==>M, or there is a \lambda-expression E such that M=E and E=N

The Church-Rosser theorm

Theorem: (Church-Rosser theorem I) If E==> M and E ==> N then there is some Z such that M ==> Z and N ==> Z

If E ==> M and E ==> N, where both M and N are in normal form, then M \cong N

Church-Rosser Theorem I If E_1 \leftrightarrow E_2 then there is an expression E, such that E_1 --> E and E_2 --> E.

Theorem: (Church-Rosser theorem II) If E==> M and E = N then there is a Z such that M ==> Z and N ==> Z

Church-Rosser Theorem II If E_1 --> E_2, and E_2 is in normal form, then there exists a *normal order* reduction sequence from E_1 to E.

Concurrent evaluation

Values and Types

Pre-defined Data Types

Integer, Real, Character, Tuples, Lists Enumerations, algebraic types (unions)

Type Systems and Polymorphism

Pattern matching, Product type, sequences, functions, ML, Miranda

Pattern matching

```
f 0 = 1
f (n+1) = (n+1)*f(n)

insert (item Empty\_Tree) = BST item Empty\_Tree Empty\_Tree
insert (item BST x LST RST) = BST x insert (item LST) RST if item < x
                              BST x LST insert( item RST ) if item > x
```

List of Figures

Introduction

Figure 1: Standard deviation using higher-order functions

```
sd(xs) = sqrt(v)
         where
            n = length( xs )
            v = fold( plus, map(sqr, xs ))/n
                - sqr( fold(plus, xs)/n)
```

Figure 2: Socrates is mortal

1a. `human(Socrates)`	Fact
1b. `human(Penelope)`	Fact
2. `mortal(X) if human(X)`	Rule
3. ¬`mortal(Y)`	Assumption
4a. `X = Y`	from 2 & 3 by unification
4b. ¬`human(Y)`	and modus tollens
5a. `Y = Socrates`	from 1 and 4 by unification
5b. `Y = Penelope`	
6. Contradiction	5a, 4b, and 1a; 5b, 4b and 1b

Figure 3: State Sequence

S_0 -O_0-> S_1 - ... -> S_{n-1} -O_{n-1}-> S_n

Syntax

Figure 2.1: G_0 a grammar for a fragment of English

The grammatical categories are: S, NP, VP, D, N, V.
The words are: a, the, cat, mouse, ball, boy, girl, ran, bounced, caught.
The grammar rules are:

```
S  --> NP VP
NP --> N
NP --> D N
VP --> V
VP --> V NP
V  --> ran | bounced | caught
D  --> a | the
N  --> cat | mouse | ball | boy | girl
```

The most general category is S, a sentence.

Figure 2.2: G_1 An expression grammar

```
N = { E }
T = { c, id, +, *, (, ) }
P = {E --> c,
     E --> id,
     E --> (E),
     E --> E + E,
     E --> E * E }
S = E
```

Figure 2.3: G_2 An abstract expression grammar

```
N = { E }
T = { c, id, add, mult}
P = {E --> c,
     E --> id,
     E --> add E E ,
     E --> mult E E }
S = E
```

Figure 2.4: Top-down Parse

```
PARSE TREE                    UNRECOGNIZED INPUT
```

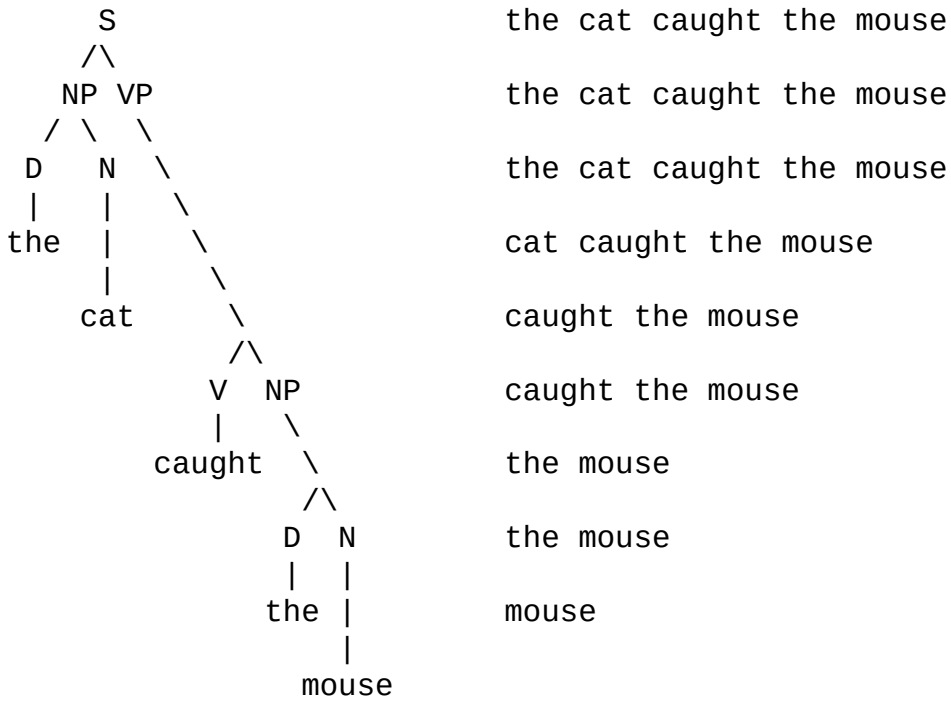
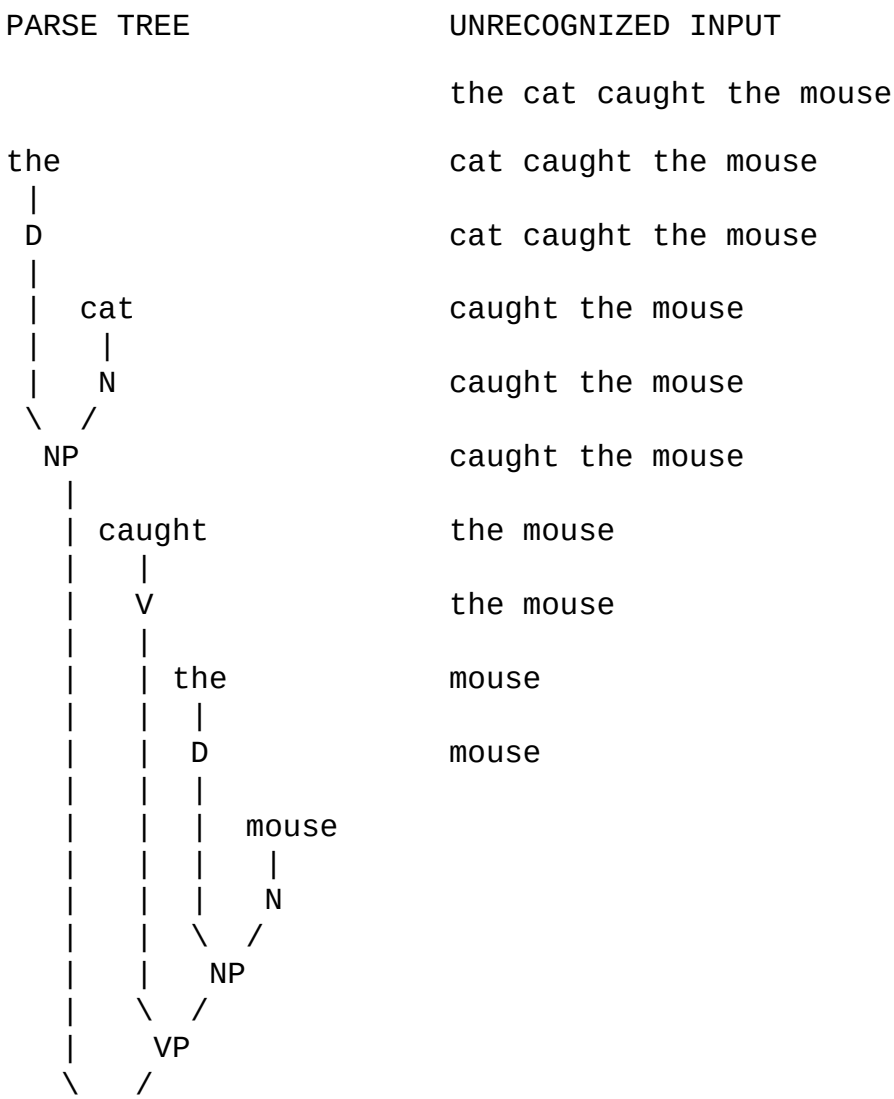


Figure 2.5: Bottom-up Parse



```
      S
```

Figure 2.6 Top-down parse of id+id*id

```
STACK        INPUT          RULE/ACTION

   E]        id+id*id]     pop & push using E --> E+E
 E+E]        id+id*id]     pop & push using E --> id
id+E]        id+id*id]     pop & consume
  +E]          +id*id]     pop & consume
   E]           id*id]     pop & push using E --> E*E
 E*E]           id*id]     pop & push using E --> id
id*E]           id*id]     pop & consume
  *E]             *id]     pop & consume
   E]              id]     pop & push using E --> id
  id]              id]     pop & consume
    ]                ]     accept
```

Figure 2.7: Bottom-up parse of id+id*id

```
STACK    INPUT        RULE/ACTION

      ]  id+id*id]    Shift
    id]    +id*id]    Reduce using E --> id
     E]    +id*id]    Shift
    +E]     id*id]    Shift
  id+E]       *id]    Reduce using E --> id
   E+E]       *id]    Shift
  *E+E]        id]    Shift
id*E+E]          ]    Reduce using E --> id
 E*E+E]          ]    Reduce using E --> E*E
   E+E]          ]    Reduce using E --> E+E
     E]          ]    Accept
```

Figure 2.8: Context-free grammar for Simple

```
program ::= LET definitions IN command_sequence END

definitions ::= e | INTEGER id_seq IDENTIFIER .

id_seq ::= e | id_seq IDENTIFIER ,

command_sequence ::= e | command_sequence command ;
```

```
command := SKIP
        |  READ IDENTIFIER
        |  WRITE exp
        |  IDENTIFIER := exp
        |  IF exp THEN command_sequence ELSE command_sequence FI
        |  WHILE bool_exp DO command_sequence END

exp ::=  exp + term | exp - term  | term
term :: term * factor | term / factor | factor
factor ::= factor^primary  | primary
primary ::=  NUMBER | IDENT | ( exp )
bool_exp ::=  exp = exp | exp < exp | exp > exp
```

Semantics

Figure N.1: Algebraic Definition of Peano Arithmetic

Domains:

Bool = {true, false} (Boolean values)
N in Nat (the natural numbers)
N ::= 0 | S(N)

Semantic functions:

= : (Nat, Nat) -> Bool
+ : (Nat, Nat) -> Nat
× : (Nat, Nat) -> Nat

Semantic axioms and equations:

not S(N) = 0
if S(M) = S(N) then M = N
(n + 0) = n
(m + S(n)) = S(m + n)
(n × 0) = 0
(m × S(n)) = ((m × n) + m)

where m,n in Nat

Figure N.2: Algebraic definition of an Integer Stack ADT

Domains:

Nat (the natural numbers
Stack (of natural numbers)

Bool (boolean values)

Semantic functions:

newStack: () -> Stack
push : (Nat, Stack) -> Stack
pop: Stack -> Stack
top: Stack -> Nat
empty : Stack -> Bool

Semantic axioms:

pop(push(N,S)) = S
top(push(N,S)) = N
empty(push(N,S)) = false
empty(newStack()) = true

Errors:

pop(newStack())
top(newStack()) where N in Nat and S in Stack.

Figure N.3: Program to compute S = $sum_{i=1}^{n}A[i]$

```
S,I := 0,0
while I < n do
   S,I := S+A[I+1],I+1
end
```

Figure N.4: Verification of S = $sum_{i=1}^{n}A[i]$

	Pre/Post-conditions	Code
1.	{ 0 = $Sum_{i=1}^{0}$A[i], 0 < |A| = n }	
2.		S,I := 0,0
3.	{S = $Sum_{i=1}^{I}$A[i], I <= n }	
4.		while I < n do
5.	{S = $Sum_{i=1}^{I}$A[i], I < n }	

6. {S+A[I+1] = $Sum_{i=1}^{I+1}A[i]$, I+1 <= n }

7. S,I := S+A[I+1],I+1

8. { S = $Sum_{i=1}^{I}A[i]$, I <= n }

9. end

10. {S = $Sum_{i=1}^{I}A[i]$, I <= n, I >= n }

11. {S = $Sum_{i=1}^{n}A[i]$ }

Figure N.5: Recursive version of summation

```
S,I := 0,0
loop: if I < n then S,I := S+A[I+1],I+1; loop
      else skip fi
```

Figure N.6: Denotational definition
of Peano Arithmetic

Abstract Syntax:

N in Nat (the Natural Numbers)
N ::= 0 | S(N) | (N + N) | (N × N)

Semantic Algebra:

Nat (the natural numbers (***0***, ***1***, ...)
+ : ***Nat*** -> ***Nat*** -> ***Nat***

Valuation Function:

D : Nat -> ***Nat***

D[(n + 0)] = ***D***[n]
D[(m + S(n))] = ***D***[(m+n)] **+** ***1***
D[(n × 0)] = ***0***
D[(m × S(n))] = ***D***[((m × n) + m)]

where m,n in Nat

Figure N.7: Denotational semantics for Simple

```
Abstract Syntax:

C in Command
E in Expression
O in Operator
N in Numeral
V in Variable

C ::= V := E | if E then C1 else C2 end |
      while E do C3 end | C1;C2 | skip
E ::= V | N | E1 O  E2  | (E)
O ::= + | - | * | / | = | < | > | <>

Semantic Algebra:
Domains:

tau in  T = {true, false}; the  boolean values
zeta in  Z = {...-1,0,1,...}; the integers
+ : Z -> Z -> Z
...
= : Z -> Z -> T
...
sigma in  S = Variable -> Numeral; the state

Valuation Functions:

C in C -> (S -> S)
E in E -> E -> (N union T)

C[ skip ] sigma = sigma
C[ V := E ] sigma = sigma [ V:E[ E ] sigma
C[ C1; C2 ] = C[ C2 ] · C[ C1]
C[ if E then C1 else C2 end ] sigma
          = C[ C1 ] sigma  if E[ E ]sigma = true
          = C[ C2 ] sigma  if E[ E ]sigma = false
C[ while E do C end}]sigma
          = lim_{n -> infty} C[ (if E then C else skip end)^n ] sigma
E[ V ] sigma = sigma(V)
E[ N ] = zeta
E[ E1+E2 ] = E[ E ] sigma + E[ E ] sigma
...
E[ E1=E2 ] sigma = E[ E ] sigma = E[ E ] sigma
```

Figure N.8: Operational semantics for Peano arithmetic

Abstract Syntax:

N in Nat (the natural numbers)

N ::= 0 | S(N) | (N + N) | (N × N)

Interpreter:

I: N -> N

I[(n + 0)] ==> n
I[(m + S(n))] ==> S(*I*[(m+n)])
I[(n × 0)] ==> 0
I[(m × S(n))] ==> *I*[((m × n) + m)]

where m,n in Nat

Figure N.9: Operational semantics for Simple

Interpreter:

I: **C** × **Sigma** -> **Sigma**
{**nu**} in **E** × **Sigma**} -> **T** union **Z**

Semantic Equations:

I(skip,**sigma**) = **sigma**
I(V := E,**sigma**) = **sigma**[V:**nu**(E,**sigma**)]
I(C_1 ;C_2,**sigma**) = *E*(C_2,*E*(C_1,**sigma**))
I(if E then C_1 else C_2 end,**sigma**) = *I*(C_1,**sigma**)&if **nu**(E,**sigma**) = true}
I(C_2,**sigma**)&if **nu**(E,**sigma**) =
false}
while E do C end = if E then (C;while E do C end) else skip
nu(V,**sigma**) = **sigma**(V)
nu(N,**sigma**) = N
nu(E_1+E_2,**sigma**) = **nu**(E_1,**sigma**) + **nu**(E_2,**sigma**)
...
nu(E_1=E_2,**sigma**) = true if **nu**(E,**sigma**) = **nu**(E,**sigma**)}
false if **nu**(E,**sigma**) != **nu**(E,**sigma**)}
otherwise
...

Translation

Figure N.1: Traditional Compiler Structure

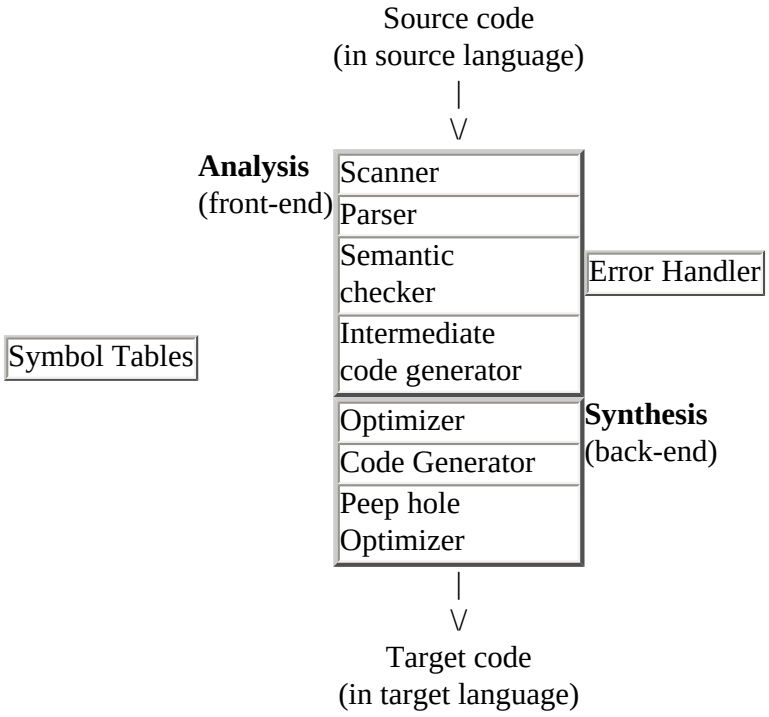


Figure N.2: Context-free grammar for Simple

```
program ::= definitions in command_sequence

definitions ::= e | variable

command_sequence ::= e | command_sequence command ;

command := SKIP
        |  READ variable
        |  WRITE exp
        |  IDENT := exp
        |  IF bool_exp THEN command_sequence ELSE command_sequence FI
        |  WHILE bool_exp DO command_sequence END

exp ::=  exp + term | exp - term  | term
term :: term * factor | term / factor | factor
factor ::= factor^primary  | primary
primary ::=  INT | IDENT | ( exp )
bool_exp ::=  exp = exp | exp < exp | exp > exp
```

Figure N.3: Grammar Transformation Rules

- Convert the grammar to EBNF
- Remove left-recursion: replace N ::= E | NF with N ::= E(F)*
- Left-factor the grammar: replace N ::= EFG | EF'G with N ::= E(F|F')G

- If N ::= E is not recursive, remove it and replace all occurrences of N in the grammar with E

Figure N.2: First[E] and Follow[N]

First[**e**]	= empty set	
First[**t**]	= {t}	t is a terminal
First[N]	= First[E]	where N ::= E
First[E F]	= First[E] union First[F]	if E generates **lambda**
	= First[E]	otherwise
First[E|F]	= First[E] union First[F]	
First[E*]	= First[E]	
Follow[N]	= {t}	in context Nt, t is terminal
	= First[F]	in context NF, F is non-terminal

Figure N.3: EBNF to Parsing Procedures

- For each grammar rule N::=E, construct a parsing procedure

```
parseN {
    parse E
}
```

- Refine *parse E*

If *parse E* is:	**then refine to:**
parse **lambda**	`skip`
parse t	`accept(t)` where t is a terminal
parse N	`parseN` where N is a non-terminal
parse E F	parse E; parse F

parse E|F	`if currentToken.class in First[E] then`
	`parse E`
	`else if currentToken.class in First[F] then`
	`parse F`
	`else` *`report a syntactic error`*

parse E*	`while currentToken.class in First[E] do` *`parse E`*

Figure N.M: RE to EBNF

- Each regular expression RE_i defining a token class T_i is put into the EBNF form: T_i ::= RE_i.

- A regular expression Sep is constructed defining the symbols which sparate tokens.
- The EBNF production S ::= Sep*(T_0|...|T_n) is added to the grammar.

Figure N.3: EBNF to Scanning Procedures

- For each grammar rule T_i::=E_i, construct a scanning procedure `scanT`$_i$ {*scan* E_i}.
- Refine *scan* E_i

scan E_i	**Refinement**
scan **lambda**	`skip`
scan ch	takeIt(t) where ch is a character
scan N	`scanN` where N is a non-terminal
scan E F	scan E; scan F

scan E|F	`if currentChar in First[E] then`
	`scan E`
	`else if currentChar in First[F] then`
	`scan F`
	`else` *`report a syntactic error`*

scan E*	`while currentChar in First[E] do` *`scan E`*

Figure : An attribute grammar for declarations

```
P ::= D(SymbolTable) B(SymbolTable)
D(SymbolTable)  ::= ...V( insert( V in SymbolTable)...

B(SymbolTable) ::= C(SymbolTable)...
C(SymbolTable) ::= V := E(SymbolTable, Error(if V not in SymbolTable)
                   | ...
```

Data Types

Figure M.N: Record implementation

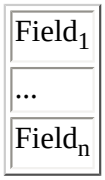


Figure M.N: Object implementation

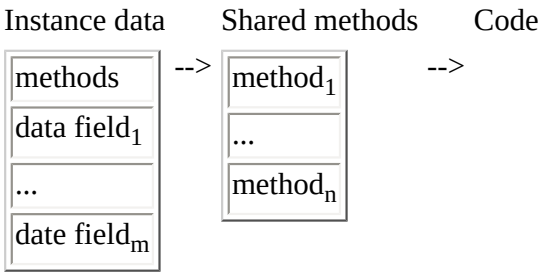
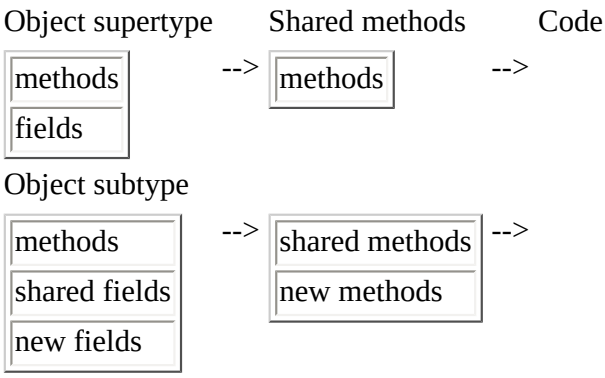


Figure M.N: Implementation of inheritance



Run-time environments

Figure M.N: Simple's Virtual Machine and Runtime Environment

Figure M.N: Virtual Machine 2

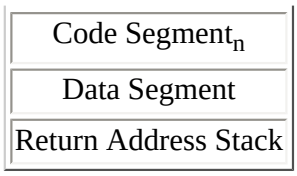


Figure M.N: Virtual Machine 3

Figure M.N: Monolithic Block Structure

F**igure M.N:** Activation Record

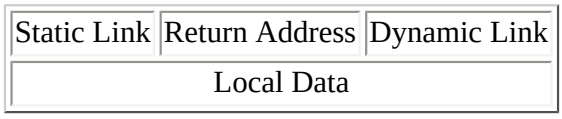


Figure M.N: Display and Runtime Stack

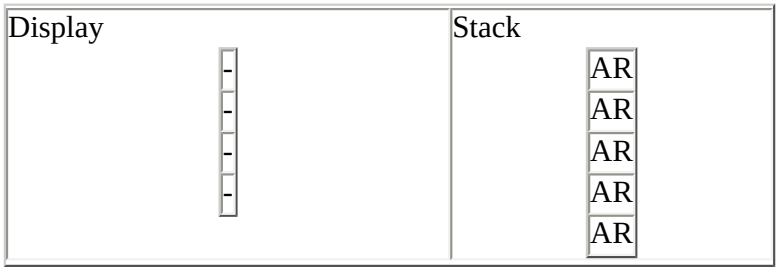
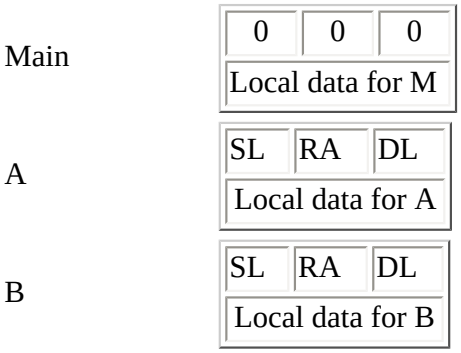


Figure M.N: Runtime Stack



Main calls A which calls B.

Figure M.N: Heap Implmentation

Figure M.N: Pascal Virtual Machine

Every sequential program residing on a computer typically comes in four parts (Figure M.N):

Figure M.N: A sequential program in the computer

code	the procedures, statements, and expressions of the program, translated into machine language by a compiler or assembler.
global data	the top-level (static) variables of the program
heap data	a pool of storage used by the `NEW` function when allocating new dynamic variables
stack	a storage area used to hold all local variables, procedure parameters, and bookkeeping information during execution

Program Counter	Code
Static pointer	Global data
Heap Manager	Heap
Stack Manager	Stack

The program counter keeps track of the location of the next instruction in memory. The stack manager allocates and deallocates stack memory. The heap manager allocates and deallocates heap memory. Code is often kept in read-only memory, because it is never changed during execution. The size of the static area is determined when the program is linked by adding up the sizes of the static data areas in each module. The heap starts out as an area of unused storage; each time additional memory is requested, a piece is allocated to the program. Either a garbage collector or the program may return storage for future use. The stack starts out as unused, but every time a prodedure is called a chunk of stroage at the top of the stack is reserved to hold the procedure's parameters and local variables. Every time a procedure returns, the storage it used on the stack is returned.

Figure M.N: A multithreaded program in the computer

$Thread_1$	...	$Thread_n$	Shared memory
Code pointer	...	Code pointer	Code
Static pointer		Static pointer	Static and shared data
Heap access		Heap access	Heap
Stack manager		Stack manager	
Local stack		Local stack	

Abstraction and Generalization

List of Definitions

Syntax

Definition 2.1: Alphabet and Language

Sigma
: An *alphabet* **Sigma** is a nonempty, finite set of symbols.

L
: A *language* L over an alphabet **Sigma** is a collection of strings of elements of **Sigma**. The *empty string* **lambda** is a string with no symbols at all.

Sigma*
: The set of all possible finite strings of elements of **Sigma** is denoted by **Sigma***. **Lambda** is an element of **Sigma***.

Definition 2.2: Context-free grammar

Context-free grammar G is a quadruple

$$G = (V, T, P, S)$$

where

V is a finite set of variable symbols,
T is a finite set of terminal symbols disjoint from V,
P is a finite set of rewriting rules (productions) of the form

```
A --> w where A in V, w in (V union T)*
```

S is an element of V called the *start* symbol.

Definition 2.3: Generation of a Language from the Grammar

Let G be a grammar. Then the set

$$L(G) = \{w \text{ in } T^* \mid S ==>^* w\}$$

is the language generated by G.

A language L is context-free iff there is a context-free grammar G such that L = L(G).

If w in L(G), then the sequence

$$S ==> w_1 ==> w_2 ==> ... ==> w_n ==> w$$

is a *derivation* of the sentence w and the w_i are called *sentential form*

Definition 2.5: Derivation Tree

Let G = (V, T, P, S) be a context-free grammar. A *derivation tree* has the following properties.

1. The root is labeled S.
2. Every interior vertex has a label from V.
3. If a vertex has label A in V, and its children are labeled (from left to right) a_1, ..., a_n, then P must contain a production of the form

   $$A --> a_1...a_n$$

4. Every leaf has a label from T union {**lambda**}.

Definition 2.6: Ambiguous Grammar

A context-free grammar G is said to be ambiguous if there exists some w in L(G) which has two distinct derivation trees.

Definition 2.6: Push-down automaton

A *push-down automaton* M is a 7-tuple (Q, **Sigma**, **Tau**, **delta**, q_0, Z_0, F)

Q	is a finite set of states
Sigma	is a finite alphabet called the input alphabet
Tau	is a finite alphabet called the stack alphabet
delta	is a transition function from Q× (**Sigma** union {e})× **Tau** to finite subsets of Q× **Tau***
q_0	in Q is the initial state
Z_0	in **Tau** is called the start symbol
F	a subset of Q; the set of accepting states

Definition 2.7: Regular expressions and Regular languages

Regular Expression E	**Language Denoted L(E)**	
ø	ø	The empty set; language
lambda	{**lambda**}	empty string; language which consists of the empty string
a	{ a }	a; the language which consists of just a
(E ·F)	{uv | u in L(E) and v in L(F) }	concatenation;
(E|F)	{u | u in L(E) or u in L(F) }	alternation; union of L(E) and L(F)
(E^*)	{$u_1u_2...u_n$| u_i in L(E) 0 <= i <=n, n >=0 }	any sequence from E

Definition 2.8: Finite State Automaton

A *finite state automaton* or *fsa* is defined by the quintuple

$$M = (Q, \textbf{Sigma}, \textbf{delta}, q_0, F),$$

where

Q is a finite set of *internal states*
Sigma is a finite set of symbols called the *input alphabet*
delta: $Q \times$ **Sigma** --> 2^Q is a total function called the *transition function*
q_0 in Q is the *initial state*
F a subset of Q is the set of *final states*

Preface

This text is built around the observation that programming languages are based on three fundamental concepts:

- Abstraction and generalization
- Data and data structuring
- Computational models

Theory is approached intuitively and motivated with prototypical examples.

Three approaches

- Mathematical: work from fundamental principles to practice
- Scientific: collect data, construct theories
- Popular culture: survey

It is the purpose of this text to explain the concepts underlying programming languages and to examine the major language paradigms that use these concepts.

Programming languages can be understood in terms of a relatively small number of concepts. In particular, a programming language is syntactic realization of one or more computational models. The relationship between the syntax and the computational model is provided by a semantic description. Semantics provide meaning to programs. The computational model provides much of the intuition behind the construction of programs. When a programming language is faithful to the computational model, programs can be more easily written and understood.

The fundamental concepts are supported bindings, abstraction and generalization. Concepts so fundamental that they are included in virtually every programming language. These concepts support the human facility for simile and metaphor which are so necessary in problem solving and in managing complexity.

Programming languages are also shaped by pragmatic considerations. Formost among these considerations are safety, efficiency and applicability. In some languages these external forces have played a more important role in shaping the language than the computational model to the point of distorting the language and actually limiting the applicability of the language. There are several distinct computational models --- imperative, functional, and logic. While these models are equivalent (all computable functions may be defined in each model), there are pragmatic reasons for prefering one model over the another.

This text is designed to formalize and consolidate the knowledge of programming languages gained in

the introductory courses a computer science curriculum and to provide a base for further studies in the semantics and translation of programming languages. It aims at covering the subject area *PL: Programming Languages* as described in the ``ACM/IEEE Computing Curricula 1991."

Special Features of the Text

The following are significant features of the text as compared to the standard texts.

- Syntax: an introduction to regular expressions, scanning, context-free grammars, parsing, attribute grammars and abstract grammars.
- Semantics: introductory treatment of algebraic, axiomatic, denotational and operational semantics.
- Programming Paradigms: the major programming paradigms are prominently featured.
    - Functional: includes an introduction to the lambda calculus and uses the programming languages Scheme and Haskell for examples
    - Logic: includes an emphasis on the formal semantics of Prolog
    - Concurrent: introduces both low- and high-level notations for concurency, stresses the importance of the logic and functional paradigms in the debate on concurrency, and uses the programming language SR for examples.
    - Object-oriented: uses the programming language Modula-3 for examples
- Language design principles: Twenty some programming language design principles are given prominence. In particular, the importance of abstraction and generalization is stressed.

Readership

This book is intended as an undergraduate text in the theory of programming languages. To gain maximum benefit from the text, the reader should be familiar with discrete mathematics, basic data structures, abstract data types, recursive algorithms, assembly level machine organization and fundamental problem solving concepts. In terms of the ``ACM/IEEE Computing Curricula 1991", AL1-AL3, AR4 and SE1.

Computer science is not a spectator sport. To gain maximum benefit from the text, the reader should construct programs in each of the paradigms, write semantic specifications; and implement a small programming language.

Organization

Since the subject area *PL: Programming Languages* as described in the ``ACM/IEEE Computing Curricula 1991" consists of a minimum of 47 hours of lecture, the text contains considerably more material than can be covered in a single course.

The first part of the text consists of chapters 1-3.
Chapter 1 is an overview of the text. It introduces the themes of the text: models of computation, syntax, semantics, abstraction, generalization and pragmatics. It is a philosopy for the design and

implementation of programming languages and a context for understanding programming languages.
Chapter 2 focuses on systax for the structural description of programming languages. It is an introductory treatment of context-free grammars, push-down automata, regular expressions, and finite state machines. Context-free grammars are utilized throughout the text and the material is a prerequisite for Chapter ??
Chapter 3 introduces semantics: algebraic, axiomatic, denotational and operational. While the chapter is optional, I introduce algebraic semantics in conjunction with abstract types and axiomatic semantics with imperative programming.
Chapter 4 is a formal treatment of abstraction and generalization as used in programming languages.
Chapter 5 deals with values, types, type constructors and type systems.
Chapter 6 deals with environments, block structure and scope rules.
Chapter 7 deals with the functional model of computation. It introduces the lambda calculus and examines Scheme and Haskell.
Chapter 8 deals with the logic model of computation. It introduces Horn clause logic, resolution and unification and examines Prolog.
Chapter 9 deals with the imperative model of computation. Features of several imperative programming languages are examined. Various parameter passing mechanisms should be discussed in conjunction with this chapter.
Chapter 10 deals with the concurrent model of programming. Its primary emphasis is from the imperative point of view.
Chapter 11 is a further elaboration of the concepts of abstraction and generalization in the module concept. It is preparatory for Chapter 12.
Chapter 12 deals with the object-oriented model of programming. Its primary emphasis is from the imperative point of view. Features of Smalltalk, C++ and Modula-3 provide examples.
Chapter 13 deals with pragmatic issues and implementation details. It may be read in conjunction with earlier chapters.
Chapter ?? deals with parsing, compiling and attribute grammars.
Chapter 14 deals with programming environments,
Chapter 15 deals with the evaluation of programming languages and a review of programming language design principles.
Chapter 16 contains a short history of programming languages.

Pedagogy

The text provides pedagogical support through various exercises and laboratory projects. Some of the projects are suitable for small group assignments. The exercises include programming exercises in various programming languages. Some are designed to give the student familiarity with a programming concept such as modules, others require the student to construct an implementation of a programming language concept. For the student to gain maximum benefit from the text, the student should have access to a logic programming language (such as Prolog), a modern functional language (such as Scheme, ML or Haskell), a concurrent programming language (Ada, SR, or Occam), an object-oriented programming language (C++, Small-Talk, Eiffel, or Modula-3), and a modern programming environment and programming tools. Free versions of Prolog, ML, Haskell, SR, and Modula-3 are available from one or more ftp sites and are recommended.

The instructor's manual contains lecture outlines and illustrations from the text which may be transferred to transparencies. There is also a laboratory manual which provides short introductions to Lex, Yacc, Prolog, Haskell, Modula-3, and SR.

The text has been used as a semester course with a weekly two hour lab. Its approach reflects the core area of programming languages as described in the report {\bf Computing as a Discipline} in CACM January 1989 Volume 32 Number 1.

Knowledge Unit Mapping

To assist in curriculum development, the follow mapping of the ACM knowledge units to the text is provided.

Knowledge Unit	**Chapter(s)**
PL1: History	6,7,8,9,11
PL2: Virtual Machines	2,6,7,8,13
PL3: Data Types	5,13
PL4: Sequence Control	9,10
PL5: Data Control	5,11,12
PL6: Run-time	2,13
PL7: Regular Expressions	2
PL8: Context-free grammars	2
PL9: Translation	2
PL10: Semantics	3
PL11: Programming Paradigms	1,7,8,9,10,12
PL12: Parallel Constructs	10
SE3: Specifications	3

Acknowledgements

There are several programming texts that have influenced this work in particular, texts by Hehner, Tennent, Pratt, and Sethi. I am grateful to my CS208 classes at Bucknell for their comments on preliminary versions of this material and to Bucknell University for providing the excellent environment in and with which to develop this text.

AA 1992

Introduction

A complete description of a programming language includes the computational model, the syntax and semantics of programs, and the pragmatic considerations that shape the language.

Keywords and phrases: Computational model, computation, program, programming language, syntax, semantics, pragmatics, bound, free, scope, environment, block.

Suppose that we have the values `3.14` and `5`, the operation of multiplication (×) and we perform the computation specified by the following arithmetic expression

```
2 × 3.14 × 5
```

the result of which is the value:

```
31.4
```

If `3.14` is an approximation for `pi`, we can replace `3.14` with `pi` *abstracting* the expression to:

```
2 × pi × 5 where pi = 3.14
```

We say that `pi` is *bound* to `3.14` and is a *constant*. The `where` introduces a *local environment* or *block* for local definitions. The *scope* of the definitions is just the expression. If `5` is intended to be the value of a radius, then the expression can be *generalized* by introducing a variable for the radius:

```
2 × pi × radius where pi = 3.14
```

Of course the value of the expression is the circumference of a circle so we may further abstract by assigning a name to the expression:

```
Circumference = 2 × pi × radius where pi = 3.14
```

This last equation binds the name `Circumference` to the expression `2 × pi × radius where pi=3.14`. The variable `radius` is said to be *free* in the right hand side of the equation. It is a variable since its value is not determined. `pi` is not a variable, it is a *constant*, the name of a particular value. Any context (*scope*), in which this equation and the variable `radius` appears and `radius` is assigned to a value, determines a value for `Circumference`. A further generalization is possible by *parameterizing* `Circumference` with the variable `radius`.

```
Circumference(radius) = 2 × pi × radius where pi = 3.14
```

The variable `radius` appearing in the right hand side is no longer free. It is bound to the parameter `radius`. `Circumference` has a value (other than the right hand side) only when the parameter is replaced with an expression. For example, in

```
Circumference(5) = 3.14
```

The parameter `radius` is *bound* to the value `5` and, as a result, `Circumference(5)` is bound to `3.14`. In this form, the definition is a recipe or program for computing the circumference of a circle from the radius of the circle. The mathematical notation (*syntax*) provides the programming language and arithmetic provides the *computational model* for the computation. The mapping from the syntax to the computational model provides the meaning (*semantics*) for the program. The notation employed in this example is based on the very pragmatic considerations of ease of use and understanding. It is so similar to the usual mathematical notation that most people have difficulty in distinguishing between the syntax and the computational model.

This example serves to illustrate several key ideas in the study of programming languages which are summarized in definition 1.1.

Definition 1.1

1. A *computational model* is a collection of values and operations.
2. A *computation* is the application of a sequence of operations to a value to yield another value.
3. A *program* is a specification of a computation.
4. A *programming language* is a notation for writing programs.
5. The *syntax* of a programming language refers to the structure or form of programs.
6. The *semantics* of a programming language describe the relationship between a program and the model of computation.
7. The *pragmatics* of a programming language describe the degree of success with which a programming language meets its goals both in its faithfulness to the underlying model of computation and in its utility for human programmers.

Data

A program can be viewed as a function, the output data values are a function of the input data values.

Output = Program(Input)

Another view of a program is that it models a problem domain and the execution of the program is a simulation of the problem domain.

Program = Model of a problem domain
Execution of a program = simulation of the problem domain

In any case, data objects are central to programs.

The values can be separated into two groups, primitive and compound. The primitive values are usually numbers, boolean values, and characters. The composite values are usually arrays, records, and recursively defined values. Strings may occur as either primitive or composite values. Lists, stacks, trees, and queues are examples of recursively defined values.

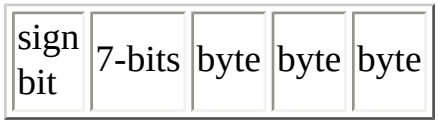
Associated with the primitive values are the usual operations (e.g., arithmetic operations for the numbers). Associated with each composite value are operations to construct the values of that type and operations to access component elements of the type. A collection of values that share a common set of operations is called a data type.

The primitive types are implemented using the underlying hardware and, sometimes, special purpose software. So that only appropriate operations are applied to values, the value's type must be known. In assembly language programs it is up to the programmer to keep track of a datum's type. Type information is contained in a descriptor.

When the type of a value is known at compile time the type descriptor is a part of the compiler's symbol table and the descriptor is not needed at run-time and therefore, the descriptor is discarded after compilation. When the type of a value is not known until run-time, the type descriptor must be associated with the value to permit type checking.

Boolean values are implemented using a single bit of storage. Since single bits are not usually addressable, the implementation is extended to be a single addressable unit of memory. In this case either a single bit within the addressable unit is used for the value or a zero value in the storage unit designates false while any non-zero value designates true. Operation on bits and boolean values are included in processor instruction sets.

Integer values are most often implemented using a hardware defined integer storage representation, often 32-bits or four bytes with one bit for the sign.



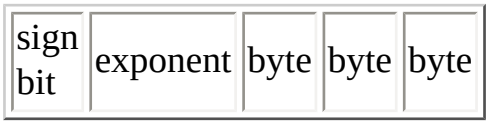
The integer arithmetic and relational operations are implemented using the set of hardware operations. The storage unit is divided into a sign and a binary number. Since the integers form an infinite set, only a subrange of integers is provided. Some languages (for example Lisp and Scheme) provide for a

greatly extended range by implementing integers in lists and providing the integer operations in software. This provides for ``infinite" precision arithmetic.

Natural number values are most often implemented using the hardware defined storage unit. The advantage of providing an natural number type is that an additional bit of storage is available thus providing larger positive values than are provided for integer values.

Rational number values may be implemented as pairs of integers. Rationals are provided when it is desired to avoid the problems of round off and truncation which occurs when floating point numbers are used to represent rational numbers.

Real number values are most often implemented using a hardware defined floating point representation. One such representation consists of 32-bits or four bytes where the first bit is the sign, the next seven bits the exponent and the remaining three bytes the mantissa.



The floating point arithmetic and relational operations are implemented using the set of hardware operations. Some floating point operations such as exponentiation are provided in software. The storage unit is divided into a mantissa and an exponent. Sometimes more than one storage unit is used to provide greater precision.

Character values are almost always supported by the underlying hardware and operating system, usually one byte per character.
Characters are encoded using the 8-bit ASCII or EBCDIC encoding scheme or the emerging 16-bit Unicode encoding scheme.

Enumeration values are usually represented by a subsequence of the integers and as such inherit an appropriate subset of the integer operations.

Where **strings** are treated as a primitive type, they are usually of fixed length and their operations are implemented in hardware.

Compound (or structured) data types include arrays, records, and files.

Abstract data types are best implemented with pointers. The user program holds a pointer to a value of the abstract type. This use of pointers is quite safe since the pointer manipulation is restricted to the implementation module and the pointer is notationally hidden.

Models of Computation

There are three basic computational models -- functional, logic, and imperative. In addition to the set

of values and associated operations, each of these computational models has a set of operations which are used to define computation. The functional model uses function application, the logic model uses logical inference and the imperative model uses sequences of state changes.

The Functional Model

The *functional model* of computation consists of a set of values, functions, and the operations of function application and function composition. In addition to the usual values, functions can take other functions as arguments and return functions as results (higher-order functions). A program is a collection of definitions of functions and a computation is function application (evaluation of an expression).

The initial example of the computation of the circumference of a circle is an example of functional programming. A more interesting example is a program to compute the standard deviation of a list of scores. The formula for standard deviation is:

$$\mathbf{sigma} = \mathrm{sqrt}\left(\left(\mathbf{Sigma}_{i=1}^{N} x_i^2\right)/N - \left[\left(\mathbf{Sigma}_{i=1}^{N} x_i\right)/N\right]^2 \right)$$

where x_i is an individual score and N is the number of scores. The formula requires computing both the sum of the scores and the sum of the squares of the scores. The higher-order function `map` applies a function to each element of a list and the higher-order function `fold` reduces a list by applying a function to the first element of a list and the result of folding the rest of the list. Figure 1 illustrates what an implementation in a functional programming language might look like.

Figure 1.1: Standard deviation using higher-order functions

```
sd(xs) = sqrt(v)
         where
            n = length( xs )
            v = fold( plus, map(sqr, xs ))/n
                - sqr( fold(plus, xs)/n)
```

The functional model is important because it has been under development for hundreds of years and its notation and methods form the base upon which a large portion of our problem solving methodologies rest. The prime concern in functional programming is defining functional relationships.

Figure 1.2: Functional Programming

values functions function definition function application function composition	Program = set of function definitions Computation = function application

---

The Logic Model

The *logic model* of computation consists of a set of values, definitions of relations and logical inference. Programs consist of definitions of relations and a computation is a proof (a sequence of inferences). For example the earlier circumference computation can be represented as:

```
circle(R, C) if Pi = 3.14 and C = 2 * pi * R.
```

The function is represented as a relation between `R` and `C`.

A better illustration logic programming is a program to determine the mortality of Socrates and Penelope. We begin with the fact that *Socrates and Penelope are human* and the rule that *all humans are mortal* or equivalently *for all X, if X is human then X is mortal*. An equivalent form of the fact and rule are:

```
human(Socrates)
human(Penelope)
mortal(X) if human(X)
```

To determine the mortality of Socrates or Penelope we make the assumption that there are no mortals i. e.

```
¬mortal(Y)
```

Figure 2 contains a computation (proof) that Socrates and Penelope are mortal.

Figure 1.3: Socrates is mortal

1a. `human(Socrates)`	Fact
1b. `human(Penelope)`	Fact
2. `mortal(X) if human(X)`	Rule
3. `¬mortal(Y)`	Assumption
4a. `X = Y`	from 2 & 3 by unification

4b. ¬human(Y)	and modus tollens
5a. Y = Socrates	from 1 and 4 by unification
5b. Y = Penelope	
6. Contradiction	5a, 4b, and 1a; 5b, 4b and 1b

The first step in the computation is the deduction of line 4 from lines 2 and 3. It is justified by the inference rule *modus tollens* which states that if the conclusion of a rule is known to be false, then so is the hypothesis. The variables X and Y may be *unified* since they may have any value. By unification, Lines 5a, 4b, and 1a; 5b, 4b and 1b produce contradictions and identify both Socrates and Penelope as mortal.

Resolution is the an inference rule which looks for a contradiction and it is facilitated by *unification* which determines if there is a substitution which makes two terms the same. The logic model is important because it is a formalization of the reasoning process. It is related to relational data bases and expert systems. The prime concern in logic programming is defining relationships.

Figure 1.4: Logic Programming

values relations logical inference	Program = set of relation definitions Computation = constructive proof (inference from definitions)

Inferences

program = set of axioms -- the formalization of knowledge *computation = constructive proof of a goal statement from the program*

The Imperative Model

The *imperative model* of computation consists of a set of values including a state and the operation of assignment to modify the state. *State* is the set of name-value pairs of the constants and variables. Programs consist of sequences of assignments and a computation is a sequence of states. Each step in the computation is the result of an assignment operation (See Figure 3).

Figure 1.5: State Sequence

$$S_0 \text{ -}O_0\text{-> } S_1 \text{ - ... -> } S_{n-1} \text{ -}O_{n-1}\text{-> } S_n$$

For example, an imperative implementation of the earlier circumference computation might be written as:

```
constant pi = 3.14

input (Radius) Circumference := 2 * pi * Radius Output (Circumference)
```

The computation requires the implementation to determine the value of `Radius` and `pi` in the state and then change the state by pairing `Circumference` with a new value.

It is easier to keep track of the state when state information is included with the code.

```
constant pi = 3.14

Radius _|_, Circumference = _|_, pi=3.14
input (Radius)
Radius x, Circumference = _|_, pi=3.14
Circumference := 2 * pi * Radius
Radius x, Circumference = 2 × x × pi, pi=3.14
Output (Circumference)
Radius x, Circumference = 2 × x × pi, pi=3.14
```

where `_|_` designates an undefined value.

The imperative model is often called the *procedural model* because groups of operations are abstracted into *procedures*.

The imperative-procedural model is important because it models change and changes are an integral part of our environment. It is the model of computation that is closest to the hardware on which programs are executed. Its closeness to hardware makes it the easiest to implement and imperative programs tend to make the least demands for system resources (time and space). The prime concern in imperative programming is defining a sequence of state changes.

Figure 1.6: Imperative Programming

memory cells values commands	Program = sequence of commands Computation = sequence of state changes

Computability

The functional, logic and imperative models of computation are equivalent in the sense that any problem that has a solution in one model is solvable (in principle) each of the other models. Other models of computation have been proposed. The other models have been shown to be equivalent to these three models. These are said to be *universal* models of computation.

The method of computation provided in a programming language is dependent on the model of computation implemented by the programming language. Most programming languages utilize more than one model of computation but one model usually predominates. Lisp, Scheme, and ML are based on the functional model of computation but provide some imperative constructs while, Miranda and Haskell provide a nearly pure implementation of the functional model of computation. Prolog provides a partial implementation of the logic computational model but, for reasons of efficiency and practicality, fails in several areas and contains imperative constructs. The language Gödel is much closer to the ideal. The imperative model requires some functional and logical elements and languages such as Pascal, C/C++, Ada and Java emphasize assignments, methods of defining various computation sequences and provide minimal implementations of the functional and logic model of computation.

Syntax and Semantics

Syntax describes the structure of programs and semantics defines the relationship between the syntax and the computational model. To simplify the task of reasoning about programs, the syntax of a programming language should be closely related to the computational model. The key principle is the principle of clarity.

Principle of Clarity

> The structure of a programming language should be well defined, and the outcome of a particular section of code easily predicted.

The notation used in the functional and logic models reflects common mathematical practice and exhibits the notational simplicity and regularity found in that discipline. Because the notation used for the imperative model must be able to specify both a variety of state sequences and expressions, it tends to be irregular and of greater complexity than the notation for functional and logic models. Because of this complexity and the wide spread use of imperative programming languages, the bulk of the work done in the area of programming language semantics deals with imperative programming languages.

Pragmatics

Pragmatics is concerned about the usability of the language, the application areas, ease of implementation and use, and the language's success in fulfilling its design goals. The forces that shape a programming language include computer architecture, software engineering practices (especially the software life cycle), computational models, and the application domain (e.g. user interfaces, systems

programming, and expert systems).

For a language to have wide applicability it must make provision for abstraction, generalization and modularity. *Abstraction* (associating a name with an object and using the name to whenever the object is required) permits the suppression of detail and provides constructs which permit the extension of a programming language. These extensions are necessary to reduce the complexity of programs. Generalization (replacing a constant with a variable) permits the application of constructs to more objects and possibly to other classes of objects. *Modularity* is a partitioning of a program into sections usually for separate compilation and into libraries of reusable code. Abstraction, generalization and modularity ease the burden on a programmer by permitting the programmer to introduce levels of detail and logical partitioning of a program. The implementation of the programming language should be faithful to the underlying computational model and be an efficient implementation.

Concurrent programming involves the notations for expressing potential parallel execution of portions of a program and the techniques for solving the resulting synchronization and communication problems. The concurrent programming may be implemented within any of the computational models. Concurrency within the functional and logic model is particularly attractive since, subexpression evaluation and inferences may be performed concurrently and requires no additional syntax. Concurrency in the imperative model requires additional syntactic elements.

Object-oriented programming OOP involves the notations for structuring a program into a collection of objects which compute by exchanging messages. Each object is bound up with a value and a set of operations which determine the messages to which it can respond. The objects are organized hierarchically and inherit operations from objects higher up in the hierarchy. Object-oriented programming may be implemented within any of the other computational models.

Programs are written and read by humans but are executed by computers. Since both humans and computers must be able to understand programs, it is necessary to understand the requirements of both classes of users.

The native programming languages of computers bear little resemblance to natural languages. Machine languages are unstructured and contain few, if any, constructs resembling the level at which humans think. The instructions typically include arithmetic and logical operations, memory modification instructions and branching instructions. For example, the circumference computation might be written in assembly language as:

```
Load Radius R1
Mult R1 2 R1
Load Pi R2
Mult R1 R2 R1
Store R1 Circumference
```

Because the imperative model is closer to actual hardware, imperative programs have tended to be more efficient in their use of time and space than equivalent functional and logic programs.

Natural languages are not suitable for programming languages because humans themselves do not use natural languages when they construct precise formulations of concepts and principles of particular knowledge domains. Instead, they use a mix of natural language, formalized symbolic notations of mathematics and logic and diagrams. The most successful of these symbolic notations contain a few basic objects which may be combined through a few simple rules to produce objects of arbitrary levels of complexity. In these systems, humans reduce complexity by the use of definitions, abstractions, generalizations and analogies. Successful programming languages do the same by catering to the natural problem solving approaches used by humans. Ideally, programming languages should approach the level at which humans reason and should reflect the notational approaches that humans use in problem solving and must include ways of structuring programs to ease the tasks of program understanding, debugging and maintenance.

Language Design Principles

Programming languages are largely determined by the importance the language designers attach to the the computational model, the intended application domain readability, write-ability and efficient execution. Some languages are largely determined by the necessity for efficient implementation and execution. Others are designed to be faithful to a computational model.

Research in computer architecture is producing Research in programming language implementation is producing more efficient implementations of programming languages for all models of computation.

Research in software engineering is producing a better understanding of the program structuring techniques that lead to programs that are easier to write, read (understand), and maintain.

All general purpose programming languages adhere to the following programming language design principle.

Principle of Computational Completeness
: The computational model for a general purpose programming language must be *universal*.

Principle of Implementation
: The implementation should be efficient in its use of space and time.

Principle of Programming The program should be written in a language that reflects the problem domain.

The line of reasoning developed above may be summarized in the following principle.

Principle of Programming Language Design
: A programming language must be designed to facilitate *readability* and *writ-ability* for its human users and efficient execution on the available hardware.

Readability and write-ability are facilitated by the following principles.

Principle of Simplicity

The language should be based upon as few

Principle of Orthogonality

Independent functions should be controlled by independent mechanisms.

Principle of Regularity

A set of objects is said to be regular with respect to some condition if, and only if, the condition is applicable to each element of the set.

Principle of Extensibility

New objects of each syntactic class may be constructed (defined) from the basic and defined constructs in a systematic way.

The principle of regularity and and extensibility require that the basic concepts of the language should be applied consistently and universally.

In the following pages we will study programming languages as the realization of computational models, semantics as the relationship between computational models and syntax, and associated pragmatic concerns.

Historical Perspectives and Further Reading

For a programming languages text which presents programming languages from the virtual machine point of view see Pratt, from the point of view of denotational semantics see Tennent, and from a programming methodology point of view see Hehner.

- Hehner, E. C. R. (1984)
  The Logic of Programming Prentice-Hall International.
- Pratt, T. W. and Zelkowitz, M. V. (1996)
  Programming Languages: Design and Implementation 3rd ed. Prentice-Hall.
- Tennent, R. D. (1981)
  Principles of Programming Languages Prentice-Hall International.

Exercises

1. Identify the applicable scope rules in Figure 2.
2. Construct a trace of the execution of the following program (i.e. complete the following proof).
    1. parentOf(john, mary).
    2. parentOf(kay, john).
    3. parentOf(bill, kay).
    4. ancestorOf(X,Y) if parentOf(X,Y).
    5. ancestorOf(X,Z) if parentOf(X,Y) and ancestorOf(Y,Z).
    6. not ancestorOf(bill,mary).
3. Construct a trace of the execution of `fac(4)` given the function definition

```
fac(N) = if N = 0 then 1
         else N*fac(N-1)
```

4. Construct a trace of the execution of the following program

```
N := 4;
F := 1;
While N > 0 do
   F := N*F;
   N := N-1;
end;
```

5. Using the following definition of a list,

```
list([ ]) -- the empty list
list([X|L]) if list(L) -- first element is X the rest of
                          the list is L
[X0,...Xn] is an abbreviation for [X0|[...[Xn|[ ]]...]
```

   complete the following computation (proof) and determine the result of concatenating the two lists.

   1. `concat([ ],L,L)` Fact
   2. `concat([X|L0],L1,[X|L2]) if concat(L0,L1,L2)` Rule
   3. `¬concat([0,1],[a,b],L)` Assumption

6. Classify the following languages in terms of a computational model: Ada, APL, BASIC, C, COBOL, FORTRAN, Haskell, Icon, LISP, Pascal, Prolog, SNOBOL.
7. For the following applications, determine an appropriate computational model which might serve to provide a solution: automated teller machine, flight-control system, a legal advice service, nuclear power station monitoring system, and an industrial robot.
8. Compare the syntactical form of the if-command/expression as found in Ada, APL, BASIC, C, COBOL, FORTRAN, Haskell, Icon, LISP, Pascal, Prolog, SNOBOL.
9. An extensible language is a language which can be extended after language design time. Compare the extensibility features of C or Pascal with those of LISP or Scheme.
10. What programming language constructs of C are dependent on the local environment?
11. What languages provide for binding of type to a variable at run-time?
12. Discuss the advantages and disadvantages of early and late binding for the following language features. The type of a variable, the size of an array, the forms of expressions and commands.
13. Compare two programming languages from the same computational paradigm with respect to the programming language design principles.
14. Construct a program in your favorite language to do one of the following:
    a. Perform numerical integration where the function is passed as a parameter.
    b. Perform sorting where the the less-than function is passed as a parameter.

Syntax

The syntax of a programming language describes the structure of programs without any consideration of their meaning.

Keywords and phrases: Regular expression, regular grammar, context-free grammar, parse tree, ambiguity, BNF, context sensitivity, attribute grammar, inherited and synthesized attributes, scanner, lexical analysis, parser, static semantics.

Syntax is concerned with the structure of programs and layout with their appearance. The syntactic elements of a programming language are determined by the computation model and pragmatic concerns. There are well developed tools (regular, context-free and attribute grammars) for the description of the syntax of programming languages. Grammars are rewriting rules and may be used for both recognition and generation of programs. Grammars are independent of computational models and are useful for the description of the structure of languages in general.

Context-free grammars are used to describe the bulk of the language's structure; regular expressions are used to describe the lexical units (tokens); attribute grammars are used to describe the context sensitive portions of the language. Attribute grammars are described in a later chapter.

Alphabets and Languages

A language is formally defined by its words and its sentence structure. In this section, we develop the basic notions of words and sentences as strings of words. The collection of words from which sentences are constructed is called an *alphabet* and a *language* is a collection of strings formed from the elements of the alphabet. A string with no words from the alphabet is called the *empty string* and is denoted by **lambda**. Definition 2.1 formalizes these notions.

Definition 2.1: Alphabet and Language

Sigma
: An *alphabet* **Sigma** is a nonempty, finite set of symbols.

L
: A *language* L over an alphabet **Sigma** is a collection of strings of elements of **Sigma**. The *empty string* **lambda** is a string with no symbols at all.

Sigma*
: The set of all possible finite strings of elements of **Sigma** is denoted by **Sigma***. **Lambda** is an element of **Sigma***.

A *string* is a finite sequence of symbols from an alphabet, **Sigma**. The *concatenation* of two strings *v* and *w* is the string *wv* obtained by appending the string *w* to the right of string *v*.

Programming languages require two levels of description, the lowest level is that of a *token*. The tokens of a programming language are the keywords, identifiers, constants and other symbols appearing in the language. In the program

```
void main()
{
  printf("Hello World\n");
}
```

the tokens are

```
void, main, (, ), {, printf, (, "Hello World\n", ), ;, }
```

The alphabet for the language of the lexical tokens is the character set while the alphabet for a programming language is the set of lexical tokens;

A string in a language L is called a *sentence* and is an element of **Sigma***. Thus a language L is a subset of **Sigma***. **Sigma**$^+$ is the set of all possible nonempty strings of **Sigma**, so **Sigma**$^+$ = **Sigma*** - { **lambda** }. A token is a sentence in the language for tokens and a program is a sentence in the language of programs.

If L_0 and L_1 are languages, then L_0L_1 denotes the language {xy | x is in L_0, and y is in L_1 }. That is L_0L_1 consists of all possible concatenations of a string from L_0 followed by a string from L_1.

Grammars and Languages

The ordering of symbols within a token are described by regular expressions. The ordering of symbols within a program are described by *context-free grammars.* In this section, we describe context-free grammars. A later section describes regular expressions. Context-free grammars describe how lexical units (tokens) are grouped into meaningful structures. The alphabet (the set of lexical units) consists of the keywords, identifiers, constants, punctuation symbols, and various operators. While context-free grammars are sufficient to describe most programming language constructs, they cannot specify context-sensitive aspects of a language such as the requirements that a name must be declared before it is referenced, the order and number of actual parameters in a procedure call must match the order and number of formal arguments in a procedure declaration, and that types must be compatible on both sides of an assignment operator.

A grammar consists of a finite collection of grammatical categories (e.g. noun phrase, verb phrase, article, noun, verb etc), individual words (elements of the alphabet), rules for describing the order in

which elements of the grammatical categories must appear and there must be a most general grammatical category. Figure 2.1 contains a context-free grammar for a fragment of English.

Figure 2.1: G_0 a grammar for a fragment of English

The grammatical categories are: S, NP, VP, D, N, V.
The words are: a, the, cat, mouse, ball, boy, girl, ran, bounced, caught.
The grammar rules are:

```
S  --> NP VP
NP --> N
NP --> D N
VP --> V
VP --> V NP
V  --> ran | bounced | caught
D  --> a | the
N  --> cat | mouse | ball | boy | girl
```

The most general category is S, a sentence.

In a context-free grammar, the grammatical categories are called *variables*, the words (tokens) are called *terminals*, the grammar rules are rewriting rules called *productions*, and the most general grammatical category is called the *start symbol*. This terminology is restated in Definition 2.2.

Definition 2.2: Context-free grammar

Context-free grammar G is a quadruple

$$G = (V, T, P, S)$$

where

V is a finite set of variable symbols,
T is a finite set of terminal symbols disjoint from V,
P is a finite set of rewriting rules (productions) of the form

```
A --> w where A in V, w in (V union T)*
```

S is an element of V called the *start* symbol.

Grammars may be used to generate the sentences of a language. Given a string w of the form

$$w = uxv$$

the production x --> y is applicable to this string since x appears in the string. The production allows us to replace x with y obtaining the string z

$$z = uyv$$

and say that w *derives* z. This is written as

$$w \Longrightarrow z$$

If

$$w_1 \Longrightarrow w_2 \Longrightarrow \ldots \Longrightarrow w_n$$

we say that w_1 derives w_n and write

$$w_1 \Longrightarrow^* w_n$$

The set of sentences of a language are derived from the start symbol of the grammar. Definition 2.3 formalizes these ideas.

Definition 2.3: Generation of a Language from the Grammar

Let G be a grammar. Then the set

$$L(G) = \{w \text{ in } T^* \mid S \Longrightarrow^* w\}$$

is the language generated by G.

A language L is context-free iff there is a context-free grammar G such that L = L(G).

If w in L(G), then the sequence

$$S \Longrightarrow w_1 \Longrightarrow w_2 \Longrightarrow \ldots \Longrightarrow w_n \Longrightarrow w$$

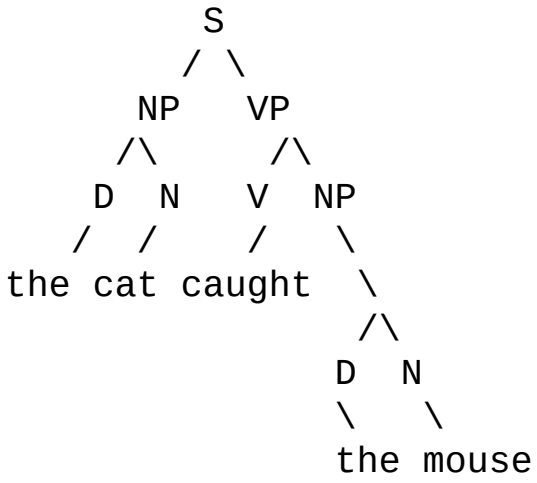
is a *derivation* of the sentence w and the w_i are called *sentential forms.*

Using the grammar G_0 the sentence *the cat caught the mouse* can be generated as follows:

```
S ==> NP VP
  ==> D N VP
  ==> the N VP
  ==> the cat VP
  ==> the cat V NP
  ==> the cat caught NP
  ==> the cat caught D N
  ==> the cat caught the N
  ==> the cat caught the mouse
```

This derivation is performed in a *leftmost* manner. That is, in each step the leftmost variable in the sentential form is replaced.

Sometimes a derivation is more readable if it is displayed in the form of a *derivation tree*.



The notion of a tree based derivation is formalized in Definition 2.5.

Definition 2.5: Derivation Tree

Let G = (V, T, P, S) be a context-free grammar. A *derivation tree* has the following properties.

1. The root is labeled S.
2. Every interior vertex has a label from V.
3. If a vertex has label A in V, and its children are labeled (from left to right) a_1, ..., a_n, then P must contain a production of the form

   A --> $a_1 ... a_n$
4. Every leaf has a label from T union {**lambda**}.

In the generation example we chose to rewrite the left-most nonterminal first. When there are two or more left-most derivations of a string in a given grammar or, equivalently, there are two distinct derivation trees for the same sentence, the grammar is said to be *ambiguous*. In some instances, ambiguity may be eliminated by the selection of another grammar for the language or adding rules which may not be context-free rules. Definition 2.6 defines ambiguity in terms of derivation trees.

Definition 2.6: Ambiguous Grammar

A context-free grammar G is said to be ambiguous if there exists some w in L(G) which has two distinct derivation trees.

Abstract Syntax

Programmers and compiler writers need to know the actual symbols used in programs -- the *concrete syntax*. A grammar defining the concrete syntax of arithmetic expressions is grammar G_1 in Figure 2.2,.

Figure 2.2: G_1 An expression grammar

```
V = { E }
T = { c, id, +, *, (, ) }
P = {E --> c,
     E --> id,
     E --> (E),
     E --> E + E,
     E --> E * E }
S = E
```

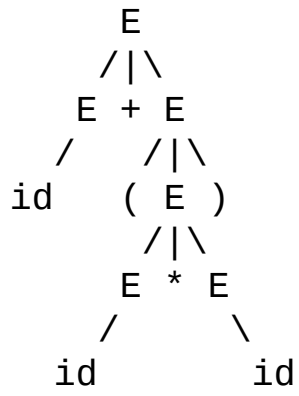
We assume that `c` and `id` stand for any constants and identifiers respectively. Concrete syntax is concerned with the hierarchical relationships and the particular symbols used. The main point of *abstract syntax* is to omit the details of physical representation, specifying the pure structure of the language by specifying the logical relations between parts of the language. A grammar defining the abstract syntax of arithmetic expressions is grammar G_2 in Figure 2.3.

Figure 2.3: G_2 An abstract expression grammar

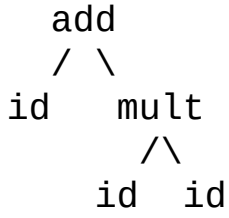
```
V = { E }
T = { c, id, add, mult}
P = {E --> c,
     E --> id,
     E --> add E E ,
     E --> mult E E }
S = E
```

The terminal symbols are names for classes of objects.

An additional difference between concrete and abstract syntax appeThe key difference in the use of concrete and abstract grammars is best illustrated by comparing the derivation tree and the abstract syntax tree for the expression `id + (id * id)`. The derivation tree for the concrete grammar is just what we would expect



while the abstract syntax tree for the abstract grammar is quite different.



In a derivation tree for an abstract grammar, the internal nodes are labeled with the operator and the the operands are their children and there are no concrete symbols in the tree. Abstract syntax trees are used by compilers for an intermediate representation of the program.

Concrete syntax defines the way programs are written while abstract syntax describes the pure structure of a program by specifying the logical relation between parts of the program. Abstract syntax is important when we are interested in understanding the meaning of a program (its semantics) and when translating a program to machine code.

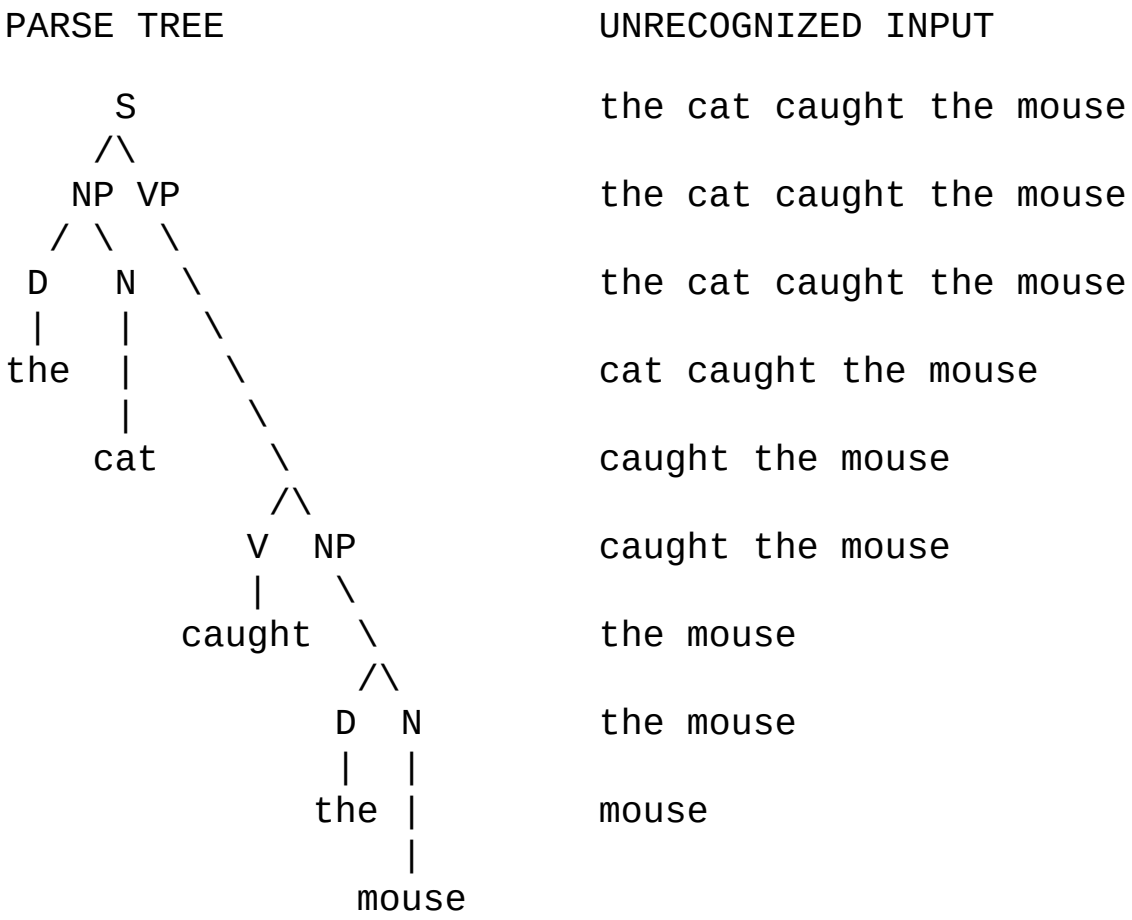
Parsing

Grammars may be used both for the generation and recognition (parsing) of sentences. Both generation and recognition requires finding a rewriting sequence consisting of applications of the rewriting rules which begins with the grammar's start symbol and ends with the sentence. The recognition of a program in terms of the grammar is called *parsing*. An algorithm which recognizes programs is called a *parser*. A parser either implicitly or explicitly builds a derivation tree for the sentence.

There are two approaches to parsing. The parser can begin with the start symbol of the grammar and attempt to generate the same sentence that it is attempting to recognize or it can try to match the input to the right-hand side of the productions building a derivation tree in reverse. The first approach is called *top-down parsing* and the second, *bottom-up* parsing.

Figure 2.4 illustrates top-down parsing by displaying both the parse tree and the remaing unrecognized input. The input is scanned from left to right one token at a time.

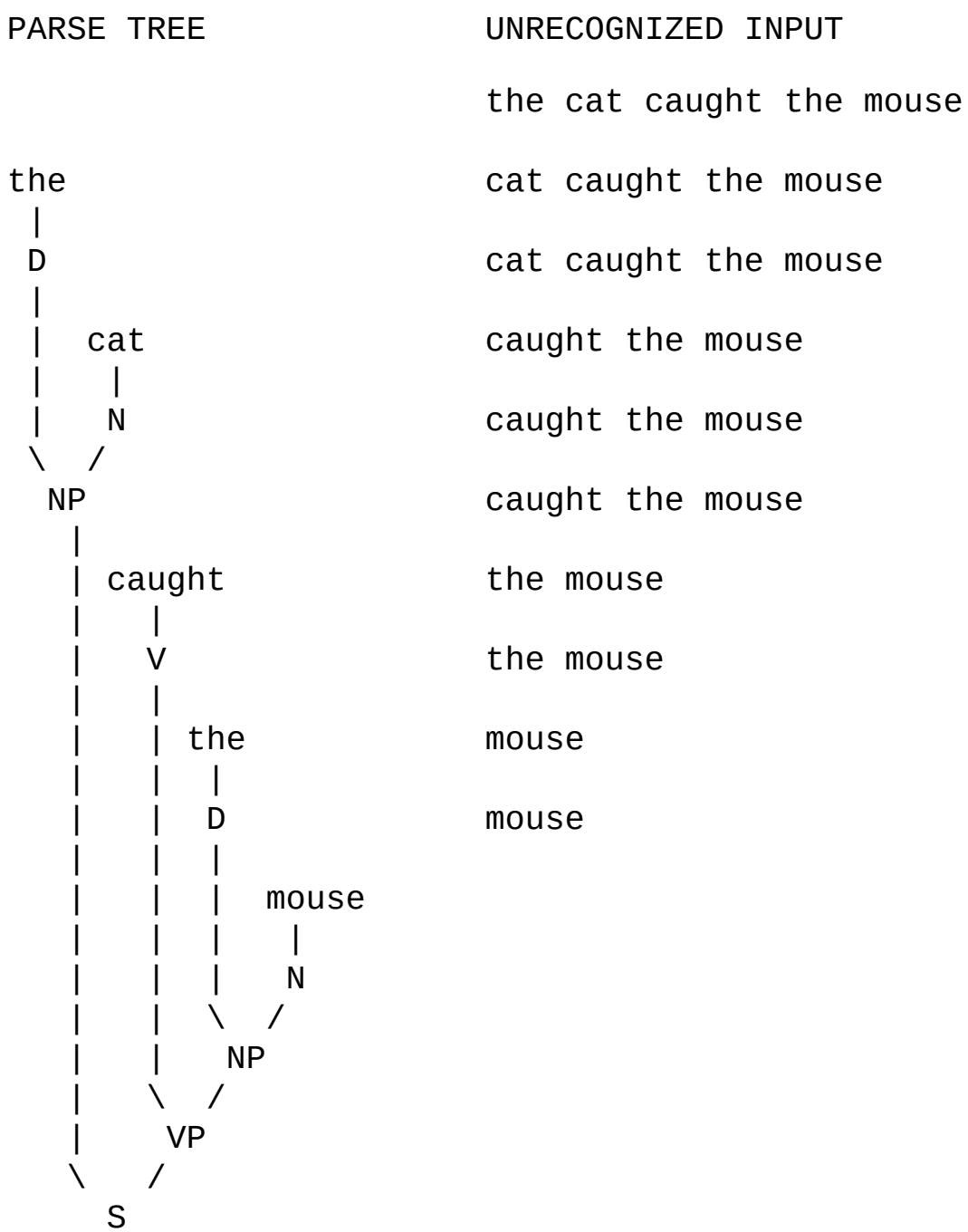
Figure 2.4: Top-down Parse



Each line in the figure represents a single step in the parse. Each nonterminal is replaced by the right-hand side defining it. Each time a terminal matches the input, the corresponding token is removed from the input.

Figure 2.5 illustrates bottom-up parsing by displaying both the parse tree and the remaining unrecognized input. Note that the parse tree is constructed up-side down, i.e., the parse tree is built in reverse.

Figure 2.5: Bottom-up Parse



Each line represents a step in the parsing sequence. The input tokens *shifted* from the input to the parse tree when the parser is unable to *reduce* branches of the tree to a variable.

Table-driven and recursive descent parsing

The simplest and most intuitive approach to constructing a parser is to translate the grammar into a collection of recursive routines. Such a parser is called a *recursive descent parser*. A procedure *parseN* is constructed for each variable *N*. The right-hand side of the productions determine the body of the parse procedure. Variables in the right-hand side become calls to a parse procedure. Terminals in the right-hand side are translated to a verification check to make sure the input corresponds to the terminal and a procedure call to get the next input token. Additional details and restrictions on grammars are given in the Chapter: Translation.

An alternate approach is to construct *top-down table-driven* parser which consists of a driver routine, a stack and the grammar (usually stored in tabular form). The driver routine follows the following algorithm:

1. Initialize the stack with the start symbol of the grammar.
2. Repeat until no further actions are possible
    a. If the top of the stack and the next input symbol are the same, pop the top of the stack and consume the input symbol.
    b. If the top of the stack is a nonterminal symbol, pop the stack and push the right hand side of the corresponding grammar rule onto the stack.
3. If both the stack and input are empty, accept the input otherwise, reject the input.

To illustrate this approach we use the grammar G_1 for expressions and parse the expression `id+id*id`. Figure 2.6 contains a trace of the parse.

Figure 2.6 Top-down parse of id+id*id

```
STACK        INPUT          RULE/ACTION

   E]        id+id*id]     pop & push using E --> E+E
 E+E]        id+id*id]     pop & push using E --> id
id+E]        id+id*id]     pop & consume
  +E]          +id*id]     pop & consume
   E]           id*id]     pop & push using E --> E*E
 E*E]           id*id]     pop & push using E --> id
id*E]           id*id]     pop & consume
  *E]             *id]     pop & consume
   E]              id]     pop & push using E --> id
  id]              id]     pop & consume
    ]                ]     accept
```

The trace shows the contents of the stack and the remaining input at each step of the parse.

A third alternative is to construct a *bottom-up table driven* parser which consists of a driver routine, a stack and a grammar stored in tabular form. The driver routine follows the following algorithm:

1. Initially the stack is empty.
2. Repeat until no further actions are possible.
    a. If the top *n* stack symbols match the right hand side of a grammar rule in reverse, then reduce the stack by replacing the *n* symbols with the left hand symbol of the grammar rule.
    b. If no reduction is possible then shift the current input symbol to the stack.
3. If the input is empty and the stack contains only the start symbol of the grammar, then accept the input otherwise, reject the input.

To illustrate this approach we use the grammar G_1 for expressions and parse the expression `id+id*id`. Figure 2.7 contains a trace of the parse.

Figure 2.7: Bottom-up parse of id+id*id

```
STACK    INPUT         RULE/ACTION

       ]  id+id*id]    Shift
     id]    +id*id]    Reduce using E --> id
      E]    +id*id]    Shift
     +E]     id*id]    Shift
   id+E]       *id]    Reduce using E --> id
    E+E]       *id]    Shift
   *E+E]        id]    Shift
 id*E+E]          ]    Reduce using E --> id
  E*E+E]          ]    Reduce using E --> E*E
    E+E]          ]    Reduce using E --> E+E
      E]          ]    Accept
```

The trace shows the contents of the stack and the remaining input at each step of the parse.

In these examples the choice of the which production to use may appear to be magical. In the case of a top-down parser, grammar G_1 should be rewritten to remove the ambiguity. For bottom up parsers, there are techniques for the analysis of the grammar to produce a set of unambiguous choices for productions. Such techniques are beyond the scope of this text.

Nondeterministic pushdown automata

A careful study of the parsing process reveals that whether the parse is top-down or bottom-up, the parser must hold some information on a stack. In the case of a recursive descent parser, the stack is

implicit in the recursion. In the case of the top-down parser, it must pop variables off the stack and push the corresponding right-hand side on the stack and pop terminals off the stack when they match the input. In the case of the bottom-up parser, it must *shift* (push) terminals onto the stack from the input and *reduce* (pop) sequences of terminals and variables off the stack replacing them with a variable where the sequence of terminals and variables correspond to the right-hand side of some production.

This observation leads us to the notion of *push-down* automata. A push-down automata has an input that it scans from left to right, a stack, and a finite control to control the operations of reading the input and pushing data on and popping data off the stack. Definition 2.6 is a formal definition of a push-down automata.

Definition 2.6: Push-down automaton

A *push-down automaton* M is a 7-tuple (Q, **Sigma**, **Tau**, **delta**, q_0, Z_0, F)

Q	is a finite set of states
Sigma	is a finite alphabet called the input alphabet
Tau	is a finite alphabet called the stack alphabet
delta	is a transition function from Q× (**Sigma** union {e})× **Tau** to finite subsets of Q× **Tau***
q_0	in Q is the initial state
Z_0	in **Tau** is called the start symbol
F	a subset of Q; the set of accepting states

PDA = < States, StartState, FinalStates, InputAlphabet, Input, StackAlphabet, Stack, TransitionFunction, >
Configuration: C = State x Stack x Input; initial configuration (StartState, [], Input)

t : C --> C
Allowed transitions
t(s, [], []) -- accept (empty stack)
t(s, [], S) -- accept s in FinalStates
t(s, I, S) = (s', I, S) -- epsilon move
t(s, [i|I], S) = (s', I, S) -- consume input
t(s, I, [x|S]) = (s', I, S) -- pop stack
t(s, I, S) = (s', I, [x|S]) -- push stack
t(s, [i|I], [x|S]) = (s', I, S) -- consume input and pop stack
t(s, [i|I], S) = (s', I, [x|S]) -- consume input and push stack

Example: palindroms program (StartState, Input, [])

```
t(push,     [],     []) = accept            // empty input
t(push, [x|I],      S) = (pop,  I,      S) // center, odd length palindrom
t(push, [x|I],      S) = (pop,  I, [x|S]) // center, even length palindrom
t(push, [x|I],      S) = (push, I, [x|S]) // left side
t(pop,  [x|I], [x|S]) = (pop,  I,      S) // right side
t(pop,      [],     []) = accept
```

Applications of PDAs are explored in the exercises.

Equivalence of PDA and CFGs

Just as a grammar defines a language L(G), so a PDA M defines a language L(M), the set of strings that it accepts. The relationship between PDAs and CFG is interesting. Any language accepted by a PDA can be shown to be shown to have a context-free grammar. Also any context-free grammar defines a PDA. While the proof of these statements is beyond the scope of this text, the idea of the proof is this. The configurations of a PDA can be described in terms of a context-free grammar. All CFGs can be put into a special form (Greibach normal form) which can be used to describe the configurations of a PDA.

Regular Expressions

While CFGs can be used to describe the tokens of a programming languages, regular expressions (RE) are a more convenient notation for describing their simple structure. The alphabet consists of the character set chosen for the language and the notation includes

- `·' to concatenate items (juxtaposition is used for the same purpose),
- `|' to separate alternatives (often `+' is used for the same purpose),
- `*' to indicate that the previous item may be repeated zero or more times, and
- `(' and `)' for grouping.

Definition 2.7: Regular expressions and Regular languages

Regular Expression E	**Language Denoted L(E)**	
ø	ø	The empty set; language
lambda	{**lambda**}	empty string; language which consists of the empty string

a	{ a }	a; the language which consists of just a
(E ·F)	{uv | u in L(E) and v in L(F) }	concatenation;
(E|F)	{u | u in L(E) or u in L(F) }	alternation; union of L(E) and L(F)
(E^*)	{$u_1u_2...u_n$| u_i in L(E) 0 <= i <=n, n >=0 }	any sequence from E

Identifiers and real numbers may be defined using regular expressions as follows:

```
integer = D D*
identifier = A(A|D)*
```

A scanner is a program which groups the characters of an input stream into a sequence of tokens. Scanners based on regular expressions are easy to write. Lex is a scanner generator often packaged with the UNIX environment. A user creates a file containing regular expressions and Lex creates a program to search for tokens defined by those regular expressions. Text editors use regular expressions to search for and replace text. The UNIX grep command uses regular expressions to search for text.

Deterministic and nondeterministic Finite State Machines

Regular expressions are equivalent to *finite state machines*. A finite state machine consists of a set of states (one of which is a start state and one or more which are accepting states), a set of transitions from one state to another each labeled with an input symbol, and an input string. Each step of the finite state machine consists of comparing the current input symbol with the set of transitions corresponding to the current state and then consuming the input symbol and moving to the state corresponding to the selected transition. Definition 2.8 states this formally.

Definition 2.8: Finite State Automaton

A *finite state automaton* or *fsa* is defined by the quintuple

$$M = (Q, \textbf{Sigma}, \textbf{delta}, q_0, F),$$

where

Q is a finite set of *internal states*
Sigma is a finite set of symbols called the *input alphabet*

delta: Q × **Sigma** --> 2^Q is a total function called the *transition function*
q_0 in Q is the *initial state*
F a subset of Q is the set of *final states*

FSM = <States, StartState, FinalStates, InputAlphabet, Input, TransitionFunction>
Configuration: C = State x Input; inititial configuration (StartState, Input)

t : C --> C
Allowed transitions
t(s, []) -- accept s in FinalStates
t(s, [x|I]) = (s', I) -- consume input
t(s, I) = (s', I) -- epsilon move

Example: identifiers (StartState, Input)

t(start, [i|I]) = (ad, I)
t(ad, [i|I]) = (ad, I)
t(ad, [d|I]) = (ad, I)
t(ad, []) -- accept

The transition function **delta** is defined on a state and an input symbol. It can be extended to a function **delta*** on strings of input symbols as follows:

1. **delta***(q,-)=q for the empty string
2. **delta***(q,wa)=**delta**(**delta***(q,w),a) for all strings w and input symbols a

A FSA is called *deterministic* if there is at most one transition from one state to another for a given input and there are no **lambda** transitions. A FSA is called *nondeterministic* if there is one or more transitions from one state to another for a given input. A Moore machine is a FSA which associates an output with each state and a Mealy machine is a FSA which associates an output with each transition. The Moore and Mealy FSAs are important in applications of FSAs.

Equivalence of deterministic and nondeterministic fsa

It might seem that a machine that could `guess' (nondeterministic) which move to make next would be more powerful than one that could not (deterministic). The following theorems show that in the case of FSAs, it is not the case.

Theorem: Every deterministic finite state automaton is a nondeterministic finite state automaton.
*Proof:*The definition of a deterministic FSA is included in the definition of a nondeterministic FSA.

Theorem: For every nondeterministic finite state automaton there is a deterministic finite state

automaton that accepts the same language.
Proof:

Let M=(S,A,t,q_0,F) be a nondeterministic FSA. Define M'=(S',A,t',F') as follows:

S' is the set of all subsets of S; an element of S' is denoted by [q_1...,q_m]
t': t'([q_1...,q_m],a) = [p_1,...,p_n] where [p_1,...,p_n] is the union of the states of S such that $t(q_i,a) = p_j$} F' is the set of all states of S' that contain an accepting state of M

The proof is completed by induction on the length of the input string.

Equivalence of fsa and regular expressions

Just as context-free grammars and PDAs are equivalent, so regular expressions and FSAs are equivalent as the following theorems show.

Theorem: If r is a regular expression then there exists an NFA that accepts L(r).
Proof (Sketch) The formal proof is by induction on the number of operators in the expression. For the base cases (empty set, empty string and singleton set) the corresponding fsa is simple to define. The more complex cases are handled by merging states of simpler fsa.

Theorem: If L is accepted by a DFA then L is denoted by a regular expression.
Proof beyond the scope of this text.

Graphical Representation

In a graphical representation, states are represented by circles, with final (or accepting) states indicated by two concentric circles. The start state is indicated by the word ``Start''. An arc from state s to state t labeled a indicates a transition from s to t on input a. A label a/b indicates that this transition produces an output b. A label a_1, a_2,..., a_k indicates that the transition is made on any of the inputs a_1, a_2,..., a_k.

```
/* NEED A NICE DIAGRAM HERE */
```

Tabular Representation

In a tabular representation, states are one index and inputs the other. The entries in the table are the next state (and actions if required).

Figure 2.8: Tabular representation for a

transition function.

state/input	I_0	...	I_n
S_0	S_{00}	...	S_{0n}
...	...	...	...
S_m	S_{m0}	...	S_{mn}

Implementation of FSAs

The transition function of a FSA can be implemented as a case statement, a collection of procedures and as a table. In a case based representation state is represented by the value of a variable, the case statement is placed in the body of a loop and on each iteration of the loop, the input is read and the state variable updated.

```
State := Start;
repeat
   get input I
   case State of
        ...
        Si : case I of
                ...
                Ci : State := Sj;
                        ...
             end
        ...
   end
until empty input and accepting state
```

In a procedural representation, each state is a procedure. Transitions to the next state occur when the procedure representing the next state is ``called''.

```
procedure StateS(I : input)
        case I of
                ...
                Ci : get input I; StateT(I)
                ...
        end
```

In the table-driven implementation, the transition function is encoded in a two dimensional array. One index is the current state the other is the current input. The array element are states.

```
state := start;
```

```
while state != final do
        get input I;
        state := table[state,I]
```

The implementations are incomplete since they do not contain code to deal with the end of input.

Pragmatics

At the semantics level, concrete syntax does not matter. However, concrete syntax does matter to the programmer and to the compiler writer. The programmer needs a language that is easy to read and write. The compiler writer wants a language that is easy to parse. Simple details such as placement of keywords, semicolons and case can complicate the life of the programmer or compiler writer.

Many languages are designed to designed to make compilation easy. The goal is to provide a syntax so that the compiler need make only one pass over the program. This requirement means that with minor exceptions, each constant, type, variable, procedure and function must be defined before it is referenced. The trade-off is between slightly increased syntactic complexity of the language with some increased in the burden on the programmer and a simpler compiler.

Some specific syntactical issues include:

- *Statement termination and/or separation.* In Pascal the semicolon is a statement separator while in C the semicolon is a statement terminator. Thus in Pascal a semicolon is not necessary after the last statement in a sequence of statements while it is required in C. If a language includes an empty statement, a misplaced semicolon can change the meaning of a program. For example, in the program fragment

  ```
  while C do; S;
  ```

  the first semicolon terminates the empty statement following the `do` and the while statement; `S` is not in the body of the while statement.

- *Case sensitivity.* Pascal is case insensitive while C is case sensitive.
- *Opening and closing keywords.* Algol-68 and Modula-2 require closing keywords. Modula-2 uses and `end` while Algol-68 uses the reverse of the opening keyword for example,

  ```
  if C then S fi
  ```
- *The assignment operator.* The assignment operator varies among imperative programming languages.

`:=`	Pascal and Ada
`=`	FORTRAN, C/C++/Java
`<--`	APL

  The choice in FORTRAN and C/C++/Java is unfortunate since assignment is different from

equality.

`.EQ.`	FORTRAN
`==`	C/C++/Java

- *Identification of function and procedure calls.* In C, procedure calls are distinguished by the presence of parentheses. Pascal does not require parentheses.
- *Return values.* In C, if a function is used as a command, its return value is ignored. In Pascal, a function cannot be used as a command. To ignore the returned value, Modula-3 requires the function call with the keyword `EVAL`.

A syntax directed editor can use color, font, and layout to assist the programmer in distinguishing between comments, reserved words, code, and can provide command completion.

Historical Perspectives and Further Reading

Backus-Naur Form

The BNF is a notation for describing the productions of a context-free grammar. The BNF uses the following symbols <, >, ::=, |. Variables are enclosed between < and >. The symbol --> is replaced with ::=. The symbol | is used to separate alternatives. Terminals are represented by themselves or are written in a type face different from the symbols of the BNF. The following is a BNF description of arithmetic expressions.

<Expression> ::= <Identifier> | <Number> |
<Expression> <Op> <Expression> |
(<Expression>)
<Op> ::= + | - | * | /
<Identifier> ::= <Letter>
<Identifier> ::= <Identifier> <Letter>
<Number> ::= <Digit>
<Number> ::= <Number> <Digit>
<Letter> ::= A | ... | Z
<Digit> ::= 0 | ... | 9

EBNF (extended BNF)

Several several extensions to improve the readability of the BNF have been suggested. One such extension is to write the names of the variables in italics and without < and >. In addition, the EBNF (extended BNF) is a combination of the BNF and the notation of regular expressions. An EBNF production rule is of the form `N ::= E`, where `N` is a nonterminal symbol and `E` is an extended regular expression. Like ordinary regular expressions, `E` may contain `|', `*', and parentheses for grouping but unlike ordinary regular expressions, it may contain variable symbols as well as terminal symbols.

Some additional extensions include the use of braces, {E}, or ellipses, E..., to indicate zero or more repetitions of an item and brackets, [E], to indicate an optional item.

Figure 2.8 contains a context-free grammar for a simple imperative programming language.

Figure 2.8: Context-free grammar for Simple

```
program ::= LET definitions IN command_sequence END

definitions ::= e | INTEGER id_seq IDENTIFIER .

id_seq ::= e | id_seq IDENTIFIER ,

command_sequence ::= e | command_sequence command ;

command := SKIP
        |  READ IDENTIFIER
        |  WRITE exp
        |  IDENTIFIER := exp
        |  IF exp THEN command_sequence ELSE command_sequence FI
        |  WHILE bool_exp DO command_sequence END

exp ::=  exp + term | exp - term  | term
term :: term * factor | term / factor | factor
factor ::= factor^primary  | primary
primary ::=  NUMBER | IDENT | ( exp )
bool_exp ::=  exp = exp | exp < exp | exp > exp
```

Syntax

Slonneger & Kurts (1995)
Formal Syntax and Semantics of Programming Languages Addison Wesley
Watt, David A. (1991)
Programming Language Syntax and Semantics Prentice-Hall International.

Language Descriptions

It is instructive to read official language descriptions. The following are listed in historical order.

FORTRAN
Backus, J. W. et. al (1956) ``The FORTRAN Automatic Coding System" in *Great Papers in*

Computer Science by Laplante, P. ed. West. 1996.

LISP

McCarthy, J. (1960) *Recursive Functions of Symbolic Expressions* ACM Communications 3 4 April 1960, 184-195.

ALGOL 60

Naur, P., ed (1963) *Revised Report on the Algorithmic Language ALGOL 60* Communications of the ACM. 6, 1-17.

Algol 68

Pascal

Jensen, K. and Wirth, N. (1974) *Pascal User Manual and Report* 2ed. Springer-Verlag

Ada

Reference Manual for the Ada Programming Language U.S. Department of Defense, ANSI/MIL-STD 1815A-1983, Washington, D. C., February, 1983.

C

Kernighan and Ritchie (1978) ``The C Reference Manual'' in *The C Programming Language* Prentice Hall.

C++

Java 1.02

(1996) *The Java Language Specification*

Scheme

Haskell 1.3

Peterson, John., ed (1996)a *The Haskell Report 1.3*

Prolog

Gödel

Parser (Compiler) Construction Tools

Lex & Yacc (or Flex/Bison)
Eli Compiler Construction System
Purdue Compiler-Construction Tool Set (PCCTS)

Watt, David A. (1993)

Programming Language Processors Prentice-Hall International

JACK (Java parser and scanner construction tool)

Formal Languages and Automata

TM = <States, InputAlphabet, TransitionFunction, FinalStates, StartState>
Configuration: C = State x Input; initial configuration (StartState, Input)

t : C --> C
Allowed transitions
t(s, I) -- accept s in FinalStates
t(s, I) = (s', I) -- epsilon move
t(s, (Left, x, [y|Right])) = (s', ([x'|Left], y, Right)) -- move right
t(s, ([x|Left], y, Right)) = (s', (Left, x, [y'|Right])) -- move left

For regular expressions and their relationship to finite automata and context-free grammars and their relationship to push-down automata see texts on formal languages and automata such as\cite{HU79}. The original paper on attribute grammars was by Knuth\cite{Knuth68}. For a more recent source and their use in compiler construction and compiler generators see \cite{DJL88,PittPet92}

Hopcroft and Ullman (1979)
Introduction to Automata Theory, Languages, and Computation Addison-Wesley

Linz, Peter (1996)
An Introduction to Formal Languages and Automata D. C. Heath and Company

Exercises

1. [time/difficulty](cfg) What is the size of L_0L_1?
2. (cfg) Is $L_0L_1 = L_1L_0$?
3. (cfg) Show that the grammar G_1 is ambiguous by producing two distinct derivation trees for the sentence: E + E * E.
4. (cfg) Define a grammar for the if-then and if-then-else control structures. Is your grammar is ambiguous? Hint: try producing two distinct derivation trees for the sentence: if C then if C then S else S.
5. (cfg, bnf, ebnf) Discuss the advantages and disadvantages of the following grammars for the if-then-else statements. Hint: consider the grammars from both the user and parser perspectives.
    a. *stmt* --> begin *stmts* end

       stmt--> if *expr*then *stmt*
       stmt--> if *expr*then *stmt*else *stmt*
    b. *stmt* --> if *expr* then *stmts* endif

       stmt--> if *expr*then *stmts*else *stmts*endif
    c. *stmt* --> if *expr* then *stmts* {elsif *expr* then *stmts*}[else *stmts*] end
6. (cfg) Does the order in which production rules are applied matter? Can they be applied in an arbitrary order including in parallel or in some random order?
7. (cfg) Can a fully abstract grammar be ambiguous?
8. (parse) In a top-down parse, what is required of the grammar so that the parser will be able to pick the correct production?
9. (parse) In a bottom-up parse, ...
10. (parse) Construct a recursive descent parser for G_0, the grammar for a fragment of English (see figure 2.1).
11. (pda) Construct a PDA which checks for matching parentheses.
12. (pda) Construct a PDA which recognizes palindromes.
13. (pda) Construct a PDA which translates arithmetic expressions from infix to post-fix.
14. (pda) Show that a PDA can recognize the language a^nb^n.
15. (pda) Show that a PDA cannot recognize the language $a^nb^nc^n$.
16. (re) Define binary numbers using regular expressions.
17. (re) Define real numbers using regular expressions.
18. (re) Construct a scanner to recognize identifiers, numbers and arithmetic operators.
19. (re, parse) Using the following grammar for expressions:

exp::= *term exp*'
exp' ::= + *term exp*' | - *term exp*' | **epsilon**
term::= *factor term*'
term' ::= * *factor term*' | / *factor term*' | **epsilon**
factor::= *primary factor*'
factor' ::= ^ *primary factor*' | **epsilon**
primary::= INT | IDENT | (*exp*)

a. Construct a trace of a top down parse for the expression `id+id*id.`
b. Construct a scanner and a recursive descent parser for the grammar.

20. (re, parse) Construct a scanner and a parser for the programming language Simple
21. (pragmatics) Discuss the advantages and disadvantages of Pascal or C style function calls (C requires empty parentheses for parameterless functions while Pascal does not).
22. (pragmatics) Discuss the advantages and disadvantages of case sensitivity for the programmer and compiler writer.
23. (pragmatics) Discuss the consequences of the number of reserved words in a programming language.
24. (pragmatics) Discuss the necessity separators and terminators.
25. (pragmatics) Discuss the advantages and disadvantages of requiring declarations before references for the compiler writer and for the programmer.

Semantics

The semantics of a programming language describe the relationship between the syntax and the model of computation.

Keywords and phrases: Algebraic semantics, axiomatic semantics, denotational semantics, operational semantics, semantic algebra, semantic axiom, semantic domain, semantic equation, semantic function, loop variant, loop invariant, valuation function, sort, signature, many-sorted algebra

---

Semantics is concerned with the interpretation or understanding of programs and how to predict the outcome of program execution. The semantics of a programming language describe the relation between the syntax and the model of computation. Semantics can be thought of as a function which maps syntactical constructs to the computational model.

semantics: syntax --> computational model

This approach is called *syntax-directed semantics*.

There are several widely used techniques (algebraic, axiomatic, denotational, operational, and translation) for the description of the semantics of programming languages.

- *Algebraic semantics* describe the meaning of a program by defining an algebra. The algebraic relationships and operations are described by axioms and equations.
- *Axiomatic semantics* defines the meaning of the program implicitly. It makes assertions about relationships that hold at each point in the execution of the program. Axioms define the *properties* of the control structures and state the properties that may be infered. A property about a program is deduced by using the axioms. Each program has a pre-condition which describes the initial conditions required by the program prior to execution and a post-condition which describes, upon termination of the program, the desired program property.
- *Denotational semantics* tell what is computed by giving a mathematical object (typically a function) which is the meaning of the program. Denotational semantics are used in comparitive studies of programming langauges.
- *Operational semantics* tell how a computation is performed by defining how to simulate the execution of the program. Operational semantics may describe the syntactic transformations which mimic the execution of the program on an abstract machine or define a translation of the program into recursive functions. Operational semantics are used when learning a programming language and by compiler writers.
- *Translation semantics* describe how to translate a program into an other langauge usually the language of a machine. Translation semantics are used in compilers.

Much of the work in the semantics of programming languages is motivated by the problems encountered in trying to construct and understand imperative programs---programs with assignment commands. Since the assignment command reassigns values to variables, the assignment can have unexpected effects in distant portions of the program.

Algebraic Semantics

An *algebraic definition* of a language is a definition of an algebra. An algebra consists of a domain of values and a set of operations (functions) defined on the domain.

Algebra = < set of values; operations >

Figure N.1 contains an example of an algebraic definition. It is an algebraic definition of a fragment of Peano arithmetic.

Figure N.1: Algebraic Definition of Peano Arithmetic

Domains:

Bool = {true, false} (Boolean values)
N in Nat (the natural numbers)
N ::= 0 | S(N)

Functions:

= : (Nat, Nat) -> Bool
+ : (Nat, Nat) -> Nat
× : (Nat, Nat) -> Nat

Axioms and equations:

not S(N) = 0
if S(M) = S(N) then M = N
(n + 0) = n
(m + S(n)) = S(m + n)
(n × 0) = 0
(m × S(n)) = ((m × n) + m)

where m,n in Nat

The equations define equivalences between syntactic elements; they specify the transformations that are used to translate from one syntactic form to another. The domain is often called a *sort* and the domain and the function sections constitute the *signature* of the algebra. Functions with zero, one, and two operands are referred to as nullary, unary, and binary operations. Because there is more than one domain, the algebra is called a *many sorted algebra*. As in this example, abstract data types may require values from several different sorts. The signature of the algebra is a set of sorts and a set of functions taking arguments and returning values of different sorts. A stack of natural numbers may be modeled as a many-sorted algebra with three sorts (natural numbers, stacks and booleans) and four operations (newStack, push, pop, top, empty). Figure N.2 contains an algebraic definition of a stack.

Figure N.2: Algebraic definition of an Integer Stack ADT

Domains:

Nat (the natural numbers
Stack (of natural numbers)
Bool (boolean values)

Functions:

newStack: () -> Stack
push : (Nat, Stack) -> Stack
pop: Stack -> Stack
top: Stack -> Nat
empty : Stack -> Bool

Axioms: or Defining Equations:

pop(push(N,S)) = S
top(push(N,S)) = N
empty(push(N,S)) = false
empty(newStack()) = true

newStack() = []
push(N,S) = [N|S]
pop([N|S]) = S
top([N|S]) = N

Errors:

pop(newStack())
top(newStack()) where N in Nat and S in Stack.

In Figure N.1, the structure of the numbers is described. In Figure N.2 the structure of a stack is not defined. This means that we cannot use equations to describe syntactic transformations. Instead, we use axioms that describe the relationships between the operations. The axioms are more abstract than equations because the results of the operations are not described. To be more specific would require decisions to be made concerning the implementation of the stack data structure. Decisions which would tend to obscure the algebraic properties of stacks. The axioms impose constraints on the stack operations that are *sound* in the sense that they are consistent with the actual behavior of stacks reguardless of the implementation. Finding axioms that are *complete*, in the sense that they completely specify the behavior of the operations of an ADT, is more difficult.

The goal of algebraic semantics is to capture the semantics of behavior by a set of axioms with purely syntactic properties. Algebraic definitions (semantic algebras) are the favored method for defining the properties of abstract data types.

Axiomatic Semantics

The *axiomatic semantics* of a programming language are the assertions about relationships that remain the same each time the program executes. Axiomatic semantics are defined for each control structure and command. The axiomatic semantics of a programming language define a *mathematical theory* of programs written in the language. A mathematical theory has three components.

- **Syntactic rules:** These determine the structure of formulas which are the statements of interest.
- **Axioms:** These describe the basic properties of the system.
- **Inference rules:** These are the mechanisms for deducing new theorems from axioms and other theorems.

The semantic formulas are triples of the form:

{P} **c** {Q}

where **c** is a command or control structure in the programming language, P and Q are *assertions* or statements concerning the properties of program objects (often program variables) which may be true or false. P is called a *pre-condition* and Q is called a *post-condition*. The pre- and post-conditions are formulas in some arbitrary logic and summarize the progress of the computation.

The meaning of

{P} **c** {Q}

is that if **c** is executed in a state in which assertion P is satisfied and **c** terminates, then **c** terminates in a state in which assertion Q is satisfied. We illustrate axiomatic semantics with a program to compute the sum of the elements of an array (see Figure N.3).

Figure N.3: Program to compute S = $sum_{i=1}^{n}A[i]$

```
S,I := 0,0
while I < n do
   S,I := S+A[I+1],I+1
end
```

The assignment statements are *simultaneous* assignment statements. The expressions on the righthand side are evaluated simultaneously and assigned to the variables on the lefthand side in the order they appear.

Figure N.4 illustrates the use of axiomatic semantics to verify the program of Figure N.3.

Figure N.4: Verification of S = $sum_{i=1}^{n}A[i]$

	Pre/Post-conditions	**Code**
1.	{ 0 = $Sum_{i=1}^{0}$A[i], 0 < |A| = n }	
2.		S,I := 0,0
3.	{S = $Sum_{i=1}^{I}$A[i], I <= n }	
4.		while I < n do
5.	{S = $Sum_{i=1}^{I}$A[i], I < n }	
6.	{S+A[I+1] = $Sum_{i=1}^{I+1}$A[i], I+1 <= n }	
7.		S,I := S+A[I+1],I+1
8.	{ S = $Sum_{i=1}^{I}$A[i], I <= n }	
9.		end
10.	{S = $Sum_{i=1}^{I}$A[i], I <= n, I >= n }	
11.	{S = $Sum_{i=1}^{n}$A[i] }	

The program sums the values stored in an array and the program is decorated with the assertions which help to verify the correctness of the code. The pre-condition in line 1 and the post-condition in line 11 are the pre- and post-conditions respectively for the program. The pre-condition asserts that the array contains at least one element zero and that the sum of the first zero elements of an array is zero. The post-condition asserts that S is sum of the values stored in the array. After the first assignment we know that the partial sum is the sum of the first I elements of the array and that I is less than or equal to the number of elements in the array.

The only way into the body of the while command is if the number of elements summed is less than the number of elements in the array. When this is the case, The sum of the first I+1 elements of the array is equal to the sum of the first I elements plus the $I+1^{st}$ element and I+1 is less than or equal to n. After the assignment in the body of the loop, the loop entry assertion holds once more. Upon termination of the loop, the loop index is equal to n. To show that the program is correct, we must show that the assertions satisfy some verification scheme. To verify the assignment commands, we use the *Assignment Axiom*:

Assignment Axiom

$$\{P[x:E]\}\ \texttt{x := E}\ \{P\}$$

This axiom asserts that:

> If after the execution of the assignment command the environment satisfies the condition P, then the environment prior to the execution of the assignment command also satisfies the condition P but with E substituted for x (In this and the following axioms we assume that the evaluation of expressions does not produce side effects.).

An examination of the respective pre- and post-conditions for the asssignment statements shows that the axiom is satisfied.

To verify the while command of lines 4. 7 and 9, we use the *Loop Axiom*:

Loop Axiom:

$$\frac{\{I \wedge \texttt{B} \wedge V > 0\}\ \texttt{C}\ \{I \wedge V > V' >= 0\}}{\{I\}\ \texttt{while B do C end}\ \{I \wedge \neg\, \texttt{B}\}}$$

The assertion above the bar is the condition that must be met before the axiom (below the bar) can hold. In this rule, {I} is called the *loop invariant*. This axiom asserts that:

> To verify a loop, there must be a loop invariant I which is part of both the pre- and post-conditions of the body of the loop and the conditional expression of the loop must be true to execute the body of the loop and false upon exit from the loop.

The invariant for the loop is: $S = sum_{i=1}^{I} A[i]$, I <= n. Lines 6, 7, and 8 satisfy the condition for the application of the Loop Axiom. To prove termination requires the existence of a *loop variant.* The loop variant is an expression whose value is a natural number and whose value is decreased on each iteration of the loop. The loop variant provides an upper bound on the number of iterations of the loop.

> A variant for a loop is a natural number valued expression V whose run-time values satisfy the following two conditions:
>
> - The value of V greater than zero prior to each execution of the body of the loop.
> - The execution of the body of the loop decreases the value of V by at least one.

The loop variant for this example is the expression n - I. That it is non-negative is guaranteed by the loop continuation condition and its value is decreased by one in the assignment command found on line 7. More general loop variants may be used; loop variants may be expressions in any well-founded set (every decreasing sequence is finite). However, there is no loss in generality in requiring the variant expression to be an integer. Recursion is handled much like loops in that there must be an invariant and a variant. The correctness requirement for loops is stated in the following:

> **Loop Correctness Principle:** Each loop must have both an invariant and a variant.

Lines 5 and 6 and lines 10 and 11 are justified by the *Rule of Consequence.*

Rule of Consequence:

$$\frac{P \rightarrow Q,\ \{Q\}\ C\ \{R\},\ R \rightarrow S}{\{P\}\ C\ \{S\}}$$

The justification for the composition the assignment command in line 2 and the while command requires the following the *Sequential Composition Axiom.*

Sequential Composition Axiom:

$$\frac{\{P\}\ C0\ \{Q\},\ \{Q\}\ C1\ \{R\}}{\{P\}\ C0;\ C1\ \{R\}}$$

This axiom is read as follows:

> The sequential composition of two commands is permitted when the post-condition of the first command is the pre-condition of the second command.

The following rules are required to complete the deductive system.

Selection Axiom:

$$\frac{\{P \wedge B\}\ C0\ \{Q\},\ \{P \wedge \neg B\ \}\ C1\ \{Q\}}{\{P\}\ \text{if B then C0 else C1 fi}\ \{Q\}}$$

Conjunction Axiom:

$$\frac{\{P\}\ C\ \{Q\},\ \{P'\}\ C\ \{Q'\}}{\{P \wedge P'\ \}\ C\ \{Q \wedge Q'\}}$$

Disjunction Axiom:

$$\frac{\{P\}\ C\ \{Q\},\ \{P'\}\ C\ \{Q'\}}{\{P \vee P'\ \}\ C\ \{Q \vee Q'\}}$$

The axiomatic method is the most abstract of the semantic methods and yet, from the programmer's point of view, the most practical method. It is most abstract in that it does not try to determine the meaning of a program, but only what may be proved about the program. This makes it the most practical since the programmer is concerned with things like, whether the program will terminate and what kind of values will be computed.

Axiomatics semantics are appropiate for program verification and program derivation.

Assertions for program construction

The axiomatic techniques may be applied to the construction of software. Rather than proving the correctness of an existing program, the proof is integrated with the program construction process to insure correctness from the start. As the program and proof are developed together, the assertions themselves may provide suggestions which facilitate program construction.

Loops and recursion are two constructs that require invention on the part of the programmer. The loop correctness principle requires the programmer to come up with both a variant and an invariant. Recursion is a generalization of loops so proofs of correctness for recursive programs also require a loop variant and a loop invariant. In the summation example, a loop variant is readily appearent from an examination of the post-condition. Simply replace the summation upper limit, which is a constant, with a variable. Initializing the sum and index to zero establishes the invariant. Once the invariant is established, either the index is equal to the upper limit in which case there sum has been computed or the next value must be added to the sum and the index incremented reestablishing the loop invariant. The position of the loop invariants define a loop body and the second occurrence suggests a recursive call. A recursive version of the summation program is given in Figure N.5.

Figure N.5: Recursive version of summation

```
S,I := 0,0
loop: if I < n then S,I := S+A[I+1],I+1; loop
      else skip fi
```

The advantage of using recursion is that the loop variant and invariant may be developed separately. First develop the

invariant then the variant.

The summation program is developed from the post-condition by replacing a constant by a variable. The initialization assigns some trivial value to the variable to establish the invariant and each iteration of the loop moves the variable's value closer to the constant.

A program to perform integer division by repeated subtraction can be developed from the post-condition { 0 <= r < d, (a = q× d + r) } by deleting a conjunct. In this case the invariant is { 0 <= r, (a = q× d + r) } and is established by setting the the quotient to zero and the remainder to a.

Another technique is called for in the construction of programs with multiple loops. For example, the post condition of a sorting program might be specified as:

{ forall i.(0 < i < n -> A[i] <= A[i+1]), s = perm(A)}

or the post condition of an array search routine might be specifies as:

{ if exists i.(0 < i <= n and t = A[i]) then location = i else location = 0}

To develop an invariant in these cases requires that the assertion be strengthened by adding additional constraints. The additional constraints make assertions about different parts of the array.

Denotational Semantics

A *denotational definition* of a language consists of three parts: the abstract syntax of the language, a semantic algebra defining a computational model, and valuation functions. The valuation functions map the syntactic constructs of the language to the semantic algebra. Recursion and iteration are defined using the notion of a limit. the programming language constructs are in the *syntactic domain* while the mathematical entity is in the *semantic domain* and the mapping between the various domains is provided by valuation functions. Denotational semantics relies on defining an object in terms of its constituent parts. The Figure N.6 is an example of a denotational definition.

Figure N.6: Denotational definition of Peano Arithmetic

Abstract Syntax:

N in Nat (the Natural Numbers)
N ::= 0 | S(N) | (N + N) | (N × N)

Semantic Algebra:

Nat (the natural numbers (***0***, ***1***, ...)
+ : ***Nat*** -> ***Nat*** -> ***Nat***

Valuation Function:

D : Nat -> ***Nat***

D[(n + 0)] = ***D***[n]
D[(m + S(n))] = ***D***[(m+n)] **+** ***1***
D[(n × 0)] = ***0***

$$\boldsymbol{D}[(\ m \times S(n))] = \boldsymbol{D}[\ ((\ m \times n) + m)\ ]$$

where m,n in Nat

It is is a denotational definition of a fragment of Peano arithmetic. Notice the subtle distinction between the syntactic and semantic domains. The syntactic expressions are mapped into an algebra of the natural numbers by the valuation function. The denotational definition almost seems to be unnecessary. Since the syntax so closely resembles that of the semantic algebra.

Programming languages are not as close to their computational model. Figure N.7 is a denotational definition of the small imperative programming language Simple encountered in the previous chapter.

Figure N.7: Denotational semantics for Simple

```
Abstract Syntax:

C in Command
E in Expression
O in Operator
N in Numeral
V in Variable

C ::= V := E | if E then C1 else C2 end |
      while E do C3 end | C1;C2 | skip
E ::= V | N | E1 O  E2  | (E)
O ::= + | - | * | / | = | < | > | <>

Semantic Algebra:
Domains:

tau in  T = {true, false}; the  boolean values
zeta in  Z = {...-1,0,1,...}; the integers
+ : Z -> Z -> Z
...
= : Z -> Z -> T
...
sigma in  S = Variable -> Numeral; the state

Valuation Functions:

C in C -> (S -> S)
E in E -> E -> (N union T)

C[ skip ] sigma = sigma
C[ V := E ] sigma = sigma [ V:E[ E ] sigma
C[ C1; C2 ] = C[ C2 ] · C[ C1]
C[ if E then C1 else C2 end ] sigma
          = C[ C_1 ] sigma  if E[ E ]sigma = true
          = C[ C_2 ] sigma  if E[ E ]sigma = false
C[ while E do C end}]sigma
          = lim_{n -> infty} C[ (if E then C else skip end)^n ] sigma
```

```
E[ V ] sigma = sigma(V)
E[ N ] = zeta
E[ E1+E2 ] = E[ E ] sigma + E[ E ] sigma
...
E[ E1=E2 ] sigma = E[ E ] sigma = E[ E ] sigma
```

Denotational definitions are favored for theoretical and comparative programming language studies. Denotational definitions have been used for the automatic construction of compilers for the programming language. Denotations other than mathematical objects are possible. For example, a compiler writer would prefer that the object denoted would be appropriate object code. Systems have been developed for the automatic construction of compilers from the denotation specification of a programming language.

Operational Semantics

An *operational definition* of a language consists of two parts: an abstract syntax and an interpreter. An interpreter defines how to perform a computation. When the interpreter evaluates a program, it generates a sequence of machine configurations that define the program's operational semantics. The interpreter is an evaluation relation that is defined by rewriting rules. The interpreter may be an abstract machine or recursive functions. Figure N.8 is an example of an operational definition.

Figure N.8: Operational semantics for Peano arithmetic

```
Abstract Syntax:

N in Nat (the natural numbers)

N ::= 0 | S(N) | (N + N) | (N × N)

Interpreter:

I: N -> N

I[ ( n + 0 ) ]     ==> n
I[ ( m + S(n) ) ]  ==> S( I[ (m+n ) ] )
I[ ( n × 0 ) ]     ==> 0
I[ ( m × S(n)) ]   ==> I[ (( m × n) + m) ]


where m,n in Nat
```

It is is an operational definition of a fragment of Peano arithmetic.

The interpreter is used to rewrite natural number expressions to a standard form (a form involving only S and 0) and the rewriting rules show how move the + and × operators inward toward the base cases. Operational definitions are favored by language implementors for the construction of compilers and by language tutorials because operational definitions describe how the actions take place.

The operational semantics of Simp is found in Figure N.9.

Figure N.9: Operational semantics for Simple

```
Interpreter:

I: C × Sigma -> Sigma
{nu} in E × Sigma} -> T union Z

Semantic Equations:

I(skip,sigma) = sigma
I(V := E,sigma) = sigma[V:nu(E,sigma)]
I(C1 ;C2,sigma) = E(C2,E(C1,sigma))
I(if E then C1 else C2 end,sigma) = I(C1,sigma)&if nu(E,sigma) = true}
                                                  I(C2,sigma)&if nu(E,sigma) =
false}
while E do C end = if E then (C;while E do C end) else skip
nu(V,sigma) = sigma(V)
nu(N,sigma) = N
nu(E1+E2,sigma) = nu(E1,sigma) + nu(E2,sigma)
...
nu(E1=E2,sigma) = true if nu(E,sigma) = nu(E,sigma)}
             false if nu(E,sigma) != nu(E,sigma)}
otherwise
...
```

The operational semantics are defined by using two semantic functions, *I* which interprets commands and **nu** which evaluates expressions. The interpreter is more complex since there is an environment associated with the program with does not appear as a syntactic element and the environment is the result of the computation. The environment (variously called the *store* or *referencing* environment}) is an association between variables and the values to which they are assigned. Initially the environment is empty since no variable has been assigned to a value. During program execution each assignment updates the environment. The interpreter has an auxiliary function which is used to evaluate expressions. The **while** command is given a recursive definition but may be defined using the interpreter instead. Operational semantics are particularly useful in constructing an implementation of a programming language.

Pragmatics

The use of formal semantic description techniques is playing an increasing role in software engineering. Algebraic semantics are useful for the specification of abstract data types. However, the lack of robust theorem provers has limited the effective use axiomatic semantics for program varification. Denotational semantics are beginning to play a role in compiler construction and a prescriptive rather than a descriptive role in the design of programming languages. Operational semantics have always proved helpful in the design of compilers.

Historical Perspectives and Further Reading

An excellent short introduction to operational, denotational and axiomatic semantics is found in

Schmidt, David A.
Programming Language Semantics. *ACM Computing Surveys*, Vol. 28, No. 1, March 1996.

Other references include:

- General texts: algebraic, axiomatic, denotational, and operational semantics
  Slonneger & Kurts (1995)
  Formal Syntax and Semantics of Programming Languages Addison Wesley
  Meyer, Bertrand Meyer. (1990)
  Introduction to the Theory of Programming Languages Prentice-Hall International.
  Watt, David A. (1991)
  Programming Language Syntax and Semantics Prentice-Hall International.
- Axiomatic semantics
  Gries, David (1981)
  The Science of Programming Springer-Verlag.
  Hehner, E. C. R. (1984)
  The Logic of Programming Prentice-Hall International.
  Hehner, E. C. R. (1993)
  A Practical Theory of Programming Springer-Verlag.
- Denotational semantics
  Schmidt, D. A. (1988)
  Denotational Semantics -- A methodology for Language Development Wm. C. Brown Publishers Dubuque, Iowa
  Stoy, J. (1977)
  Denotational Semantics -- the Scott-Strachey approach to programming language theory, MIT Press, Cambridge, Massachusetts, United States.

Exercises

1. (axiomatic) Give axiomatic semantics for the following:
   a. Multiple assignment command: $x_0,...,x_n := e_0,...,e_n$
   b. The following commands are a nondeterministic if and a nondeterministic loop. The IF command allows for a choice between alternatives while the DO command provides for iteration. In their simplest forms, an IF statement corresponds to an If condition then command and a LOOP statement corresponds to a While condition Do command.

      IF *guard* --> *command* FI = if *guard* then *command*

      LOOP *guard* --> *command* POOL = while *guard* do *command*

      A command proceded by a guard can only be executed if the guard is true. In the general case, the semantics of the IF - FI and LOOP - POOL commands requires that only one command corresponding to a guard that is true be selected for execution. The selection is nondeterministic.. Define the axiomatic semantics for the IF and LOOP commands:
      i. if c_0 -> s_0
         ...
         c_n -> s_n
         fi
      ii. do c_0 -> s_0
         ...
         c_n -> s_n
         od
   c. A for statement
   d. A repeat-until statement
2. (axiomatic) Use assertions to guide the construction of the following programs.
   a. Linear search
   b. Integer division implemented by repeated subtraction.
   c. Factorial function
   d. F_n the *n*-th Fibonacci number where $F_0 = 0$, $F_1 = 1$, and $F_{i+2} = F_{i+1} + F_i$ for i >= 0.
   e. Binary search

   f. Quick sort
3. (algebraic) Construct algebraic semantics for the following:
   a. Stack
   b. List
   c. Queue
   d. Binary search tree
   e. Graph
   f. Grade book
   g. Complex numbers
   h. Rational numbers
   i. Floating point numbers
   j. Simple
4. (denotational) Construct denotational semantics for the following:
   a. Stack
   b. List
   c. Queue
   d. Binary search tree
   e. Graph
   f. Grade book
   g. Complex numbers
   h. Rational numbers
   i. Floating point numbers
   j. Simple
   k. Show that the following code denotes the same function.

```
int f (int n)
{ if n > 1 then n*f(n-1)
           else 1
}

int f (int n)
{ int t = 1;
  while n > 1 do {
    t := t*n;
    n := n-1}
}
```

5. (operational) Construct operational semantics for the following:
   a. Stack
   b. List
   c. Queue
   d. Binary search tree
   e. Graph
   f. Grade book
   g. Complex numbers
   h. Rational numbers
   i. Floating point numbers
   j. Simple
6. (correctness) Construct an implementation of the following and show that your implementation is correct by showing that it satisfies a semantics.
   a. Stack
   b. List
   c. Queue
   d. Binary search tree
   e. Graph
   f. Grade book

g. Complex numbers
h. Rational numbers
i. Floating point numbers
j. Simple

Pragmatics

The pragmatics of a programming language includes issues such as ease of implementation, efficiency in application, and programming methodology. -- Slonneger & Kurtz

Keywords and phrases: strict, non-strict, eager evaluation, lazy evaluation, normal-order evaluation, binding time, passing by value, passing by reference, passing by name, passing by value, passing by result, passing by value-result, aliasing

Heaps and Pointers

Treating procedure and function abstractions as first-class values is another potential cause of dangling references. (Watt)

A *heap variable* is one that can be created and deleted at any time. Heap variables are anonymous and are accessed through *pointers*. A *heap* is a block of storage within which pieces are allocated and freed in some relatively unstructured mannner.

Initially the elements of the heap are linked together in some fashion to form a *free-space list*. The creation of a heap variable is requested by an operation called an *allocator* which returns a pointer to the newly created variable. To allocate an element, the first element on the list is removed from the list and a pointer to it is returned to the operation requesting the storage.

The heap variable's lifetime extends from the time it is created until it is no longer accessable.

Often there is an operation called a *deallocator* which forcibly deletes a given heap variable. When an element is deallocated (freed), it is simply linked back in at the head of the free-space list. If all references to heap variable are destroyed, the heap variable is inaccessable and becomes *garbage*. When a variable becomes garbage, its memory space is unusable by other variables since a means of referencing it must exist in order to return the space to the free-space list.

When deallocation is under the control of the programmer, it is a potential source of problems. If a programmer deallocates a variable, any remaining pointers to the deleted heap variable become *dangling references*.

Garbage and dangling references are potentially troublesome for the programmer. If garbage accumulates, available storage is gradually reduced until the program may be unable to continue for lack of known free space (this is also called a memory leak). If a program attempts to modify through a dangling reference a structure that has been deallocated (destroyed), the contents of an element of

free-space may be modified. This may cause the remainder of the free-space to become garbage or a portion of the program to become linked to free-space. The deallocated space could be reallocated to some other structure resulting in similar problems.

The problem of dangling references can be eliminated.

One solution is to restrict assignment so that references to local variables may not be assigned to variables with a longer lifetime. This restriction may require runtime checks and sometimes restrict the programmer.

Another solution is to maintain reference counts with each heap variable. An integer called the *reference count* is associated with each heap element. The reference count indicates the number of pointers to the element that exist. Initially the count is set to 1. Each time a pointer to the element is created the reference count is increased and each time a pointer to the element is destroyed the reference count is decreased. Its space is not deallocated until the reference count reaches zero. The method of reference counting results in substantial overhead in time and space.

Another solution is to provide *garbage collection*. The basic idea is to allow garbage to be generated in order to avoid dangling references. When the free-space list is exhausted and more storage is needed, computation is suspended and a special procedure called a *garbage collector* is started which identifies garbage and returns it to the free-space list.

There are two stages to garbage collection a *marking phase* and a *collecting phase*.

- Marking phase: The marking phase begins outside the heap with the pointers that point to active heap elements. The chains of pointers are followed and each heap element in the chain is marked to indicate that it is active. When this phase is finished only active heap elements are marked as active.
- Collecting phase: During the collecting phase the heap is scanned and each element which is not active is returned to the free-space list and the marked bits are reset to prepare for a later garbage collection.

This unuseable space may be reclamed by a *garbage collector*. A heap variable is alive as long as any reference to it exists.

Coroutines

Coroutines are used in discrete simulation languages and, for some problems, provide a control structure that is more natural than the usual hierarchy of subprogram calls.

Coroutines may be thought of as subprograms which are not required to terminate before returning to the calling routine. At a later point the calling program may ``resume'' execution of the coroutine at the point from which execution was suspended. Coroutines then appear as equals with control passing from one to the other as necessary. From two coroutines it is natural to extend this to a set of coroutines.

From the description given of coroutines, it is apparent that coroutines should not be recursive. This permits us to use just one activation record for each coroutine and the address of each activation record can be statically maintained.

Each activation record is extended to include a location to store the *CI* for the corresponding coroutine. It is initialized with the location of the first instruction of the coroutine. When coroutine encounters a resume operation, it stores the address of its next instruction in it own activation record. The address of the *CI* for the resumed coroutine is obtained from the activation record of the resumed coroutine.

Safety

The purpose of declarations is two fold. The requirement that all names be declared is essential to provide a check on spelling. It is not unusual for a programmer to mispell a name. When declarations are not required, there is no way to determine if a name is new or if it is a misspelling of a privious name.

The second purpose of declarations is assist the type checking algorithm. The type checker can determine if the intended type of a variable matches the use of the variable. This sort of type checking can be performed at compile time permitting the generation of more efficient code since run time type checks need not be performed.

type checking--static, dynamic

import/export

Declarations and strong type checking facilitate safety by providing redundancy. When the programmar has to specify the type of every entity, and may declare only one entity with a given identifier within a given scope; the compiler then simply checks each the usage of each entity against rigid type rules. With overloading or type inference, the compiler must deduce information not supplied by the programmer. This is error prone since slight errors may radically affect what the compiler does.

Overloading and type inference lack redundancy.

Historical Perspectives and Further Reading

Wadler, Philip (1996)
Lazy Versus Strict. *ACM Computing Surveys*, *28*, 2, (June 1996), 318-320.

Exercises

1. [time/difficulty](section) Problem statement.
2. (compiler) Implement a virtual machine which provides ....

Abstraction and Generalization

Abstraction *is an emphasis on the idea, qualities and properties rather than the particulars (a suppression of detail).*

Generalization *is a broadening of application to encompass a larger domain of objects of the same or different type.*

A ***parameter*** *is a quantity whose value varies with the circumstances of its application.*

Substitution*---To put something in the place of another.*

Encapsulate*---To completely enclose.*

Keywords and phrases: Name, binding, abstract, definition, declaration, variables, parameters, arguments, formals, actuals. Modularity, encapsulation, function, procedure, abstract type, generic, library, object, class, inheritance, partition, package, unit, separate compilation, linking, import, export, instance, scope. block, garbage collection, static and dynamic links, display, static and dynamic binding, activation record, environment, Static and Dynamic Scope, aliasing, variables, value, result, value-result, reference, name, unification, eager evaluation, lazy evaluation, strict, non-strict, Church-Rosser, overloading, polymorphism, monomorphism, coercion, transfer functions.

The ability to abstract and to generalize is an essential part of any intellectual activity. Abstraction and generalization are fundamental to mathematics and philosophy and are essential in computer science as well.

The importance of abstraction is derived from its ability to hide irrelevant details and from the use of names to reference objects. Programming languages provide abstraction through procedures, functions, and modules which permit the programmer to distinguish between *what* a program does and *how* it is implemented. The primary concern of the user of a program is with *what* it does. This is in contrast with the writer of the program whose primary concern is with *how* it is implemented. Abstraction is essential in the construction of programs. It places the emphasis on what an object is or does rather than how it is represented or how it works. Thus, it is the primary means of managing complexity in large programs.

Of no less importance is generalization. While abstraction reduces complexity by hiding irrelevant detail, generalization reduces complexity by replacing multiple entities which perform similar functions with a single construct. Programming languages provide generalization through variables, parameterization, generics and polymorphism. Generalization is essential in the construction of programs. It places the emphasis on the similarities between objects. Thus, it helps to manage

complexity by collecting individuals into groups and providing a representative which can be used to specify any individual of the group.

Abstraction and generalization are often used together. Abstracts are generalized through parameterization to provide greater utility. In parameterization, one or more parts of an entity are replaced with a name which is new to the entity. The name is used as a parameter. When the parameterized abstract is invoked, it is invoked with a binding of the parameter to an argument. Figure N.1 summarizes the notation which will be used for abstraction and generalization.

Figure N.1: Abstraction and Generalization

Abstraction	name : abstract		
Invocation	name		
Substitution	E[p:a]	*(a replaces p in E)*	
Generalization	**lambda** p.E		
Specialization	(**lambda** p.E a) = E[p:a]		
Abstraction and generalization	name : **lambda** p.E	name(p) : E	name p : E
Invocation and specialization	(name a)	name(a)	

When an abstraction is fully parameterized (all free variables bound to parameters) the abstraction may be understood without looking beyond the abstraction.

Abstraction and generalization depend on the principle of referential transparency.

> **Principle of Referential Transparency** The meaning of an entity is unchanged when a part of the entity is replaced with an equal part.

Abstraction

> **Principle of Abstraction** An *abstract* is a named entity which may be invoked by mentioning the name.

Giving an object a name gives permission to substitute the name for the thing named (or vice versa) without changing the meaning. We use the notation

name : *abstract*

to denote the *binding* of a name to an abstract. Declarations and definitions are all instances of the use of abstraction in programming languages

In addition to naming there is a second aspect to abstraction. It is that the abstract is encapsulated, that

is, the details of the abstract are hidden so that the name is sufficient to represent the entity. This aspect of abstraction is considered in more detail in a later chapter.

An object is said to be *fully abstract* if it can be understood without reference to any thing external to the object.

> **Terminology.** The naming aspect of abstraction is captured in the concepts of *binding*, *definition* and *declaration* while the hiding of irrelevant details is captured by the concept of encapsulation. A binding is an association of two entities. A definition is a binding of a name to an entity, a declaration is a definition which binds a name to a variable, and an assignment is a binding of a value and a variable.
>
> We could equally well say *identifier* instead of name. A *variable* is an entity whose value is not fixed but may vary. Names are bound to variables in declaration statements. Among the various terms for abstracts found in other texts are *module*, *package*, *library*, *unit*, *subprogram*, *subroutine*, *routine*, *function*, *procedure*, *abstract type*, *object*.

Binding

The concept of binding is common to all programming languages. The objects which may be bound to names are called the bindables of the language. The bindables may include: primitive values, compound values, references to variables, types, and executable abstractions. While binding occurs in definitions and declarations, it also occurs at the virtual and hardware machine levels between values and storage locations.

> **Aside.** The imperative programming paradigm is characterized by permitting names to be bound successively to different objects, this is accomplished by the assignment statement (often of the form; name := object) which means ``let name stand for object until further notice.'' In other words, until it is reassigned. This is in contrast with functional and logic programming paradigms in which names may not be reassigned. Thus languages in these paradigms are often called single assignment languages.

Typically the text of a program contains a number of bindings between names and objects and the bindings may be composed collaterally, sequentially or recursively.

A **collateral binding** is to perform the bindings independently of each other and then to combine the bindings to produce the completed set of bindings. Nether binding can reference a name used in any other binding. Collateral bindings are not very common but occur in Scheme and ML.

The most common way of composing bindings is sequentially. A **sequential binding** is to perform the bindings in the sequence in which they occur. The effect is to allow later bindings to use bindings produced earlier in the sequence. It must be noted that sequential bindings do not permit mutually recursive definitions.

In C/C++ and Pascal, constant, variable, and procedure and function bindings are sequential. To

provide for mutually recursive definitions of functions and procedures, C/C++ and Pascal provide for the separation of the signature of a function or procedure from the body by the means of function prototypes & forward declarations so that so that mutually recursive definitions may be constructed.

A **recursive binding** is one in which the name being bound is used (directly or indirectly) in its own binding.

Programming languages that require "declaration before reference" have to invent special mechanisms to handle forward references. For dynamic data types, the rule is relaxed to permit the definition of pointer types. For functions and procedures, there are separate declarations for the signature of the function or procedure and its body. Pascal with its "forward" declarations and C++ with its function prototypes are typical.

The "declaration before reference" is often chosen to simplify the construction of the compiler. In Modula-3 and Java the choice has been made to simplify the programmer's task rather than the compiler's and permit forward references.

Encapsulation

The abstract part of a binding often contains other bindings which are said to be local definitions. Such local definitions are not visible or available to be referenced outside of the abstract. Thus the abstract part of a binding involves ``information hiding''. This hidden information is sometimes made available by exporting the names.

A module system provides a way of writing large program so that the various pieces of the program don't interfere with on another because of name clashes and also provides a way of hiding implementation details. ... A module generally consists of two parts, the export part and the local part. The export part of a module consists of language declarations for the symbols available for use in either part of the module and in other modules which import them and module declaration giving the symbols from other modules which are available for use in either part of the module and in other modules which import them. The local part of a module consists of language declarations for the symbols avaliable for use only in this part. *TGPL-Hill and Lloyd*

The work of constructing large programs is divided among several people, each of whom must produce a part of the whole. Each part is called a module and each programmer must be able to construct his/her module without knowing the internal details of the other parts. This is only possible when each module is is separated into an interface part and an implementation part. The interface part describes all the information required to use the module while the implementation part describes the implementation. This idea is already present in most programming languages in the manner in which functions and procedures are defined. Function and procedure definitions usually are separated into two parts. The first part gives the subprogram's name and parameter requirements and the second part describes the implementation. A module is a generalization of the concept of abstraction in that a module is permitted to contain a collection of definitions. An additional goal of modules is to confine changes to a few modules rather than throughout the program.

While the concept of modules is a useful abstraction, the full advantages of modules are gained only when modules may be written, compiled and possibly executed separately. In many cases modules should be able to be tested independently of other modules.

EXPANDTHIS!!! Advantages

% marcotty

- reduction in complexity
- team programming
- maintainability
- reusability of code
- project management

Implementation

% marcotty

- common storage area -- Fortran
- include directive -- C++
- subroutine library

Typical applications:

- subroutine packages -- mathematical, statistical etc
- ADTs

examples from Ada, C++, etc

Generalization

> **Principle of Generalization** A *generic* is an entity which may be specialized (elaborated) upon invocation.

Generalization permits the use of a single pattern to represent each member of a group. We use the notation: **lambda** *p.B′*

(called a lambda abstraction) to denote the generalization of *B* where *p* is called a *parameter* and *B′* is *B* with *p* replacing any number of occurrences of some part of *B* by *p*. The parameter *p* is said to be *bound* in the expression but *free* in *B′* and the *scope* of *p* is said to be *B′*.

The symbol **lambda** is a *quantifier*. Quantifiers are used to replace constants with variables.

The specialization (elaboration) of a generic is called application and takes the form:

(**lambda** *p.B a)*

It denotes the expression B' obtained from the lambda expression when the free occurences of *p* in *B* are replaced by *a.*

> **Aside.** The symbol **lambda** was introduced by Church for variable introduction in the lambda calculus. It roughly corresponds to the symbol forall, the universal quantifier, of first-order logic. The appendix contains a brief introduction to first-order logic. The functional programming chapter contains a brief introduction to the lambda calculus.

Generalization is often combined with abstraction and takes the following form:

```
n( p ) : B
```

where *p* is the name, *x* is the parameter, and *B* is the abstract. The invocation of the abstract takes the form:

```
n(a)
```

or occaisionally `(n a)` where `n` is the name and `a` is called the *argument* whose value is substituted for the parameter. Upon invocation of the abstract, the argument is bound to the parameter. Figure N.1 summarizes the variety of notation that is used to denote the elaboration of a generalization.

Most programming languages permit an implicit form of generalization in which variables may be introduced without providing for an invocation procedure which replaces the parameter with an argument. For example, consider the following psudocode for a program which computes the circumference of a circle:

```
pi : 3.14

c : 2*pi*r

begin
   r := 5
   write c
   r := 20
   write c
end
```

The value of `r` depends on the context in which the function is defined. The variable `r` is a global name and is said to be *free*. In the first write command, the circumference is computed for a circle of radius 5 while in the second write command the circumference is computed for a circle of radius 20. The write commands cannot be understood without reference to both the definition of `c` and to the environment (`pi` is viewed as a constant). Therefore, this program is not ``fully abstract''. In contrast, the following program is fully abstract:

```
pi : 3.14

c(r) : 2*pi*r

begin
   FirstRadius := 5
   write c(FirstRadius)
   SecondRadius := 20
   write c(SecondRadius)
end
```

The principle of generalization depends on the analogy principle.

> **Analogy Principle** When there is a conformation in pattern between two different objects, the objects may be replaced with a single object parameterized to permit the reconstruction of the original objects.

It is the analogy principle which permits the introduction of a variable to represent an arbitrary element of a class.

The Principle of Generalization makes no restrictions on parameters or the parts of an entity that may be parameterized. Neither should programming languages. This is emphasized in the following principle:

> **Principle of Parameterization** A parameter of a generic may be from any domain.
>
> **Terminology.** The terms *formal parameters* (*formals*) and *actual parameters* (*actuals*) are sometimes used instead of the terms parameters and arguments respectively.

Substitution

The utility of both abstraction and generalization depend on substitution. The tie between the two is captured in the following principle:

> **Principle of Correspondence** Parameter binding mechanisms and definition mechanisms are equivalent.

The Principle of Correspondence is a formalization of that aspect of the Principle of Abstraction that implies that definition and substitution are intimately related.

We use the notation

E[p:a]

to denote the substitution of *a* for *p* in *E*. The notation is read as ``E[p:a] is the expression obtained

from E by replacing all free occurrences of p with a" .

> **Terminology.** The notation for substitution was chosen to emphasize the relationship between abstraction and substitution. Other texts use the notation E[p:=a] for substitution. Their notation is motivated by the assignment operation which assigns the value a to p. Other texts use the notation E[a/p] for substitution. This latter notation is motivated by the cancelation that occurs when a number is multiplied by its inverse (p(a/p) = a).

Together, abstraction, invokation, generalization and specialization provide powerful mechanisms for program development. Generalization provides a mechanism for the construction of common subexpressions and abstraction a mechanism for the factoring out of the common subexpressions. In the following example, the factors are first generalized to contain common subexpressions and then abstracted out. The product

(a+b-c)*(x+y-z)

is formed from two very similar factors. The factors generalize to a common expression

lambda i j k. i+j-k.

The lambda expression can use to rewrite the product as:

(**lambda** i j k. i+j-k) a b c * (**lambda** i j k. i+j-k) x y z.

The lambda expression can be abstracted to a name with three arguments,

f(i j k) : i+j-k,

which can be used to replace the lambda expressions with the name and we get the expression

f(a b c) * f(x y z) where f(i j k) : i+j-k

which clearly indicates the similarity of the the factors.

- Partitions
- Separate compilation
  - Linking
  - Name and Type consistency
- Scope rules
  - Import
  - Export
- Modules--collection of objects--definitions
- Package

Block structure

A *block* is a construct that delimits the scope of any definitions that it may contain. It provides a local environment i.e., a opportunity for local definitions. The *block structure* (the textual relationship between blocks) of a programming language has a great deal of influence over program structure and modularity. There are three basic block structures--monolithic, flat and nested. In the discussion that follows, we will refer to the block structures found in Figure N.2.

Figure M.N: Block Syntax

`let` *Definitions* `in` *Body* `end`

Body `where` *Definitions*

Figure N.2 presents two styles of blocks, the first requires the definitions to proceed the body and the second requires definitions to follow the body.

A program has a *monolithic* block structure if it consists of just one block. This structure is typical of BASIC and early versions of COBOL. The monolithic structure is suitable only for small programs. The scope of every definition is the entire program. Typically all definitions are grouped in one place even if they are used in different parts of the program.

Figure M.N: **Monolithic Block Structure**

Global Data

Return Address$_1$

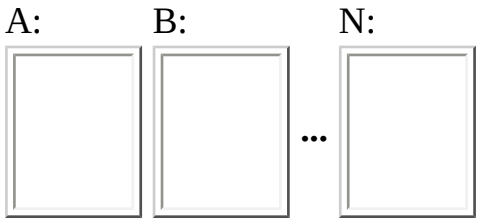
...

Return Address$_n$

A program has a *flat* block structure if it is partitioned into distinct blocks, an outer all inclosing block one or more inner blocks i.e., the body may contain additional blocks but the inner blocks may not contain blocks. This structure is typical of FORTRAN and C. In these languages, all subprograms (procedures and functions) are separate, and each acts as a block. Variables can be declared inside a subprogram are then local to that subprogram. Subprogram names are part of the outer block and thus their scope is the entire program along with global variables. All subprogram names and global

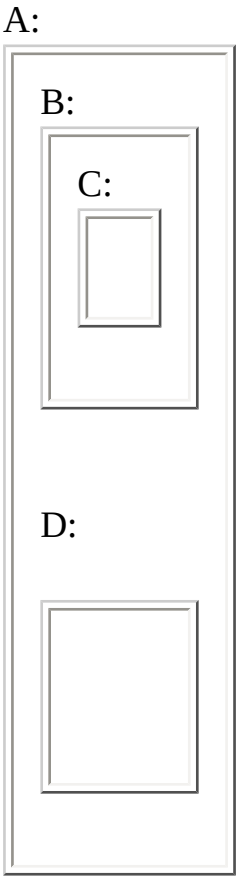
variables must be unique. If a variable cannot be local to a subprogram then it must be global and accessable by all subprograms even though it is used in only a couple of subprograms.

Figure M.N: Flat Block Structure



A program has *nested* block structure if blocks may be nested inside other blocks i.e., there is no restriction on the nesting of blocks within the body. This is typical of the block structure of the Algol-like languages. A block can be located close to the point of use. In blocks visibility is controlled by nesting. All names are visible (implicitly exported) to internally nested blocks. No names are visible (exported) to enclosing blocks. In a block, the only names visible are those that are declared in all enclosing blocks or are declared in the block, but not those declared in nested blocks.

Figure M.N: Nested blocks



A:

- Can see all names in A including the names A and B.
- Cannot see names in B, C, or D.

B:

- Can see all names in A and B including the names A, B, and C.
- Cannot see names in C or D.

C:

- Can see all names in A, B, and C including the names A, B, and C.
- Cannot see names in D..

D:

- Can see all names in A and D including the names A, B, and D.
- Cannot see names in B or C.

A *local name* is one that is declared within a block for use only within that block.

A *global name* is a name that when referenced within a block refers to a name declared outside the block.

An *activation* of a block is a time interval during which that block is being executed.

The three basic block structures are sufficient for what is called *programming in the small* (PITS). These are programs which are comprehensible in their entirety by an individual programmer. However, they are not general enough for very large programs. Large programs which are written by many individuals and which must consist of modules that can be developed and tested independently of other modules. Such programming is called *programming in the large* (PITL*)*.

Activation Records

Each block Storage for local variables.

Scope Rules

The act of partitioning a program raises the issue of the scope of names. Which objects with in the partition are to be visible outside the partition? The usual solution is to designate some names to be *exported* and others to be *private* or local to the partition and invisible to other partitions. In case there might be name conflict between exported names from partitions, partitions are often permitted to designate names that are to be *imported* from designated partitions or to qualify the name with the partition name.

The scope rules for modules define relationships among the names within the partitions. There are four choices.

- All local names visible globally.
- All external names visible locally.
- Only local explicitly exported names visible globally.
- Only external names explicitly imported are visible locally.

Name conflict is resolved via qualification with the partition name.

Dynamic scope rules

A dynamic scope rule defines the dynamic scope of each association in terms of the dynamic course of program execution. Lisp.

implementation ease, cheap generalization for parameterless functions.

Static scope rules

Teminology. Static scope rules are also called *lexical scope rules*.

Cobol, BASIC, FORTRAN, Prolog, Lambda calculus, Scheme, Miranda, Algol-60, Pascal

Environment

An environment is a set of bindings.

Scope has to do with the range of visibility of names. For example, a national boundary may encapsulate a natural language. However, some words used within the boundary are not native words. They are words borrowed from some other language and are defined in that foreign language. So it is in a program. A definition introduces a name and a boundary (the object). The object may contain names for which there is no local definition (assuming definitions may be nested). These names are said to be free. The meaning assigned to these names is to be found outside of the definition. The rules followed in determining the meaning of these free names are called scope rules.

Scope It is concerned with name control.

ADTs

An even more effective approach is to separate the signatures of the operations from the bodies of the operations and the type representation so that the operation bodies and type representation can be compiled separately. This facilitates the development of software in that when an abstract data type's representation is changed (e.g. to improve performance) the changes are localized to the abstract data type.

```
name : adt
operation signatures
...

name : adt body
type representation definition
operation bodies
...
```

Pragmatics

Bindings and Binding Times

Bindings may occur at various times from the point of language definition through program execution. The time at which the binding occurs is termed the *binding time*. Four distinct binding times may be distinguished.

1. *Language design time.* Much of the structure of a programming language is fixed and language design time. Data types, data structures, command and expression forms, and program structure are examples of language features that are fixed at language design time. Most programming languages make provision for extending the language by providing for programmer defined data types, expressions and commands.
2. *Language implementation time.* Some language features are determined by the implementation. Programs that run on one computer may not run or give incorrect results when run on another machine. This occurs when the hardware differs in its representation of numbers and arithmetic operations. For example, the *maxint* of Pascal is determined by the implementation. The C programming language provides access to the underlying machine and therefore programs which depend on the characteristics of the underlying machine may not perform as expected when moved to another machine.
3. *Program translation time.* The binding between the source code and the object code occurs at program translation time. Programmer defined variables and types are another example of bindings that occur at program translation time.
4. *Program execution time.* Binding of values to variables and formal parameters to actual parameters occur during program execution.

Early binding often permits more efficient execution of programs though translation time type checking while late binding permits more flexibility through program modification a run-time. The implementation of recursion may require allocation of memory at run-time in contrast a one time to allocation of memory at compile-time. An Example from Pratt 1984:

X := X + 10

1. Set of possible data types for X (Language design time: Fortran; Translation time: C, Pascal (user defined))
2. Type of variable X (Translation time: C; Execution time: Lisp, APL)
3. Set of possible values for X (Language implementation time (often constrained by hardware))
4. Value of the variable X (Execution time (assignment))
5. Representation of the constant 10 (language definition time (base 10); language implementation time (base 2)).
6. Properties of the operater + (language definition time - addition operations; translation time - type of addition; execution-time - APL)

Procedures and Functions

THISSHOULDBEGENERALIZETOINCLUDEOTHERABSTRACTIONS!
In the discussion which follows, the term *subprogram* will be used to refer to whole programs, procedures and functions.

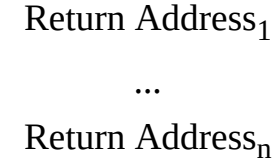
A program may be composed of a main program which during execution may call subprograms which in turn may call other subprograms and so on. When a subprogram is called, the calling subprogram waits for the called subprogram to terminate. Each subprogram is expected to eventually terminate and return control to the calling subprogram. The execution of the calling subprogram resumes at the point immediately following the point of call. Each subprogram may have its own local data which is found in an *activation record*. An activation record consists of an association between variables and the value to which they are assigned. An activation record may be created each time a subprogram is called and destroyed when the subprogram terminates.

DYNAMIC VS. STATIC ALLOCATION

The run time environment must keep track of the current instruction and the referencing environment for each active or waiting program so that when a subprogram terminates, the proper instruction and data environment may be selected for the calling subprogram.

The current instruction of the calling subprogram is maintained on a stack. When a subprogram is called, the address of the instruction following the call of the calling program is pushed on the stack. When a subprogram terminates, the instruction pointer is set to the address on the top of the stack and the address popped off the stack. The stack is often called the *return address stack*.

Figure M.N: **Return Address Stack**



The addresses of the current environment is also maintained on a stack. The top of the stack always points to the current environment. When a subprogram is called, the address of the new environment is pushed on the stack. When a subprogram terminates, the stack is popped revealing the previous environment. The stack is often called the *dynamic links* because the stack contains links (pointers) which reveal the dynamic history of the program.

When a programming language does not permit recursive procedures and data structure size is independent of computed or input data, the maximum storage requirements of the program can be determined at compile time. This simplifies the run time support required by the program and it is possible to statically allocate the storage used during program execution.

Parameters and Arguments

Strict, Non-strict, Eager and Lazy

An generic is said to be *strict* in a parameter if it is sure to need the value of the parameter and *non-strict* in a parameter if it may not require the value of the parameter. Arithmetic operators are strict

$$A + B$$

because their arguments must be evaluated to determine the value of the arithmetic expression but the conditional expression

$$\text{if } B \text{ then } E_1 \text{ else } E_2$$

is not strict in its second and third arguments since the selection of the second or third argument is dependent on the value of the boolean condition (the first argument).

Most programming languages assume that abstracts are strict in their parameters and, therefore, the parameters are evaluated when the function is called. This evaluation scheme is called *eager evaluation*. This is not always desirable and so some languages provide a mechanism for the programmer to inform a function not to evaluate its parameters. Scheme provides for the quote operator to prevent the evaluation of an argument. Logic languages like Prolog and functional languages languages like Haskell and Miranda are non-strict languages and the arguments are evaluated only when the value is required. This evaluation scheme is called *normal-order evaluation* and is often implemented using *lazy evaluation* (the argument is evaluated only when it is first needed).

Most languages use strict evaluation because it is more efficient and simplifies the implementation of parameter passing for imperative programming languages. Normal-order evaluation coupled with side-effects found in imperative langauges produces unexpected results. Algol-60's provides a parameter passing mechanism (pass by name) which is based on that does not provide the generality that is required in the imperative model as the following example shows.

Figure M.N: Algol-60, Jensen's device

```
procedure swap(x,y:sometype);
var t:sometype
begin
   t := x; x := y; y := t
end;
...
I := 1
a[I] := 3
swap(I,a[I])
```

Based on the code in the body of the procedure, it would seem that the values of the arguments would be swaped. That this is not the case is easily seen when the formal parameters are textually replaced

with the actual parameters and the resulting code is executed in the context of the actual parameters. In this case, prior to the call to sort, `I` is 1 and `A[i]` is 3. Upon textual substitution, we have

```
...
I := 1
{I = 1}
a[I] := 3
{I=1, a[I] = a[1] = 3}
t := I; I := a[I]; a[I] := t -- replaces call to swap
{T=1, I=3, a[i] = a[3] = 1, a[1] = 3}
```

After execution, `I` is 3 and `a[1]` is still 3, but `a[3]` is now 1.

Argument Passing Mechanisms

In the previous chapter (Abstraction and Generalization), it appears that when an argument is passed to an abstract, it replaces the parameter, that is, it textually replaces the parameter. If the argument is large, the space and time requirements can be a significant overhead. Especially since the each time the argument is referenced, it must be evaluated not in the internal (local) environment of the abstract but in the environment external to (global) the abstract. This need not be the case and several mechanisms have been developed to make passing arguments simpler and more efficient.

The **copy mechanism** requires values to be copied into an generic when it is entered and copied out of the generic when the generic is exited. This form of parameter passing is often referred to as *passing by value*. The formal parameters are local variables and the argument is copied into the local variable on entry to the generic and copied out of the local variable to the argument on exit from the generic.

The *value parameter* of Pascal and the *in parameter* of Ada are examples of parameters which may be passed by using the copy mechanism. The value of the argument is copied into the parameter on entry but the value of the formal parameter is not copied to the actual parameter on exit. In imperative languages, copying is unnecessary if the language prohibits assignment to the formal parameter. In such a case, the parameter may be passed by reference.

Ada's *out parameter* and function results are examples of parameters which may be passed by using the copy mechanism. The value of the argument (actual parameter) is not copied into the formal parameter on entry but the value of the parameter is copied into the argument upon exit. In Pascal the function name is used as the parameter and assignments may be made to the function name. This form of parameter passing is often referred to as *passing by result*.

When the passing by value and result are combined, the passing mechanism is referred to as *passing by value-result*. Ada's *in out parameter* is an example of a parameter which may be passed by this form of the copy mechanism. The value of the actual parameter is copied into the formal parameter on entry and the value of the formal parameter is copied into the actual parameter upon exit.

The copy mechanism has some disadvantages. The copying of large composite values (arrays etc) is

expensive and the parameters must be assignable (e.g. expressions and file types in Pascal are not assignable).

The effect of a **definitional mechanism** is as if the abstact were surrounded by a block, in which there is a definition that binds the parameter to the argument.

Parameter : Argument

An parameter is said to be passed by *reference* if the argument is an address. References to the parameter are references to the argument. Assignments to the parmeter are assignments to the argument. The *reference parameter* of Pascal and the array and structure parameters of C++ are passed using this mechanism. A toster provides an illustration of the effect of passing by reference.

If an argument --- A parameter is said to be passed by *name* if, in effect, the argument replaces parameter throughout the body of the subroutine (textual substitution with suitable renaming of local variables to avoid conflicts between local variables and variables occuring in the argument) i.e., in the subprogram, each reference to the parameter results in an evaluation of the argument in the calling environment.

In addition to the problems of the pass-by-name mechanism of Algol-60, imperative languages with with reference parameters present the possibility of *aliasing*. Aliasing occurs when two or more names reference the same object. For example, the following procedure and call,

```
procedure confuse (var m, n : Integer );
   begin
      n := 1; n := m + n
   end;
...
i := 5;
confuse(i,i)
```

both m and n are bound to the same variable, i, and i is initially 5, then after the call to the procedure, the value of i is 2 not 6. m and n are both bound to i and after the assignment n := 1, the value of m is also 1.

Scope and Blocks

A variables declared within a block have a *lifetime* which extends from the moment an activation record is created for the block until the activation record for the block is destroyed. A variable is bound to an offset within the activation record at compile time. It is bound to a specific storage location when the block is activated and becomes a part of storage.

STATIC/DYNAMIC ALLOCATION

Data Access

block structure, COMMON, ADT's, aliasing

Scope Rules

Conceptually the **dynamic scope rules** may be implemented as follows. Each variable is assigned a stack to hold the current values of the variable.

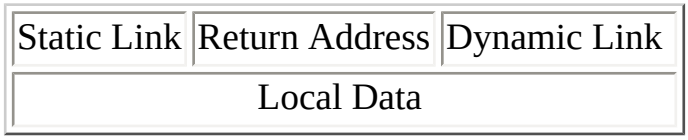
When a subprogram is called, a new uninitialized stack element is pushed on the stack corresponding to each variable in the block.

A reference to a variable involves the inspection or updating of the top element of the appropriate stack. This provides access to the variable in closest block with respect to the dynamic calling sequence.

When a subprogram terminates, the stacks corresponding to the variables of the block are popped, restoring the calling environment. The **static scope rules** may be implemented as follows. The data section of each procedure is associated with an *activation record*. The activation records are dynamically allocated space on a *runtime* stack. Each recursive call is associated with it own activation record. Associated with each activation record is a *dynamic link* which points to the previous activation records, a *return address* which is the address of the instruction to be executed upon return from the procedure and a *static link* which provides access to the referencing environment.

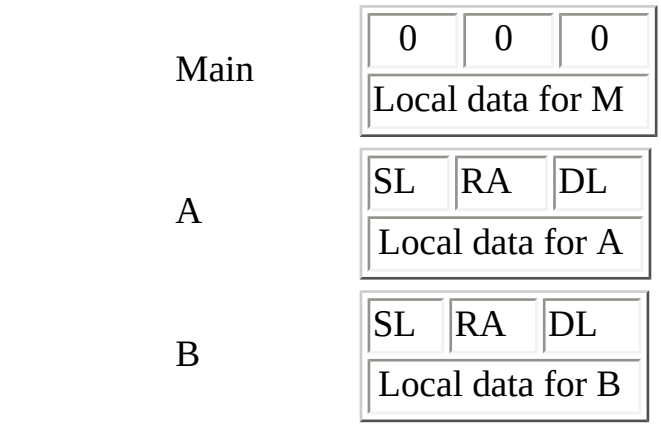
An activation record consists of storage for local variables, the static and dynamic links and the return address.

F**igure M.N:** Activation Record



The runtime stack of activation records (local data, static and dynamic links).

Figure M.N: Runtime Stack



Main calls A which calls B.

Global data values are found by following the static chain to the appropriate activation record.

An alternative method for the implementation of static scope rules is the display. A *display* is a set of registers (in hardware or software) which contain pointers to the current environment. On procedure call, the current display is pushed onto the runtime stack and a new display is constructed containing the revised environment. On procedure exit, the display is restored from the copy on the stack.

Figure M.N: **Display & Runtime Stack**

Partitions

A *partition* of a set is a collection of disjoint sets whose union is the set.

There are a number of mechanisms for partitioning program text. Functions and procedures are among the most common. However, the result is still a single file. When the partitions of program text are arranged in separate files, the partitions are called modules. Here are several program partitioning mechanisms.

- Separate declaration of data and code
- Procedures
- Functions
- ADTs
- Modules

Partitioning of program text is desirable to provide for separate compilation and for pipeline processing of data.

There are a number of mechanisms for combining the partitions into a single program for the purposes of compilation and execution. The `include` statement is provided in a number of languages. It is a compiler directive with directs the compiler to textually include the named file in the source program. In some systems the partitions may be separately compiled and there is a linking phase in which the compiled program modules are linked together for execution. In other systems, at run-time any missing function or procedure results in a run-time search for the missing module which if found is then executed or if not found results in a run-time error.

Modules

A *module* is a program unit which is an (more or less) independent entity. A module consists of a number of definitions (of types, variables, functions, procedures and so on), with a clearly defined interface stating what it exports to other modules which use it. Modules have a number of advantages for the construction of large programs.

- Modules facilitate parallel and independant development
- Modules facilitate separate compilation
- Modules facilitate code reuse

Modules are used to construct libraries, ADTs, classes, interfaces, and implementations. A module is the compilation unit. A module which contains only type abstractions is a specification or interface module.

In program construction the module designer must answer the following questions.

- *What* is the module's purpose?
- *How* does it achieve that purpose?

Programming in the large is concerned with programs that are not comprehensible by a single individual and are developed by teams of programmers. At this level programs must consist of modules that can be written, compiled, and tested independently of other modules. A module has a single purpose, and has a narrow interface to other modules. It is likely to be reuseable (able to be incorporated into may programs) and modifiable with out forcing changes in other modules.

Modules must provide answers to two questions:

- *What* is the purpose of the module?
- *How* does it achieve that purpose?

The *what* is of concern to the user of the module while the *how* is of concern to the implementer of the module.

Functions and procedures are simple modules. Their signature is a description of *what* they do while their body describes *how* it is achieved. More typically a module *encapsulates* a group of components

such as types, constants, variables, procedures, functions and so on.

To present a narrow interface to other modules, a module makes only a few components visible outside. Such components are said to be *exported* by the module. The other components are said to be *hidden* inside the module. The hidden components are used to implement the exported components.

Access to the components is often by a qualified name -- *module name. component name*. When strong safety considerations are important, modules using components of another module may be required to explicitly *import* the required module and the desired components.

Historical Perspectives and Further Reading

Exercises

1. [time/difficulty] (section) Problem statement
2. Extend the compiler to handle constant, type, variable, function and procedure definitions and references to the same.
3. What is the effect of providing lazy evaluation in an imperative programming language? % Ans: Due to the presence of side effects, the value of the actual % parameter may be different between the point of entry and the point % of evaluation.
4. Extend the compiler to handle parameterization of functions and procedures.
5. For a specific programming language, report on its provision for abstraction and generalization. Specifically, what entities may be named, what entities may be parameterized, what entities may be passed as parameters, and what entities may be returned as results (of functions). What irregularities do you find in the language?
6. Algebraic Semantics: stack, tree, queue, grade book etc
7. Lexical Scope Rules
8. Dynamic Scope Rules
9. Parameter Passing
10. Run-time Stack

Values, Domains and Types

A value is any thing that may be evaluated, stored, incorporated in a data structure, passed as an argument or returned as a result.

What is a type?

- *Realist:* A type is a set of values.
- *Idealist:* No. A type is a conceptual entity whose values are accessible only through the interpretive filter of type.
- *Beginning Programmer:* Isn't a type a name for a set of values?
- *Intermediate Programmer:* A type is a set of values and operations.
- *Advanced Programmer:* A type is a way to classify values by their properties and behavior.
- *Algebraist:* Ah! So a type is an algebra, a set of values and operations defined on the values.
- *Type checker:* Types are more practical than that, they are constraints on expressions to ensure compatibility between operators and their operand(s).
- *Type Inference System:* Yes and more, since a type system is a set of rules for associating with every expression a unique and most general type that reflects the set of all meaningful contexts in which the expression may occur.
- *Program verifier:* Lets keep it simple, types are the behavioral invariants that instances of the type must satisfy.
- *Software engineer:* What is important to me is that types are a tool for managing software development and evolution.
- *Compiler:* All this talk confuses me, types specify the storage requirements for variables of the type.

Keywords and phrases: value, domain, type, type constructor, Cartesian product, disjoint union, map, power set, recursive type, binding, strong and weak typing, static and dynamic type checking, type inference, type equivalence, name and structural equivalence, abstract types, generic types. block, garbage collection, static and dynamic links, display, static and dynamic binding, activation record, environment, Static and Dynamic Scope, aliasing, variables, value, result, value-result, reference, name, unification, eager evaluation, lazy evaluation, strict, non-strict, Church-Rosser, overloading, polymorphism, monomorphism, coercion, transfer functions.

A computation is a sequence of operations applied to a value to yield a value. Thus *values* and *operations* are fundamental to computation. Values are the subject of this chapter and operations are the subject of later chapters.

In mathematical terminology, the sets from which the arguments and results of a function are taken are known as the function's ``domain'' and ``codomain'', respectively. Consequently, the term *domain* will denote any set of values that can be passed as arguments or returned as results. Associated with every domain are certain ``essential'' operations. For example, the domain of natural numbers is equipped with an the ``constant'' operation which produces the number zero and the operation that constructs the successor of any number. Additional operations (such as addition and multiplication) on the natural numbers may be defined using these basic operations.

Programming languages utilize a rich set of domains. Truth values, characters, integers, reals, records, arrays, sets, files, pointers, procedure and function abstractions, environments, commands, and definitions are but some of the domains that are found in programming languages. There are two approaches to domains. One approach is to assume the existence of a *universal* domain. It contains all those objects which are of computational interest. The second approach is to begin with a small set of values and some rules for combining the values and then to construct the *universe* of values. Programming languages follow the second approach by providing several basic sets of values and a set of domain constructors from which additional domains may be constructed.

Domains are categorized as *primitive* or *compound*. A *primitive domain* is a set that is fundamental to the application being studied. Its elements are atomic. A *compound domain* is a set whose values are constructed from existing domains by one or more domain constructors.

> **Aside.** It is common in mathematics to define a set but fail to give an effective method for determining membership in the set. Computer science on the other hand is concerned with determining membership with in a finite number of steps. In addition, a program is often constrained by requirements to complete its work with in bounds of time and space. % In computer science ... streams ... infinite sequences ... % halting problem ... robust
>
> **Terminology.** *Domain theory* is the study of structured sets and their operations. A domain is a set of elements and an accompanying set of operations defined on the domain.
>
> The terms *domain*, *type*, and *data type* may be used interchangeably.
>
> The term *data* refers to either an element of a domain or a collection of elements from one or more domains.
>
> The terms compound, composite and structured when applied to values, data, domains, types are used interchangeably.

Elements of Domain Theory

There are many compound domains that are useful in computer science: arrays, tuples, records, variants, unions, sets, lists, trees, files, relations, definitions, mappings, etc., are all examples of compound domains. Each of these domains may be constructed from simpler domains by one or more applications of domain constructors.

Compound domains are constructed by a domain builder. A domain builder consists of a set of operations for assembling and disassembling elements of a compound domain. The domain builders are:

- Product Domains
- Sum Domains
- Function Domains
- Power Domains
- Recursive Domains

Product Domain

The domains constructed by the *product* domain builder are called *tuples* in ML, *records* in Cobol, Pascal and Ada, and *structures* in C and C++. Product domains form the basis for relational databases and logic programming.

In the binary case, the product domain builder ,×, builds the domain **A × B** from domains **A** and **B**. The domain builder includes the assembly operation, ordered pair builder, and a set of disassembly operations called projection functions. The assembly operation, ordered pair builder, is defined as follows:

> if *a* is an element of *A* and *b* is an element of *B* then (*a, b*) is an element of $A \times B$. That is,
>
> $$A \times B = \{ (a,b) \mid a \text{ in } A, b \text{ in } B \}$$

The disassembly operations *fst* and *snd* are projection functions which extract elements from tuples. For example, *fst* extracts the first component and *snd* extracts the second element.

$$nsnd(a,b) = b$$

The product domain is easily generalized (see Figure N.1) to construct the product of an arbitrary number of domains.

Figure N.1: **Product Domain: $D_0 \times ... \times D_n$**

Assembly operation: $(a_0,...,a_n)$ in $D_0 \times ... \times D_n$ where a_i in D_i and
$D_0 \times ... \times D_n = \{ (a_0,...,a_n) \mid a_i \text{ in } D_i \}$

Disassembly operation: $(a_0,...,a_n)|_i = a_i$ for $0 <= i <= n$

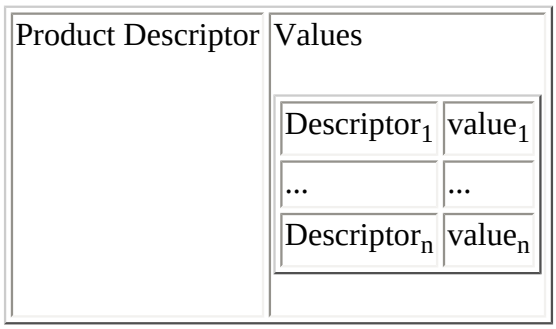
Both relational data bases and logic programming paradigm (Prolog) are based on programming with tuples.

Elements of product domains are usually implemented as a contiguous block of storage in which the components are stored in sequence. Component selection is determined by an *offset* from the address of the first storage unit of the storage block. An alternate implementation (possibly required in functional or logic programming languages) is to implement the value as a list of values. Component selection utilizes the available list operations.

> **Terminology.** The product domain is also called the ``Cartesian" or ``cross" product. In Pascal it is called a record and in C a structure.

Implementation.

Product domain elements are usually implemented as contiguous locations in memory. Using the notation introduced in the Introduction,



Sum Domain

Domains constructed by the *sum* domain builder are called *variant records* in Pascal and Ada, *unions* in Algol-68, *constructions* in ML and *algebraic types* in Miranda.

In the binary case, the sum domain builder, +, builds the domain *A* + *B* from domains *A* and *B*. The domain builder includes a pair of assembly operations and a disassembly operation. The two assembly operations of the sum builder are defined as follows:

> if *a* is an element of *A* and *b* is an element of *B* then (*A*,*a*) and (*B*, *b*) are elements of *A* + *B*. That is,
> $A + B = \{ (A,a) \mid a \text{ in } A \} \text{ union } \{ (B,b) \mid b \text{ in } B \}$

where the A and B are called tags and are used to distinguish between the elements contributed by A and the elements contributed by B.

The disassembly operation returns the element iff the tag matches the request.

$$A(A,a) = a$$

The sum domain differs from ordinary set union in that the elements of the union are labeled with the parent set. Thus even when two sets contain the same element, the sum domain builder tags them differently.

The sum domain generalizes (see Figure N.2) to sums of an arbitrary number of domains.

Figure N.2: **Sum Domain: $D_0 + ... + D_n$**

Assembly operations: (D_i, d_i) in $D_0 + ... + D_n$ and $D_0 + ... + D_n = \text{Union}_{i=0}^{n} \{ (D_i,d) \mid d \text{ in } D_i \}$

Disassembly operations: $D_i(D_i, d_i) = d_i$

Terminology. The *sum* domain is also called the *disjoint union* or *co-product* domains. In Pascal it is called a variant record and in C a union.

Pascal it is called a record and in C a structure.

Implementation.

Sum domain elements are usually implemented as a contiguous piece of memory large enough to hold a value of any of the domains and a tag which is used to determine the domain to which the value belongs. Using the notation introduced in the Introduction,

Sum Descriptor	Tag, Value

Function Domain

The domains constructed by the *function* domain builder are called *functions* in Haskell, *procedures* in Modula-3, and *procs* in SR. Although their syntax often differs from that of functions, arrays are also examples of domains constructed by the function domain builder.

The function domain builder creates the domain *A* --> *B* from the domains *A* and *B*. The domain *A* --> *B* consists of all the functions from *A* to *B*. *A* is called the domain and *B* is called the co-domain.

The assembly operation is:

(**lambda** x.e) is an element in A --> B whenever *e* is an expression containing occurrences of an identifier *x*, such that whenever a value a in A replaces the occurrences of *x* in *e*, the value e[a:x] in B results, then.

The disassembly operation is function application. It takes two arguments, an element *f* of A --> B and an element *a* of *A* and produces f(a) an element of *B*. In the case of arrays, the disassembly operation is called *subscripting*.

The function domain is summarized in Figure N.3.

Figure N.3: **Function Domain: A --> B**

Assembly operation: (**lambda** x.E) in A --> B where for all a in A, E[x:a] is a unique value in B.

Disassembly operation: (g a) in B, for g in A --> B and a in A.

Mappings (or functions) from one set to another are an extremely important compositional method. The map *m* from a element *x* of *S* (called the domain) to the corresponding element *m(x)* of *T* (called the range) is written as:

m : S --> T

where if m(x) = a and m(y) = a then x = y. Mappings are more restricted than the Cartesian product since, for each element of the domain there is a unique range element. Often it is either difficult to specify the domain of a function or an implementation does not support the full domain or range of a function. In such cases the function is said to be a *partial function*. It is for efficiency purposes that partial functions are permitted and it becomes the programmer's responsibility to inform the users of the program of the nature of the unreliability.

Arrays are mappings from an index set to an array element type. An array is a finite mapping. Apart from arrays, mappings occur as operations and function abstractions. Array values are implemented by allocating a contiguous block of storage where the size of the block is based on the product of the size of an element of the array and the number of elements in the array.

The operations provided for the primitive types are maps. For example, the addition operation is a mapping from the Cartesian product of numbers to numbers.

+: number × number --> number

The functional programming paradigm is based on programming with maps.

Terminology. The function domain is also called the *function space*.

Implementation.

Function domain elements are usually implemented in code. However, arrays are a special case of function domain and they are usually implemented in contiguous memory elements. Using the notation introduced in the Introduction,

Power Domain

Set theory provides an elegant notation for the description of computation. However, it is difficult to provide efficient implementation of the the set operations. SetL is a programming language based on sets and was used to provide an early compiler for Ada. The Pascal family of languages provide for set union and intersection and set membership. Set variables represent subsets of user defined sets.

The set of all subsets of a set is the power set and is defined:

$$P^S = \{ s \mid s \text{ is a subset of } S\}$$

Subtypes and subranges are examples of the power set constructor.

Functions are subsets of product domains. For example, the square function can be represented as a subset of the product domain Nat × Nat.

$$sqr = \{(0,0),(1,1),(2,4),(3,9),...\}$$

Generalization helps to simplify this infinite list to:

$$sqr = \{(x,x*x) \mid x \text{ in Nat}\}$$

The programming language SetL is based on computing with sets.

Set values may be implemented by using the underlying hardware for bit-strings. This makes set operations efficient but constrains the size of sets to the number of bits(typically) in a word of storage. Alternatively, set values may be implemented using software, in which case, hash-coding or lists may be used.

Some languages provide mechanisms for decomposing a type into subtypes

- one is the enumeration of the elements of the subtype.
- another is subranges since, enumeration is tedious for large sub-domains and many types have a natural ordering.

The power domain construction builds a domain of sets of elements. For a domain *A*, the power domain builder *P*() creates the domain *P*(*A*), a collection whose members are subsets of *A*.

Figure N.4: **Power Domain: P^D**

Assembly operations: Ø in P^D, { a } in P^D for *a* in D, and S_i union S_j in P^D for S_i, S_j in P^D

Recursively Defined Domain

Recursively defined domains are domains whose definition is of the form:

$$D : ... D ...$$

The definition is called recursive because the name of the domain ``recurs" on the right hand side of the definition. Recursively defined domains depend on abstraction since the name of the domain is an essential part the definition of the domain. The context-free grammars used in the definition of programming languages contain recursive definitions so programming languages are examples of recursive types.

More than one set may satisfy a recursive definition. However, it may be shown that a recursive definition always has a least solution. The least solution is a subset of every other solution.

The least solution of a recursively defined domain is obtained through a sequence of approximations (D_0, D_1,...) to the domain with the domain being the *limit* of the sequence of approximations ($D = \lim_{i \text{ --> infty}} D_i$). The limit is the smallest solution to the recursive domain definition.

We illustrate the limit construction (see Figure N.5) with three examples.

Figure N.5: **Limit Construction**

$$D_0 = \text{null}$$
$$D_{i+1} = e[D:D_i] \text{ for } i=0,...$$
$$D = \lim_{i \text{ --> infty}} D_i$$

The Natural Numbers

A representation of the natural numbers given earlier in the text was:

$$N ::= 0 \mid S(N)$$

The defining sequence for the natural numbers is:

$$N_0 = \text{Null}$$
$$N_{i+1} = 0 \mid S(N_i) \text{ for } i = 0,...$$

The definition results in the following:

$$N_0 = \text{Null}$$
$$N_1 = 0$$
$$N_2 = 0 \mid S(0)$$
$$N_3 = 0 \mid S(0) \mid S(S(0))$$
$$N_4 = 0 \mid S(0) \mid S(S(0)) \mid S(S(S(0)))$$
$$...$$

The factorial function

For functions, Null can be replaced with _|_ which means undefined.

The factorial function is often recursively defined as:

$$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fac(n-1) & \text{otherwise} \end{cases}$$

The factorial function is approximated by a sequence of functions where the function fac_0 is defined as

$$fac_0(n) = \_|\_$$

And the function fac_{i+1} is defined as

$$fac_{i+1}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fac_i(n-1) & \text{otherwise} \end{cases}$$

Writing the functions as sets of ordered pairs helps us to understand the limit construction.

$fac_0 = \{ \}$

$fac_1 = \{ (0,1) \}$

$fac_2 = \{ (0,1), (1,1) \}$

$fac_3 = \{ (0,1), (1,1), (2,2) \}$

$fac_4 = \{ (0,1), (1,1), (2,2), (3,6) \}$

...

Note that each function in the sequence includes the previously defined function and the sequence suggests that

$$fac = \lim_{i \text{ --> infty}} fac_i$$

The proof of this last equation is beyond the scope of this text. This construction suggests that recursive definitions can be understood in terms of a family of non-recursive definitions and in format common to each member of the family.

Ancestors

For logical predicates, Null can be replaced with *false*. A recursive definition of the ancestor relation is:

$$ancestor(A,D), \text{ if } \begin{cases} parent(A,D) \text{ or} \\ parent(A,I) \ \& \ ancestor(I,D) \end{cases}$$

The ancestor relation is approximated by a sequence of relations:

$$ancestor_0(A,D) = \text{false}$$

And the relation $ancestor_i$ is defined as

$$ancestor_{i+1}(A,D), \text{ if } \begin{cases} parent(A,D) \text{ or} \\ parent(A,I) \ \& \ ancestor_i(I,D) \end{cases}$$

Writing the relations as sets of order pairs helps us to understand the limit construction. An example will help. Suppose

we have the following:

parent(John, Mary)
parent(Mary, James)
parent(James, Alice)

then we have:

$ancestor_0 = \{ \}$

$ancestor_1 = \{ (John, Mary), (Mary, James), (James, Alice)\} \}$

$ancestor_2 = ancestor_1$ union {(John, James), (Mary, Alice)}

$ancestor_3 = ancestor_2$ union { (John,Alice) }

Again note that each predicate in the sequence includes the previously defined predicate and the sequence suggests that

$$ancestor = \lim_{i \text{ --> } infty} ancestor_i$$

Linear Search

The final example of domain construction is a recursive variant of linear search.

```
Loop : if i < n --> if a[i] != target --> i := i + 1; Loop
                    fi
       fi
```

$Loop_0$ is defined as:

$$Loop_0 = \_|\_$$

and $Loop_{i+1}$ is defined as:

```
Loop_{i+1} : if i < n --> if a[i] != target --> i := i + 1; Loop_i
                          fi
             fi
```

with the result of unrolling the recursion into a sequence of if-commands.

Implementation

Since recursively defined domains like lists, stacks and trees are unbounded (in general may be infinite objects) they are implemented using a product domain where one domain is a node and one or more are address domains. In Pascal, Ada and C such domains are defined in terms of pointers while Prolog and functional languages like ML and Miranda allow recursive types to be defined directly.

Type Systems

A large percentage of errors in programs is due to the application of operations to objects of incompatible types. Type systems have been developed to assist the programmer in the detection of these errors. A *type system* is a set of rules for defining types and associating a type with expression in the language. A type system *rejects* an expression if it does not associate a type with the expression. Type checking may performed at compile time or run time or both.

Definition N.M: Type **System**

A *type system* is a set of rules for defining types and associating a type with expression in the language. A type system *rejects* an expression if it does not associate a type with the expression.

If the errors are to be detected at compile time then a *static* type checking system is required. One approach to static type checking is to require the programmer to specify the type of each object in the program. This permits the compiler to perform type checking before the execution of the program and this is the approach taken by languages like Pascal, Ada, C++, and Java. Another approach to static type checking is to add type inference capabilities to the compiler. In such a system, the compiler performs type checking by means of a set of type inference rules and is able to flag type errors prior to runtime. This is the approach taken by Miranda and Haskell.

If the error detection is to be delayed until execution time, then *dynamic* type checking is required. In dynamic type checking, each data value is tagged with type information so that the run time environment can check type compatibility and possibly perform type conversions if necessary. The programming languages Lisp, Scheme and Small-talk are examples of dynamically typed languages

Type Checking

Machine operations manipulate bit patterns. Whether a bit pattern represents a character, an integer, a real, an address, or an instruction, any machine operation may be applied to any data item. There is no type checking at the assembly language level. Languages which permit operations to be applied to data of any type are called *untyped*. Prolog is one of the few high-level languages that is an untyped language. In Prolog, lists can consist of elements of any type and different sorts of values may be compared with the equality relation `=', but such comparison will always yield *false*.

Example. In C, the decision portion of any control structure can be any expression that produces a value. If the value is 0, it is treated as false and any nonzero values is treated as true. Since the value of an assignment command is the value of its right-hand side, the command `if x = 4 ...` any else clause will be ignored. In C characters are treated as integers and thus may occur in arithmetic expressions. C's type system is not robust enough to protect novice programmers these and other errors.

The advantage of untyped languages is their flexibility. The programmer has complete control over how a data value is used but must assume full responsibility for detecting the application of operations to objects of incompatible type.

Figure N.M: **Type Checking Definitions**

A language is said to be

- *untyped* if no type abstractions are inforced,
- *strongly typed* if it enforces type abstractions (operations may be applied only to objects of the appropriate type),
- *statically typed* if the type of each expression can be determined from the program text,
- *dynamically typed* if the determination of the type of some expression depends on the runtime behavior of the program.

A *strongly typed* language enforces type abstractions. Most languages are strongly typed with respect to the primitive types supported by the language. So, for example, the mixing of numeric and character types that is permissible in C is not permitted in Pascal or Ada.

Strong typing helps to insure the security and portability of the code and it often requires the programmer to explicitly define the types of each object in a program. It is also important in compilation for picking appropriate operations and for optimization.

If the types of all variables can be known from an examination of the text (i.e. at compile time), then the a language is said to be *statically typed*. Pascal, Ada, and Haskell are examples of statically typed languages.

Static typing is widely recognized as a requirement for the production of safe and reliable software. Static type checking implies that the types are checked at compile time. Static typing is chosen when efficiency in execution time is important and compiler support is used to support good software engineering practices.

If the type of a variable can only be known at run-time, then the language is said to be *dynamically typed*. Lisp and Smalltalk are examples of dynamically typed languages.

Dynamic type checking implies that the types are checked at execution time and that every value is tagged to identify its type in order to make the type checking possible. The penalty for dynamic type checking is additional space and time overheads.

Dynamic typing is often justified on the assumption that its flexibility permits the rapid prototyping of software.

Prolog relies on pattern matching to provide a semblance of type checking. There is active research on adapting type checking systems for Prolog.

Modern functional programming languages such as Miranda and Haskell and object-oriented languages combine the safety of static type checking with the flexibility of dynamic type checking through polymorphic types.

Type Equivalence

Two unnamed types (sets of objects) are the same if they contain the same elements. The same cannot be said of named types for if they were, then there would be no need for the disjoint union type. When types are named, there are two major approaches to determining whether two types are equal.

Name Equivalence

In name equivalence two types are the same if they have the same name. Types that are given different names are treated as distinct and cannot be accidentally mixed just because their structure happens to be the same. Name equivalence requires type definitions to be global.

Name equivalence was chosen for Modula-2, Ada, C (for records), and Miranda. The predecessor of Modula-2, Pascal violates name equivalence since file type names are not required to be shared by different programs accessing the same file.

Structural Equivalence

In structural equivalence, the names of the types are ignored and the elements of the types are compared for equality. It is possible that two logically different types may turn out to be the same by coincidence and may be mixed. Type definitions are not required to be global. Structural equivalence is important in programming distributed systems, in which separate programs must communicate typed data.

Definition N.1:

Two types *T, T'* are *name equivalent* iff *T* and *T'* are the same name.

Two types *T, T'* are *structurally equivalent* iff *T* and *T'* have the same set of values.

The following three rules may be used to determine if two types are structurally equivalent.

- A type name is structurally equivalent to its self.
- Two types are structurally equivalent if they are formed by applying the same type constructor (recursively) to structurally equivalent types.
- After a type declaration, *type n = T*, the type name *n* is structurally equivalent to *T*.

Structural equivalence was chosen by Algol-68 and C (except for records) because it is easy to implement.

Type Inference

Type inference is the general problem of transforming untyped or partially typed syntax into well-typed terms. Pascal constant declarations are an example of type inference, the type of the name is inferred from the type of the constant. In Pascal's for loop the type of the loop index can be inferred from the types of the loop limits and thus the loop index should be a variable local to the loop. The programming languages Miranda and Haskell are statically types and provide powerful type inference systems so that a programmer need not declare any types. The languages also permit programmers to provide explicit type specifications.

A type checker must be able to

- determine if a program is well typed and
- if the program is well typed, determine the type of any expression in the program.

Type inference axioms

Axiom

given that: f is of type A --> B and x is of type A
infer that: f(x) is type correct and has type B

Type declarations

Even languages that provide a type inference system permit programmers to make explicit declarations of type. Even if the compiler can correctly infer types, human readers may have to scan several pages of code to determine the type of a function. Slight errors by the programmer can cause the compiler to emit obscure error messages or to infer a different type than intended. For these reasons it is good programming practice to explicitly state types on all but the most

obvious cases.

Examples. In Miranda (a functional language) the types for the arithmetic + operation are declared as follows:

+ :: num --> num --> num

In Pascal the type of a function for computing the circumference of a circle is declared as follows:

function circumference(radius : real) : real;

Polymorphism

A type system is *monomorphic* if each constant, variable, parameter, and function result has a unique type. Type checking in a monomorphic system is straightforward. But purely monomorphic type systems are unsatisfactory for writing reusable software. Many algorithms such as sorting and list and tree manipulation routines are *generic* in the sense that they depend very little on the type of the values being manipulated. For example, a general purpose array sorting routine cannot be written in Pascal. Pascal requires that the element type of the array be part of the declaration of the routine. This means that different sorting routines must be written for each element type and array size.

Completely monomorphic systems are rare. Most programming languages contain some operators or procedures which permit arguments of more than one type. For example, Pascal's input and output procedures permit variation both in type and in number of arguments. This is an example of *overloading*.

Definition N.2:

- *Monomorphism*: every constant, variable, parameter, operator and function has a unique type.
- *Overloading* refers to the use of a single syntactic identifier to refer to several different operations discriminated by the type and number of the arguments to the operation.
- *Polymorphism*: an operator, function or procedure that has a family of related types and operates uniformly on its arguments regardless of type.
- A *polymorphic* operation is one that can be applied to different but related types of arguments.

The type of the plus operation defined for integer addition is

+: int × int --> int

When the same operation symbol is used for the plus operation for rational numbers and for set union, the symbol as in Pascal it is overloaded. Most programming languages provide for the overloading of the arithmetic operators. A few programming languages (Ada among others) provide for programmer defined overloading of both built-in and programmer defined operators.

When overloaded operators are applied to mixed expressions such as plus to an integer and a rational number there are two possible choices. Either the evaluation of the expression fails or one or more of the subexpressions are *coerced* into a corresponding object of another type. Integers are often coerced into the corresponding rational number. This type of coercion is called *widening*. When a language permits the coercion of a real number into an integer (by truncation for example) the coercion is called *narrowing*. Narrowing is not usually permitted in a programming language since information is usually lost. Coercion is an issue in programming languages because numbers do not have a uniform representation. This type of overloading is called *context-dependent overloading*.

Many languages provide type transfer functions so that the programmer can control where and when the type coercion is performed. Truncate and round are examples of type transfer functions.

Overloading is sometimes called *ad-hoc polymorphism*.

Most sorting algorithms can be explained without referring to the kind (type) of data being sorted. Typically, the data is an array of pointers to records each with an associated *key*. The type of the key does not matter as long as there is a "comparison" procedure which finds the minimum of a pair of keys. The sorting procedures use the compare two keys using the comparison procedure and swap the records by resetting pointers accordingly. However, in a strongly typed language this is not possible since the pointer type depends on the record type. This forces us to write a separate procedure for each type of data.

Stacks, queues, lists and trees also are largely type independent and yet in a strongly typed language, separate code must be written for each element type. Some languages permit type variables and these data structures can be defined with a type variable which then allows the user

A type system is *polymorphic* if abstractions operate uniformly on arguments of a family of related types.

This type of polymorphism is sometimes called *parametric polymorphism*.

Generalization can be applied to may aspects of programming languages.

Sometimes there are several domains which share a common operation. For example, the natural numbers, the integers, the rationals, and the reals all share the operation of addition. So, most programming languages use the same addition operator to denote addition in all of these domains. Pascal extends the use of the addition operator to represent set union. The multiple use of a name in different domains is called *overloading*. Ada permits user defined overloading of built in operators.

Prolog permits the programmer to use the same functor name for predicates of different arity thus permitting the overloading of functor names. This is an example of *data generalization* or polymorphism.

While the *parameterization* of an object gives the ability to deal with more than one particular object, *polymorphism* is the ability of an operation to deal with objects of more than a single type.

Generalization of control has focused on advanced control structures (RAM): iterators, generators, backtracking, exception handling, coroutines, and parallel execution (processes).

Type Completeness

> **Principle of Type Completeness** No operation should be arbitrarily restricted in the types of the values involved.

Pragmatics

type declarations: spelling, type checking, type inference vs type declaration

Historical Perspectives and Further Reading

Declarations

Constants

literals

User Defined Types

Miranda's universe of values is numbers, characters and boolean values while Pascal provides boolean, integer, real, and char.

Declarations of variables of the primitive types have the following form in the imperative languages.

```
I : T;    { Modula-2: item I of type T}
T  I;     // C++: item I of type T
```

Declarations of enumeration types involve listing of the values in the type.

Here are the enumerations of the items I_1,...,I_n of type T.

```
T = {I1,...,In};    { Modula-2}\\
enum T {I1,...,In}; // C++ \\
T ::= I1 | ... | In  || Miranda
```

Modula-2, Ada, C++, Prolog, Scheme, Miranda -- list primitive types Haskell provides the built in functions `fst` and `snd` to extract the first and second elements from binary tuples.

Imperative languages require that the elements of a tuple be named. Modula-2 is typical; product domains are defined by record types:

```
record
  I1 : T1;
  ...
  In : Tn;
end
```

The I_is are names for the component of the tuple. The individual components of the record are accessed by the use of qualified names. for example, if MyRec is a element of the above type, then the first component is referenced by MyRec.I_1 and the last component is referenced by MyRec.I_n.

C and C++ calls a product domain a structure and uses the following type declaration:

```
struct name {
  T1 I1;
  ...
  Tn : In;
};
```

The I_is are names for the entries in the tuple.

Prolog does not require type declaration and elements of a product domain may be represented in a number of ways, one way is by a term of the form:

$$name(I_1,...I_n)$$

The I_is are the entries in the tuple. The entries are accessed by pattern matching. Miranda does not require type declaration and the elements of a product domain are represented by tuples.

$$(I_1,...I_n)$$

The I_is are the entries in the tuple.

Here is an example of a variant record in Pascal.

```
% From condensed pascal
type Shape = (Square, Rectangle, Rhomboid, Trapezoid, Parallelogram);
   Dimensions = record
      case WhatShape : Shape of
         Square : (Side1: real);
         Rectangle : (Length, Width : real);
         Rhomboid : (Side2: real; AcuteAngle: 0..360);
         Trapezoid : (Top1, Bottom, Height: real);
         Parallelogram : (\=Top2, Side3: real;
             ObtuseAngle: 0..360)
      end;
var FourSidedObject : Dimensions;
```

The initialization of the record should follow the sequence of assigning a value to the tag and then to the appropriate subfields.

```
FourSidedObject.WhatShape := Rectangle;
FourSidedObject.Length := 4.3;
FourSidedObject.Width := 7.5;
```

The corresponding definition in Miranda is

```
Dimensions :: Square num |
   Rectangle num num |
   Rhomboid num num |
   Trapezoid num num num |
   Parallelogram num num num

area Square S = S*S
area Rectangle L W = L * W
...
```

Modula-2, Ada, C++, Prolog, Scheme, Miranda Disjoint union values are implemented by allocating storage based on the largest possible value and additional storage for the tag.

Modula-2

```
array[domain_type] of range_type {Modula-2}
range_type identifier [natural number] // C++
```

Prolog and Miranda do not provide for an array type and while Scheme does, it is not a part of the purely functional part of Scheme. **Modula-2, Ada, C++, Prolog, Scheme, Miranda -- mapping type** In Pascal the notation [i..j] indicates the subset of an ordinal type from element i to element j inclusive. In addition to subranges, Miranda provides infinite lists [i..] and finite and infinite arithmetic series [a,b..c], [a,b..] (the interval is (b-a)). Miranda also provides list comprehensions which are used to construct lists (sets). A list comprehension has the form [exp | qualifier]

```
sqs = [ n*n | n <-[1..] ]

factors n = [ r | r <-[1..n div 2]; n mod r = 0 ]

knights_moves [i,j] = [ [i+a,j+b] | a,b <-[-2..2]; a\verb+^+2+\verb+^+2=5 ]
```

Modula-2, Ada, C++, Prolog, Scheme, Miranda -- power set Prolog

```
[]
[I0,...In]
[H | T]
```

The Miranda syntax for lists is similar to that of Prolog however, elements of lists must be all of the same type.

```
[*]
[I_0,...In]
[H | T]
```

Recursive types in imperative programming languages are usually defined using a pointer type. Pointer types are an additional primitive type. Pointers are addresses.

```
{Modula-2: the pointer and the list}
type NextItem = \verb+^+ListType
   ListType = record
     item : Itemtype;
     next : NextItem
   end;

// C++: the list type
struct list {
  ItemType Item;
  list* Next; // pointer to list
};

|| Miranda: list of objects of type T and
|| a binary tree of type T
[T]
tree ::= Niltree | Node T tree tree
```

Referencing/Dereferencing

```
type ListType = record
      item : Itemtype;
```

```
        next : ListType
    end;
```

Recursive values are implemented using pointers. The run-time support system for the functional and logic programming languages, provides for automatic allocation and recovery of storage (garbage collection). The alternative is for the language to provide access to the run-time system so that the programmer can explicitly allocate and recover the linked structures.

Variables

\marginpar{state:store} It is frequently necessary to refer to an arbitrary element of a type. Such a reference is provided through the use of *variables*. A *variable* is a name for an arbitrary element of a type and it is a generalization of a value since it can be the name of any element.

Functions and Procedures

Persistent Types

A *persistent variable* is one whose lifetime transcends an activation of any particular program. In contrast, a *transient variable* is one whose lifetime is bounded by the activation of the program that created it. Local and heap variables are transient variables.

Most programming languages provide different types for persistent and transient variables. Typically *files* are the only type of persistent variable permitted in a programming language. Files are not usually permitted to be transient variables.

Most programming languages provide different types for persistent and transient The type completeness principle suggests that all the types of the programming language should be allowed both for transient variables and persistent variables. A language applying this principle would be simplified by having no special types, commands or procedures for input/output. The programmer would be spared the effort of converting data from a persistent data type to a transient data type on input and *vice versa* on output.

A persistent variable of array type corresponds to a *direct file*. If heap variables were persistent then the storage of arbitrary data structures would be possible.

Exercises

1. Extend the compiler to handle constant, type, variable, function and procedure definitions and references to the same.
2. Static and dynamic scope
3. Define algebraic semantics for the following data types.
    1. Boolean

```
ADT Boolean
Operations
  and(boolean,boolean) --> boolean
  or(boolean,boolean) --> boolean
  not(boolean) --> boolean
Semantic Equations
  and(true,true) = true
  or(true,true) = true
```

```
not(true) = false
not(false) = true
```

Restrictions

2. Integer
3. Real
4. Character
5. String

4. Name or Structure equivalence (type checking)
5. Algebraic Semantics: stack, tree, queue, grade book etc.
6. Abstraction
7. Generalization
8. Name or Structure equivalence (type checking)
9. Extend the compiler to handle additional types. This requires modifications to the syntax of the language with extensions of the scanner, parser, symbol table and code generators.

Logic Programming

N. Wirth: *Program = data structure + algorithm*
R. Kowalski: *Algorithm = logic + control*
J. A. Robinson: *A program is a theory (in some logic) and computation is deduction from the theory.*

Logic programming is characterized by programming with relations and inference.

Keywords and phrases: Horn clause, Logic programming, inference, modus ponens, modus tollens, logic variable, unification, unifier, most general unifier, occurs-check, backtracking, closed world assumption, meta programming, pattern matching. set, relation, tuple, atom, constant, variable, predicate, functor, arity, term, compound term, ground, nonground, substitution, instance, instantiation, existential quantification, universal quantification, unification, modus ponens, proof tree, goal, resolvent.

A logic program consists of a set of axioms and a goal statement. The rules of inference are applied to determine whether the axioms are sufficient to ensure the truth of the goal statement. The execution of a logic program corresponds to the construction of a proof of the goal statement from the axioms.

In the logic programming model the programmer is responsible for specifying the basic logical relationships and does not specify the manner in which the inference rules are applied. Thus

Logic + Control = Algorithms

Logic programming is based on tuples. Predicates are abstractions and generalization of the data type of tuples. Recall, a tuple is an element of

$$S0 \times S1 \times ... \times Sn$$

The squaring function for natural numbers may be written as a set of tuples as follows:

{(0,0), (1,1), (2,4) ...}

Such a set of tuples is called a relation and in this case the tuples define the squaring relation.

sqr = {(0,0), (1,1), (2,4) ...}

Abstracting to the name sqr and generalizing an individual tuple we can define the squaring relation as:

sqr = (x,x2)

Parameterizing the name gives:

sqr(X,Y) <-- Y is } X*X

In the logic programming language Prolog this would be written as:

sqr(X,Y) <-- Y is } X*X.

Note that the set of tuples is named sqr and that the parameters are X and Y. Prolog does not evaluate its arguments unless required, so the expression Y is X*X forces the evaluation of X*X and unifies the answer with Y. The Prolog code

P <-- Q.

may be read in a number of ways; it could be read P where Q or P if Q. In this latter form it is a variant of the first-order predicate calculus known as Horn clause logic. A complete reading of the sqr predicate the point of view of logic is: for every X and Y, Y is the sqr of X if Y is X*X. From the point of view of logic, we say that the variables are universally quantified. Horn clause logic has a particularly simple inference rule which permits its use as a model of computation. This computational paradigm is called Logic programming and deals with relations rather than functions or assignments. It uses facts and rules to represent information and deduction to answer queries. Prolog is the most widely available programming language to implement this computational paradigm.

Relations may be composed. For example, suppose we have the predicates, male(X), siblingof(X,Y), and parentof(Y,Z) which define the obvious relations, then we can define the predicate uncleof(X,Z) which implements the obvious relation as follows:

```
uncleof(X,Z) <-- male(X), siblingof(X,Y), parentof(Y,Z).
```

The logical reading of this rule is as follows: ``for every X,Y and Z, X is the uncle of Z, if X is a male who has a sibling Y which is the parent of Z." Alternately, ``X is the uncle of Z, if X is a male and X is a sibing of Y and Y is a parent of Z." %fatherof(X,Y),fatherof(Y,Z) defines paternalgrandfather(X,Z)

The difference between logic programming and functional programming may be illustrated as follows. The logic program

```
f(X,Y) <-- Y = X*3+4
```

is an abreviation for

\forall X,Y (f(X,Y) <-- Y = X*3+4)

which asserts a condition that must hold between the corresponding domain and range elements of the function. In contrast, a functional definition introduces a functional object to which functional operations such as functional composition may be applied.

Logic programming has many application areas:

- Relational Data Bases
- Natural Language Interfaces
- Expert Systems
- Symbolic Equation solving
- Planning
- Prototyping
- Simulation
- Programming Language Implementation

Syntax

There are just four constructs: constants, variables, function symbols, predicate symbols, and two logical connectives, the comma (and) and the implication symbol.

Core Prolog

P in Programs
C in Clauses
Q in Queries
A in Atoms
T in Terms
X in Variables

Program ::= Clause... Query | Query
Clause ::= Predicate . | Predicate :- PredicateList .
PredicateList ::= Predicate | PredicateList , Predicate
Predicate ::= Atom | Atom(TermList)
TermList ::= Term | TermList , Term
Term ::= Numeral | Atom | Variable | Structure
Structure ::= Atom (TermList)
Query ::= ?- PredicateList .
Numeral ::= an integer or real number
Atom ::= string of characters beginning with a lowercase letter or encluded in apostrophes.
Variable ::= string of characters beginning with an uppercase letter or underscore

Terminals = {Numeral, Atom, Variable, :-, ?-, comma, period, left and right parentheses }

While there is no standard syntax for Prolog, most implementations recognize the grammar in Figure M.N.

Figure M.N: Prolog grammar

P in Programs
C in Clauses
Q in Query
H in Head
B in Body
A in Atoms
T in Terms
X in Variable

P ::= C... Q...
C ::= H [:- B] .
H ::= A [(T [,T]...)]
B ::= G [, G]...
G ::= A [([X | T]...)]
T ::= X | A [(T...)]
Q ::= ?- B .

CLAUSE, FACT, RULE, QUERY, FUNCTOR, ARITY, ORDER, UNIVERSAL QUANTIFICATION, EXISTENTIAL QUANTIFICATION, RELATIONS

In logic, relations are named by predicate symbols chosen from a prescribed vocabulary. Knowledge about the relations is then expressed by sentences constructed from predicates, connectives, and formulas. An n-ary predicate is constructed from prefixing an n-tuple with an n-ary predicate symbol.

A logic program is a set of axioms, or rules, defining relationships between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have the desired meaning.

The basic constructs of logic programming, terms and statements are inherited from logic. There are three basic statements: facts, rules and queries. There is a single data structure: the logical term.

Facts, Predicates and Atoms

Facts are a means of stating that a relationship holds between objects.

```
father(bill,mary).
plus(2,3,5).
...
```

This fact states that the relation father holds between bill and mary. Another name for a relationship is predicate.

Queries

A query is the means of retrieving information from a logic program.

```
?- father(bill,mary).
?- father(bill,jim).
```

Note that the text terminates queries with a question mark rather than preceding.

Semantics

The operational semantics of logic programs correspond to logical inference. The declarative semantics of logic programs are derived from the term model commonly referred to as the Herbrand base. The denotational semantics of logic programs are defined in terms of a function which assigns meaning to the program.

There is a close relation between the axiomatic semantics of imperative programs and logic programs. A logic program to sum the elements of a list could be written as follows.

```
sum([Nth],Nth).
sum([Ith|Rest],Ith + Sum_Rest) <-- sum(Rest,Sum_Rest).
```

A proof of its correctness is trivial since the logic program is but a statement of the mathematical properties of the sum.

```
A[N] = sum_{i=N}^N A[i]}
sum([A[N]],A[N]).
sum_{i=I}^N A[i] = A[I] + S if 0 < I, sum_{i=I+1}^N A[i] = S}
sum([A[I],...,A[N]], A[I]+S) <--  sum([A[I+1],...,A[N]],S).
```

Operational Semantics

Definition: The meaning of a logic program P, M(P), is the set of unit goals deducible from P.

- Logic Program A logic program is a finite set of facts and rules.
- Interpretation and meaning of logic programs.

  The rule of instantiation (P(X) deduce P(c)). The rule of deduction is modus ponens. From A :- B1, B2, ..., Bn. and B1', B2', ..., Bn' infer A'. Primes indicate instances of the corresponding term.

  The meaning M(P) of a logical program P is the set of unit goals deducible from the program.

  A program P is correct with respect to some intended meaning M iff the meaning of P M(P) is a subset of M (the program does not say things that were not intended).

  A program P is complete with respect to some intended meaning M iff M is a subset of M(P) (the program says everything that was intended).

  A program P is correct and complete with respect to some intended meaning M iff M = M(P).

The operational semantics of a logic program can be described in terms of logical inference using unification and the inference rule resolution. The following logic program illustrates logical inference.

```
a.
b <--  a.
b?
```

We can conclude b by modus ponens given that b <-- a and a. Alternatively, if b is assume to be false then from b <-- a and modus tollens we infer ¬ a but since a is given we have a contradiction and b must hold. The following program illustrates unification.

```
parent_of(a,b).
parent_of(b,c).
ancestor_of(Anc,Desc) <-- parent_of(Anc,Desc).
ancestor_of(Anc,Desc) <-- parent_of(Anc,Interm) \wedge
   ancestor_of(Interm,Desc).
parent_of(a,b)?
ancestor_of(a,b)?
ancestor_of(a,c)?
ancestor_of(X,Y)?
```

Consider the query `ancestor_of(a,b)?'. To answer the question ``is a an ancestor of b", we must select the second rule for the ancestor relation and unify a with Anc and b with Desc. Interm then unifies with c in the relation parent_of(b,c). The query, ancestor_of(b,c)? is answered by the first rule for the ancestor_of relation. The last query is asking the question, ``Are there two persons such that the first is an ancestor of the second." The variables in queries are said to be existentially quantified. In this case the X unifies with a and the Y unifies with b through the parent_of relation. Formally,

---

Definition M.N:

A unifier of two terms is a substitution making the terms identical. If two terms have a unifier, we say they unify.

For example, two identical terms unify with the identity substitution. concat([1,2,3],[3,4],List) and concat([X|Xs],Ys,[X|Zs]) unify with the substitutions {X = 1, Xs = [2,3], Ys = [3,4], List = [1|Zs]}

There is just one rule of inference which is resolution. Resolution is much like proof by contradiction. An instance of a relation is ``computed" by constructing a refutation. During the course of the proof, a tree is constructed with the statement to be proved at the root. When we construct proofs we will use the symbol ¬ to mark formulas which we either assume are false or infer are false and the symbol [] for contradiction. Resolution is based on the inference rule modus tollens and unification. This is the modus tollens inference rule.

```
From  ¬ B
and  B <-- A0,...,An
infer ¬ A0 or...or ¬ An
```

Notice that as a result of the inference there are several choices. Each ¬ A_{i} is a formula marking a new branch in the proof tree. A contradiction occurs when both a formula and its negation appear on the same path through the proof tree. A path is said to be closed when it contains a contradiction otherwise a path is said to be open. A formula has a proof if and only if each path in the proof tree is closed. The following is a proof tree for the formula B under the hypothesises A0 and B <-- A0,A_{1}.

```
1  From   ¬ B
2  and   A0
3  and   B <-- A0,A_{1}
```

```
4  infer   ¬ A0   or}  ¬ A_{1}
5  choose   ¬ A0
6  contradiction   []
7  choose   ¬ A_{1}
8  no further possibilities}  open}
```

There are two paths through the proof tree, 1-4, 5, 6 and 1-4, 7, 8. The first path contains a contradiction while the second does not. The contradiction is marked with [].

As an example of computing in this system of logic suppose we have defined the relations parent and ancestor as follows:

```
1.  parent_of(ogden,anthony)
2.  parent_of(anthony,mikko)
3.  parent_of(anthony,andra)
4.  ancestor_of(A,D) <-- parent_of(A,D)
5.  ancestor_of(A,D) <-- parent_of(A,X)
6.  ancestor_of(X,D)
```

where identifiers beginning with lower case letters designate constants and identifiers beginning with an upper case letter designate variables. We can infer that ogden is an ancestor of mikko as follows.

```
¬ ancestor(ogden,mikko)  the assumption
¬ parent(ogden,X)  or}  ¬ ancestor(X,mikko)
     resolution}
¬ parent(ogden,X)  first choice
¬ parent(ogden,anthony)  unification with first entry
[]  produces a contradiction
¬ ancestor(anthony,mikko)  second choice
¬ parent(anthony,mikko)  resolution
[]  A contradiction of a
  fact.}
```

Notice that all choices result in contradictions and so this proof tree is a proof of the proposition that ogden is an ancestor of mikko. In a proof, when unification occurs, the result is a substitution. In the first branch of the previous example, the term anthonoy is unified with the variable X and anthony is substituted for all occurences of the variable X.

UNIVERSAL QUANTIFICATION, EXISTENTIAL QUANTIFICATION

The unification algorithm can be defined in Prolog. Figure~\ref{lp:unify} contains a formal definition of unification in Prolog

Figure MN: Unification Algoririthm

```
unify(X,Y) <-- X == Y.
unify(X,Y) <-- var(X), var(Y), X=Y.
unify(X,Y) <-- var(X), nonvar(Y), \+ occurs(X,Y), X=Y.
unify(X,Y) <-- var(Y), nonvar(X), \+ occurs(Y,X), Y=X.
unify(X,Y) <-- nonvar(X), nonvar(Y), functor(X,F,N), functor(Y,F,N),
X =..[F|R], Y =..[F|T], unify_lists(R,T).

unify_lists([ ],[ ]).
unify_lists([X|R],[H|T]) <-- unify(X,H), unify_lists(R,T).

occurs(X,Y) <-- X==Y.
occurs(X,T) <-- functor(T,F,N), T =..[F|Ts], occurs_list(X,Ts).

occurs_list(X,[Y|R]) <-- occurs(X,Y).
```

```
occurs_list(X,[Y|R]) <-- occurs_list(X,R).
```

Unification subsumes

- single assignment
- parameter passing
- record allocation
- read/write-once field-access in records

```
\begin{array}{l}
\frac{A1 <-- B.},  ?- A1, A2,...,An.}}{?- B, A2,...,An.}}

\frac{?- true, A1, A2,...,An.}}{?- A1, A2,...,An.}}
\caption{Inference Rules\label{lp:ir}}
```

To illustrate the inference rules, consider the following program consisting of a rule, two facts and a query:

a <-- b \wedge c . b <-- d . b <-- e . ?- a .

By applying the inference rules to the program we derive the following additional queries:

?- b \wedge c . ?- d \wedge c . ?- e \wedge c. ?- c. ?-

Among the queries is an empty query. The presence of the empty query indicates that the original query is satisfiable, that is, the answer to the query is yes. Alternatively, the query is a theorem, provable from the given facts and rules.

Inference and Unification

Definition: The law of universal modus ponens says that from

```
R = (A :- B1,...,Bn) and
B'1.
...
B'n.
```

A' can be deduced, if A' :- B'1,...,B'n is an instance of R.

Definition: A logic program is a finite set of rules.

Definition: An existentially quantified goal G is a logical consequence of a program P iff there is a clause in P with an instance A :- B1,...,Bn, n \geq 0 such that B1,...,Bn are logical consequences of P and A is an instance of G.

The control portion of the the equation is provide by an inference engine whose role is to derive theorems based on the set of axioms provided by the programmer. The inference engine uses the operations of resolution and unification to construct proofs.

Resolution says that given the axioms

```
f if a0, ..., a_m.

g if f, b0, ..., bn.
```

the fact

```
g if a0, ..., a_m, b0, ..., bn.
```

can be derived.

Unification is the binding of variables. For example

A query containing a variable asks whether there is a value for the variable that makes the query a logical consequence of the program.

```
?- father(bill,X).
?- father(X,mary).
?- father(X,Y).
```

Note that variables do not denote a specified storage location, but denote an unspecified but single entity.

Definition: Constants and variables are terms. A compound term is comprised of a functor and a sequence of terms. A functor is characterized by its name, which is an atom and its arity, or number of arguments.

```
X, 3, mary, fatherof(F,mary), ...
```

Definition: Queries, facts and terms which do not contain variables are called ground. Where variables do occur they are called nonground.

Definition: A substitution is a finite set (possibly empty) of pairs of the form Xi=ti, where Xi is a variable and ti is a term, and Xi\neq Xj for every i \neq j, and Xi does not occur in tj, for any i and j.

```
p(a,X,t(Z)), p(Y,m,Q); theta = { X=m,Y=a,Q=t(Z) }
```

Definition: A is an instance of B if there is a substitution theta such that A = Btheta.

Definition: Two terms A and B are said to have a common instance C iff there are substitutions theta1 and theta2 such that C = Atheta1 and C = Btheta2.

```
A = plus(0,3,Y), B = plus(0,X,X). C = plus(0,3,3)
since C = A{ Y=3} and C = B{ X=3}.
```

Definition: A unifier of two terms A and B is a substitution making the two terms identical. If two terms have a unifer they are said to unify.

```
p(a,X,t(Z))theta = p(Y,m,Q)theta where theta = { X=m,Y=a,Q=t(Z) }
```

Definition: A most general unifier or mgu of two terms is a unifier such that the associated common instance is most general.

```
unify(A,B) :- unify1(A,B).

unify1(X,Y) :- X == Y.
unify1(X,Y) :- var(X), var(Y), X=Y. % The substitution
unify1(X,Y) :- var(X), nonvar(Y), \+ occurs(X,Y), X=Y. % The substitution
unify1(X,Y) :- var(Y), nonvar(X), \+ occurs(Y,X), Y=X. % The substitution
unify1(X,Y) :- nonvar(X), nonvar(Y), functor(X,F,N), functor(Y,F,N),
X =..[F|R], Y =..[F|T], match_list(R,T).

match_list([],[]).
match_list([X|R],[H|T]) :- unify(X,H), match_list(R,T).

occurs(A,B) :- A == B.
occurs(A,B) :- nonvar(B), functor(B,F,N), occurs(A,B,N).
```

```
occurs(A,B,N) :- N > 0, arg(N,B,AN), occurs(A,AN),!. % RED
occurs(A,B,M) :- M > 0, N is M - 1, occurs(A,B,N).
```

A Simple Interpreter for Pure Prolog

An interpreter for pure Prolog can be written in Prolog.

Figure M.N: A simple interpreter for pure Prolog

```
is_true( Goals ) <-- resolved( Goals ).
is_true( Goals ) <-- write( no ), nl.

resolved([]).
resolved(Goals) <-- select(Goal,Goals,RestofGoals),
                      % Goal unifies with head of some rule
                      clause(Head,Body), unify( Goal, Head ),
                      add(Body,RestofGoals,NewGoals),
                      resolved(NewGoals).

prove(true).
prove((A,B)) <-- prove(A), prove(B). % select first goal
prove(A) <-- clause(A,B), prove(B). % select only goal and find a rule
```

---

is the Prolog code for an interpreter.

The interpreter can be used as the starting point for the construction of a debugger for Prolog programs and a starting point for the construction of an inference engine for an expert system.

The operational semantics for Prolog are given in Figure~\ref{lp:opsem}

```
Logic Programming (Horn Clause Logic) -- Operational Semantics
Abstract Syntax:

   P in Programs
   C in Clauses
   Q in Queries
   T in Terms
   A in Atoms
   X in Variables

P ::= (C | Q)...
C ::= G [ <-- G_{1} [ \wedge G_{2}]...] .
G ::= A [ ( T [,T]... ) ]
T ::=  X | A [ ( T [,T]... ) ]
Q ::= G [,G]... ?

Semantic Domains:

   beta in  {\bf B} = Bindings
   \epsilon in  {\bf E} = Environment
```

```
Semantic Functions:

   R in Q} --> B} -->  B } + (B} × }yes }) + } no } }
   U in C } × C } --> B } --> B }

Semantic Equations:

R[ ?} ] beta , \epsilon &=&  (beta, yes})
R[ G ] beta , \epsilon &=&  beta'
&  where }
&&G' in \epsilon ,   U [ G, G' ] beta = beta'
R[ G ] beta , \epsilon &=& R[ B ] beta' , \epsilon &  where }
&&(G' <-- B) in \epsilon ,   U [ G, G' ] beta = beta'
R[ G1,G2 ] beta , \epsilon &=& R[ B,G2 ] ( R [ G1 ] beta , \epsilon ), \epsilon
R[ G } ] beta , \epsilon &=&  no }
&  where no other rule applies}

\caption{Operational semantics\label{lp:opsem}}
```

Declarative Semantics

The declarative semantics of logic programs is based on the standard model-theoretic semantics of first-order logic.

Definition M.N:

Let P be a logic program. The Herbrand universe of P, denoted by U(P) is the set of ground terms that can be formed from the constants and function symbols appearing in P

Definition M.N:

The Herbrand base, denoted by {\cal B}(P), is the set of all ground goals that can be formed from the predicates in P and the terms in the Herbrand universe.

The Herbrand base is infinite if the Herbrand universe is.

Definition M.N:

An interpretation for a logic program is a subset of the Herbrand base.

An interpretation assigns truth and falsity to the elements of the Herbrand base. A goal in the Herbrand base is true with respect to an interpretation if it is a member of it, false otherwise.

Definition M.N:

An interpretation I is a model for a logic program if for each ground instance of a clause in the program A <-- B1, ... , Bn A is in I if B1, ... , Bn are in I.

This approach to the semantics is often called the term model.

Denotational Semantics

Denotational semantics assignes meanings to programs based on associating with the program a function over the domain computed by the program. The meaning of the program is defined as the least fixed point 0f the function, if it exists.

Pragmatics

Logic Programming and Software Engineering

Programs are theories and computation is deduction from the theory. Thus the process of software engineering becomes:

- obtain a problem description
- define the intended model of interpretation (domains, symbols etc)
- devise a suitable theory (the logic component) suitably restricted so as to have an efficient proof procedure.
- describe the control component of the program
- use declarative debugging to isolate errors in definitions

Pros and Cons

- Pro
  - Closer to problem domain thus higher programmer productivity
  - Separation of logic and control (focuses on the logical structure of the problem rather than control of execution)
  - Simple declarative semantics and referential transparency
  - Suitable for prototyping and exploratory programming
  - Strong support for meta-programming
  - Transparent support for parallel execution
- Con
  - Operational implementation is not faithful to the declarative semantics
  - Unsuited for state based programming
  - Often inefficient

The Logical Variable

The logical variable, terms and lists are the basic data structures in logic programming.

Here is a definition of the relation between the prefix and suffixes of a list. The relation is named concat because it may be viewed as defining the result of appending two lists to get the third list.

{l} concat([ ],[ ]) concat([H|T],L,[H|TL]) <-- concat(T,L,TL)

Logical variables operate in a way much different than variables in traditional programming languages. By way of illustration, consider the following instances of the concat relation.

1. ?- concat([a,b,c],[d,e],L). L = [a, b, c, d, e] the expected use of the concat operation.
2. ?- concat([a,b,c],S,[a,b,c,d,e]). S = [d, e] the suffix of L.
3. ?- concat(P,[d,e],[a,b,c,d,e]). P = [a, b, c] the prefix of L.
4. ?- concat(P,S,[a,b,c,d,e]). P = [ ], S = [a,b,c,d,e] P = [a], S = [b,c,d,e] P = [a,b], S = [c,d,e] P = [a,b,c], S = [d,e] P = [a,b,c,d], S = [e] P = [a,b,c,d,e], S = [ ] the prefixes and sufixes of L.
5. ?- concat(_,[c|_],[a,b,c,d,e]). answers Yes since c is the first element of some suffix of L.

Thus concat gives us 5 predicates for the price of one.

```
concat(L1,L2,L)
prefix(Pre,L) <-- concat(Pre,_,L).
sufix(Suf,L) <-- concat(_,Suf,L).
split(L,Pre,Suf) <-- concat(Pre,Suf,L).
member(X,L) <-- concat(_,[X|_],L).
```

The underscore _ designates an anonymous variable, it matches anything.

There two simple types of constants, string and numeric. Arrays may be represented as a relation. For example, the two dimensional matrix

data} = \left(\begin{array}{lr} mary 18.47 john 34.6 jane 64.4 \end{array} \right)

may be written as {ll} data(1,1,mary)&data(1,2,18.47) data(2,1,john)&data(2,2,34.6) data(3,1,jane)&data(3,2,64.4)

Records may be represented as terms and the fields accessed through pattern matching.

```
book(author( last(aaby), first(anthony), mi(a)),
   title('programming language concepts),
   pub(wadsworth),
   date(1991))

book(A,T,pub(W),D)
```

Lists are written between brackets [and], so [ ] is the empty list and [b, c] is the list of two symbols b and c. If H is a symbol and T is a list then [H|T] is a list with head H and tail T. Stacks may then be represented as a list. Trees may be represented as lists of lists or as terms.

Lists may be used to simulate stacks, queues and trees. In addition, the logical variable may be used to implement incomplete data structures.

Incomplete Data Structures

The following code implements a binary search tree as an incomplete data structure. It may be used both to construct the tree by inserting items into the tree and to search the tree for a particular key and associated data.

```
lookup(Key,Data,bt(Key,Data,LT,RT))
lookup(Key,Data,bt(Key0,Data0,LT,RT)) <-- Key @< Key0,
   lookup(Key,Data,LT)
lookup(Key,Data,bt(Key0,Data0,LT,RT)) <-- Key @> Key0,
   lookup(Key,Data,RT)
```

This is a sequence of calls. Note that the initial call is with the variable BT.

```
lookup(john,46,BT), lookup(jane,35,BT), lookup(allen,49,BT), lookup(jane,Age,BT).
```

The first three calls initialize the dictionary to contain those entries while the last call extracts janes's age from the dictionary.

The logical and the incomplete data structure can be used to append lists in constant time. The programming technique is known as difference lists. The empty difference list is X/X. The concat relation for difference lists is defined as follows:

```
concat_dl(Xs/Ys, Ys/Zs, Xs/Zs)
```

Here is an example of a use of the definition.

```
?- concat_dl([1,2,3|X]/X,[4,5,6|Y]/Y,Z).

   _X = [4,5,6 | 11]
   _Y = 11
   _Z = [1,2,3,4,5,6 | 11] / 11
Yes
```

The relation between ordinary lists and difference lists is defined as follows:

```
ol_dl([ ],X/X) <-- var(X)
ol_dl([F|R],[F|DL]/Y) <-- ol_dl(R,DL/Y)
```

Arithmetic

Terms are simply patterns they may not have a value in and of themselves. For example, here is a definition of the relation between two numbers and their product.

```
times(X,Y,X×Y)
```

However, the product is a pattern rather than a value. In order to force the evaluation of an expression, a Prolog definition of the same relation would be written

```
times(X,Y,Z) <-- Z is X×Y
```

Iteration vs Recursion

Not all recursive definitions require the runtime support usually associated with recursive subprogram calls. Consider the following elegant mathematical definition of the factorial function.

```
 n! = 1 if n = 0
      n× (n-1)! if n > 0
```

Here is a direct restatement of the definition in a relational form.

```
factorial(0,1)
factorial(N,N×F) <-- factorial(N-1,F)
```

In Prolog this definition does not evaluate either of the expressions N-1 or N×F thus the value 0 will not occur. To force evaluation of the expressions we rewrite the definition as follows.

```
factorial(0,1)
factorial(N,F)<-- M is N-1, factorial(M,Fm), F is N×Fm
```

Note that in this last version, the call to the factorial predicate is not the last call on the right-hand side of the definition. When the last call on the right-hand side is a recursive call ({\it tail recursion}) then the definition is said to be an iterative definition. An iterative version of the factorial relation may be defined using an accumulator and tail recursion.

```
fac(N,F) <-- fac(N,1,F)
fac(0,F,F)
fac(N,P,F) <-- NP is N×P, M is N-1, fac(M,NP,F)
```

In this definition, there are two different fac relations, the first is a 2-ary relation, and the second is a 3-ary relation.

As a further example of the relation between recursive and iterative definitions, here is a recursive version of the relation between a list and its reverse.

```
reverse([ ],[ ])
reverse([H|T],R) <-- reverse(T,Tr), concat(Tr,[H],R)
```

and here is an iterative version.

```
rev(L,R)<-- rev(L,[ ],R)
rev([ ],R,R)
rev([H|T],L,R) <-- rev(T,[H|L],R)
```

Efficient implementation of recursion is possible when the recursion is tail recursion. Tail recursion is implementable as iteration provided no backtracking may be required (the only other predicate in the body are builtin predicates).

Backtracking

When there are multiple clauses defining a relation it is possible that either some of the clauses defining the relation are not applicable in a particular instance or that there are multiple solutions. The selection of alternate paths during the construction of a proof tree is called backtracking.

Exceptions

Logic programming provides an unusually simple method for handling exception conditions. Exceptions are handled by backtracking.

Logic Programming vs Functional Programming

Functional composition vs composition of relations, backtracking, type checking

Prolog and Logic

The Logic of Prolog

Horn Clauses

Translation of first-order predicate logic to horn clause logic:
Replace

- A <--> B with A --> B and B --> A
- A --> B with not A \/ B
- Move negations inward (from the outside inward). Replace
  - not (A and B) with not A or not B
  - not (A or B) with not A and not B
  - not Exists x. P with Forall x. not P
  - not Forall x. P with Exists x. not P

- Skolemize (replace existential variables with skolem constants or skolem fuctions of universal variables (from the outside inward). Replace
  - Exists x. P(x) with P(c) where c is new
  - Forall x. ... Exists y. P(y) with Forall x. ... P(f_c(c_k)) where f_c and c_k are new
- Move universal quantifiers outward. Replace

  ... Forall x.P(x) with Forall x. ... P(x)
  (we can just drop the quantifiers)
- Put quantifier free portion into conjunctive normal form (conjunction of disjunctions). Replace
  - (A and B) or C with (A or C) and (B or C)
  - (A and C) or (B and C) with (A or B) and C

  (move conjunctions out and disjunctions in)

Each disjunction is of the form: not A_1\/...\/not A_m\/B_1\/...\/B_n
which is equivalent to: A_1/\.../\A_m --> B_1\/...\/B_n

- If m=0 and n=1 then we have a Prolog fact.
- If m>0 and n=1 then we have a Prolog rule.
- If m>0 and n=0 then we have a Prolog query.

If n always is 1 then the logic is called Horn Clause Logic which is equivalent in computational power to the Universal Turing Machine.

Resolution and unification, forward and backward chaining

The resolution rule combines clauses when a negated and a non-negated literal match.

If Aj and B_y `match' then by resolution: ...not Ai \/ not Aj \/ B_k...
...not A_x \/ B_y \/ B_z...

...not Ai \/...\/ A_x \/ B_z...\/ B_k
Matching is called unification.

Direction of Proof

- Forward chaining: proofs proceed from facts through rules to conclusions (goals). Also called bottom-up.
- Backward chaining: proofs proceed from goals back through rules toward facts. Also called top-down and goal-directed.

The Illogic of Prolog

Prolog (for efficiency reasons) departs from the logic programming model in several ways. Prolog does not perform the ``occurs check''. Prolog is implemented as a sequential programming language by processing goals from left to right and selecting rules in textual order (depth-first search).

Prolog is not logic programming. The execution model for Prolog omits the occurs check, searches the rules sequentially, and selects goals sequentially. Backtracking, infinite search trees ...

As in functional programming, lists are an important data structure logic programming. The empty list is represented by [ ], a list of n elements by [X1, ... , Xn] and the first i elements of a list and the rest of the list by [X1,...,Xi|R]. In addition, data structures of arbitrary complexity may be constructed from the terms.

- Depth-first, left-right search instead of breadth-first parallel search means that rule and clause order can matter. Instead of combinatorial explosion in the size of the search tree, we may have infinite recursion.
- There is no `occurs check' when performing unification. This means that X unifies with f(X) -- infinite terms may be constructed during unification. Since this is an infrequent occurrence, we are trading correctness for reduction in running time.

- Negation by failure, `not'. Closed world assumption. Horn clause logic does not include the `not' operator, however its use simplifies programs.
- The `cut', prunes unnecessary branches. Encourages a `goto' style programming.

Incompleteness

Incompleteness occurs when there is a solution but it cannot be found. The depth first search of Prolog will never answer the query in the following logic program.

```
p( a, b ).
p( c, b ).
p( X, Z ) <-- p( X, Y ), p( Y, Z).
p( X, Y ) <-- p( Y, X ).
?- p( a, c ).
```

The result is an infinite loop. The first and fourth clauses imply p(b, c). The first and third clauses with the p(b, c) imply the query. Prolog gets lost in an infinite branch no matter how the clauses are ordered, how the literals in the bodies are ordered or what search rule with a fixed order for trying the clauses is used. Thus logical completeness requires a breadth-first search which is too inefficient to be practical.

Unfairness

Unfairness occurs when a permissible value cannot be found.

```
concat( [ ], L, L ).
concat( [H|L1], L2, [X|L] ) <-- concat( L1, L2, L ).
concat3( L1, L2, L3, L ) <-- concat( L1, L2, L12 ),
   concat( L12, L3 L ).
?- concat3( X, Y, [2], L).
```

Result is that X is always [ ]. Prologs depth-first search prevents it from finding other values.

Unsoundness

Unsoundness occurs when there is a successful computation of a goal which is not a logical consequence of the logic program.

```
test <-- p( X, X ).
p( Y, f( Y )).
?- test.
```

Lacking the occur check Prolog will succeed but \verb+test+ is not a logical consequence of the logic program.

The execution of this logic program results in the construction of an infinite data structure.

```
concat( [ ], L, L ).
concat( [H|L1], L2, [X|L] ) <-- concat( L1, L2, L ).
?- concat( [ ], L, [1|L] ).
```

In this instance Prolog will succeed (with some trouble printing the answer). There are two solutions, the first is to change the logic and permit infinite terms, the second is to introduce the occur check with the resulting loss of efficiency.

Negation

Negative information cannot be expressed in Horn clause logic. However, Prolog provides the negation operator not and defines

negation as failure to find a proof.

```
p( a ).
r( b ) <-- not p( Y ).
?- not p(b).
```

The goal succeeds but is not a logical consequence of the logic program.

```
q( a ) <-- r( a ).
q( a ) <-- not r( a ).
r( X ) <-- r( f( X ) ).
?- q ( a ).
```

The query is a logical consequence of the first two clauses but Prolog cannot determine that fact and enters an infinite derivation tree. However the closed world assumption is useful from a pragmatic point of view.

Control Information

Cut (!): prunes the proof tree.

```
a(1).
a(2).
a(3).
p <-- a(I),!,print(I),nl,fail.
?- p.
1

No
```

Extralogical Features

Input-output primitives cannot be fully described in first-order logic. These primitives produce input-output by side-effects.

Some other extralogical primitives include bagof, setof, assert, retract, univ. These are outside the scope of first-order logic.

Input and output introduce side effects.

The extralogical primitives \verb+bagof+, \verb+setof+, \verb+assert+, and \verb+retract+ are outside the scope of first-order logic but are useful from the pragmatic point of view.

In Prolog there are builtin predicates to test for the various syntactic types, lists, numbers, atoms, clauses. Some predicates which are commonly available are the following.

{ll} var(X)&X is a variable atomic(A)&A is an atom or a numeric constant functor(P,F,N)&P is an N-ary predicate with functor F clause(Head,Body)&Head <-- Body is a formula. L =..List, call(C), assert(C), retract(C), bagof(X,P,B), setof(X,P,B)

Figure M.N:

```
trace(Q) <-- trace1([Q])
trace1([])
trace1([true|R]) <-- !, trace1(R).
trace1([fail|R]) <-- !, print('< '), print(fail), nl, fail.
trace1([B|R]) <-- B =..[','|BL], !, concat(BL,R,NR), trace1(NR).
```

```
trace1([F|R]) <-- builtin(F),
      print('> '), print([F|R]), nl,
      F,
      trace1(R),
      print('< '), print(F), nl

trace1([F|R]) <-- clause(F,B),
      print('> '), print([F|R]),nl,
      trace1([B|R]),
      print('< '), print(F), nl

trace1([F|R]) <--  \+ builtin(F), \+ clause(F,B),
      print('> '), print([F|R]),nl,
      print('< '), print(F), print(' '), print(fail), nl, fail

\caption{Program tracer for Prolog\label{lp:trace}}
```

---

contains an example of meta programming. The code implements a facility for tracing the execution of a Prolog program. To trace a Prolog program, instead of entering {\tt ?- P.} enter {\tt ?- trace(P).}

Multidirectionality

Computation of the inverse function must be restricted for efficiency and undecidability reasons. For example consider the query {l} ?- factorial(N,5678). An implementation must either generate and test possible values for N (which is much too inefficient) or if there is no such N the undecidability of first-order logic implies that termination may not occur.

Rule Order

Rule order affects the order of search and thus the shape of the proof tree. In the following program

```
concat([ ],L,L).
concat([H|T],L,[H|R]) <-- concat(T,L,R).
?- concat(L1,[2],L).
```

the query results in the sequence of answers.

```
L1 = [ ], L = [2]
L1 = [V1], L = [V1,2]
L1 = [V1,V2], L = [V1,V2,2]
...
```

However, if the order of the rules defining $append$ are interchanged,

```
append([H|T],L,[H|R]) :- append(T,L,R).
append([ ],L,L).
?- append(L1,[2],L).
```

then the execution fails to terminate, entering an infinite loop since the first rule is always applicable.

Historical Perspectives and Further Reading

- Intuitionistic mathematics and proof theory.

- Literal normal form, conjunctive normal form and Horn Clause Logic
- Robinson's unification algorithm and the resolution principle. Two terms are said to be unifiable iff there is are substitutions which applied to each makes them the same.
- Kowalski normal form -- Kowalski
- Definite clause grammars -- Colmerauer
- Relational Data Bases -- Codd

  Relations and the Relational Algebra
  DataLog
- Prolog -- Colmerauer, Warren (David)

Logic programming languages are abstractions and generalization of tuples (relations). History

- Aristotle: (384-322 BCE) -- Theory of syllogistic
- Liebniz (1646-1716): De Arte Combinatoriua 1666 -- calculus of reasoning
- Boole: 1854 -- Boolean logic
- Frege: 1879 Begriffsschrift -- separation of logic from mathematics
- Russell, B, & Whitehead, A. N.: 1910-13 -- Logicism (reduction of mathematics to logic)
- Hilbert, David: 1900 -- Formalism (finitary proofs of consistency)
- Brouwer, L.E.J.: (1881-1966) -- Intuitionism (mathematical certitude is in intuition & explicit construction)
- Gödel, Kurt: 1933 -- incompleteness theorem
- Tarski, Alfred: 1936 -- separation of logic and models
- Church, Alonso: 1936 -- non-termination of proof algorithm for non-theorems
- Robinson, J. Alan: 1965 -- resolution principle
- Kowalski, Robert: 1974 -- predicate logic as a programming language
- Tärnlund, S-A: 1977 -- Horn clause computability
- Pereira, Fernando: -- implementation of Prolog
- Warren, David: -- implementation of Prolog, Warren abstract machine (WAM)
- Classical logic (propositional, predicate/first-order)
- Other logics: Fuzzy, non-monotonic

Future

- Improved implementations: the Gödel programming language
- Combination of logic and functional paradigms: the Escher programming language

Integration of Database management systems and logic programming and parallel programming languages based on the logic paradigm. References

Clocksin & Mellish, Programming in Prolog 4th ed. Springer-Verlag 1994.
Hill, P. & Lloyd, J. W., The Gödel Programming Language MIT Press 1994.
Hogger, C. J., Introduction to Logic Programming Academic Press 1984.
Lloyd, J. W., Foundations of Logic Programming 2nd ed. Springer-Verlag 1987.
Nerode, A. & Shore, R. A., Logic for Applications Springer-Verlag 1993.
Robinson, J. A., Logic: Form and Function North-Holland 1979.

1969 J Robinson and Resolution 1972 Alain Colmerauer History Kowalski's paper\cite{Kowalski79} Logic programming techniques Implementation of Prolog SQL DCG

Exercises

1. Modify concat to include an explicit occurs check.
2. Construct a Prolog based family database. Include the following relations: parentof, grandparentof, ancestorof, uncleof, auntof, and any others of your choice.
3. The relational algebra is ... query languages of relational database management systems is another approach to the logic model.

The fundamental entity in a relational database is a *relation* which is viewed as a table of rows and columns, where each row, called a *tuple*, is an object and each column is an *attribute* or propery of the object.

A database consists of one or more relations. The data stored in the relations is manipulated using commands written in a *query language*. The operations provided the query language include union, set difference, cartesian product, projection, and selection. The first-order predicate logic can be used to represent knowledge and as a language for expressing operations on relations. -- Ullman (Principles of Database and Knowledge-base Systems) CSP 1988.

- The tables of a relational database are represented as Prolog facts.
- The Relational algebra implemented via Prolog rules and queries.
  - Selection:

    select(variables) :- conditions on the constants.
    Constants select rows in the relation.
  - Intersection:

    r_1in_r$_2$(Vars) :- r_1(Vars), r$_2$(Vars).
    selects the entities that are in both r_1 and r$_2$(use the same variables).
  - Difference:

    diff_r_1_r$_2$(Vars) :- r_1(Vars), not r$_2$(Vars).
    selects the entities in r_1 that are not in r$_2$.
  - Projection:

    pr(variables) :- r(variables and don't cares).
    Don't cares represent columns to be deleted.
  - Cartesian product:

    prod(variables) :- r_1(vars), r$_2$(vars).
    prod variables is the list of variables both in r_1 and r$_2$.
  - Union: (two rules are required to perform union.)

    union(variables) :- first_relation(variables).
    union(variables) :- second_relation(variables).

  - Natural Join: In the rule,

    ```
    nat_join( variables shared variables ) :-
        r_1(variables, shared variables),
        r2(variables, shared variables).
    ```

  -

    the shared variables restrict search to common elements, reduced number of variables in the join eliminate multiple columns.

4. Construct a family data base f_db(f,m,c,sex) and define the following relations, f_of, m_of, son_of, dau_of, gf, gm, aunt, uncle, ancestor, half_sis, half_bro.
5. Business Data base
6. Blocks World
7. CS Degree requirements; course(dept,name,prereq). don't forget w1 and w2 requirements.
8. Circuit analysis
9. Tail recursion
10. Compiler
11. Interpreter
12. Tic-Tac-Toe
13. DCG

14. Construct Prolog analogues of the relational operators for union, set difference, cartesian product, projection and selection.
15. Airline reservation system

Objective:

- Atoms and Terms
- Relations, predicates and facts
- Queries
- Terms, logical variable, substitutions, instances
- Unification and the MGU
- Variables and Quantification
- Rules
- Inference
- Abstract Interpreter for Logic Programs
- The Meaning of Logic Programs

The logical variable, substitutions and instances

Rules

```
A :- B1, B2, ..., Bn.
```

A is the *head*. B1, B2, ..., Bn. is the body. Facts, rules, and queries are also called Horn clauses or just clauses. Facts are also called unit clauses.

```
ancestor(X,Y) :- father(X,Y).
ancestor(X,Z) :- father(X,Y), ancestor(Y,Z).
```

How to read rules. For every X and Y, if X is the father of Y and Y is an ancestor of Z, X is the ancestor of Z. For every X and Y, X is the ancestor of Z, if X is the father of Y and Y is an ancestor of Z.

```
father(bill,jim).
father(jim,jane).
?- father(bill,Y), father(Y,jane).
```

Operationally, to solve a conjunctive query, a single substitution must be found applicable to each conjunct.

```
For A1,...,An and theta
A1theta,...,Antheta each is deducible.
```

Quantifiers

Variables in queries are existentially quantified. Operationally, to answer a query, using a program, is to perform a computation whose output is the substitution that unifies the query with an instance of the query which is deducible from the program. Note that it may be possible to compute more than one substitution. Variables occurring in facts are universally quantified.

```
father(adam,X).
```

Variables in the body of a rule are read as universally quantified out side the rule and read as existentially quantified inside the

rule. For every X and Y, if X is the father of Y and Y is an ancestor of Z, X is the ancestor of Z. if there exist X and Y such that X is the father of Y and Y is an ancestor of Z, then X is the ancestor of Z.

```
father(bill,jim).
father(jim,jane).
?- father(bill,Y), father(Y,jane).
```

Operationally, to solve a conjunctive query, a single substitution must be found applicable to each conjunct.

> For A1,...,An and theta
> A1theta,...,Antheta each is deducible.

The Imperative Programming Paradigm

Imperative programming is characterized by programming with a state and commands which modify the state.
Imperative:
a command or order
Procedure:
a. *the act, method or manner of proceeding in some process or course of action*
b. *a particular course of action or way of doing something.*
When imperative programming is combined with subprograms it is called procedural *programming. In either case the implication is clear. Programs are directions or orders for performing an action.*

Keywords and phrases: Assignment, goto, structured programming, command, statement, procedure, control-flow, imperative language, assertions, axiomatic semantics. state, variables, instructions, control structures.

The imperative programming paradigm is an abstraction of real computers which in turn are based on the Turing machine and the Von Neumann machine with its registers and store (memory). At the heart of these machines is the concept of a modifiable store. Variables and assignments are the programming language analog of the modifiable store. The store is the object that is manipulated by the program. Imperative programming languages provide a variety of commands to provide structure to code and to manipulate the store. Each imperative programming language defines a particular view of hardware. These views are so distinct that it is common to speak of a Pascal machine, C machine or a Java machine. A compiler implements the virtual machine defined by the programming language in the language supported by the actual hardware and operating system.

In imperative programming, a name may be assigned to a value and later reassigned to another value. The collection of names and the associated values and the location of control in the program constitute the *state*. The state is a logical model of storage which is an association between memory locations and values. A program in execution generates a sequence of states(See Figure N.1). The transition from one state to the next is determined by assignment operations and sequencing commands.

Figure N.1: State Sequence

S_0 -- O_0 -> S_1 - ... -> S_{n-1} -- O_{n-1}-> S_n

Unless carefully written, an imperative program can only be understood in terms of its execution

behavior. The reason is that during the execution of the code, any variable may be referenced, control may be transferred to any arbitrary point, and any variable binding may be changed. Thus, the whole program may need to be examined in order to understand even a small portion of code.

Since the syntax of C, C++ and Java are similar, in what follows, comments made about C apply also to C++ and Java.

Variables and Assignment

Imperative programs are characterized by sequences of bindings (state changes) in which a name may be bound to a value at one point in the program and later bound to a different value. Since the order of the bindings affects the value of expressions, an important issue is the proper sequencing of bindings.

> **Terminology.**When speaking of hardware we use terms like *bit pattern*, *storage cell,*and *storage address.*Somewhat analogous terms in programming languages are *value,variable,*and *name*. Since variables are usually bound to a name and to a value, the word variable is often used to mean the name of a value.
>
> **Aside.** Most descriptions of imperative programming languages are tied to hardware and implementation considerations where a name is bound to an address, a variable to a storage cell, and a value to a bit pattern. Thus, a name is tied to two bindings, a binding to a location and to a value. The location is called the **l-value** and the value is called the **r-value**. The necessity for this distinction follows from the implementation of the assignment. For example,
>
> ```
> X := X+2
> ```
>
> the X on the **l**eft of the assignment denotes a location while the X on the **r**ight hand side denotes the value. Assignment changes the value at a location.
>
> A variable may be bound to a hardware location at various times. It may be bound at compile time (rarely), at load time (for languages with static allocation) or at run time (for languages with dynamic allocation). From the implementation point of view, variable declarations are used to determine the amount of storage required by the program.

The following examples illustrate the general form for variable declarations in imperative programming languages.

Pascal style declaration: var *name* : *Type;*

C style declaration: *Type name;*

A variable and a value are **bound** by an assignment. A variety of notations is used to indicate the binding of a variable *V* and the value of an expression *E*.

Pascal V := E

C	V = E
APL	V <-- E
Scheme	(setq V E)

> **Aside.** The use of the assignment symbol, =, in C confuses the distinction between definition, equality and assignment. The equal symbol, =, is used in mathematics in two distinct ways. It is used to define and to assert the equality between two values. In C it neither means define nor equality but assign. In C the double equality symbol, ==, is used for equality, while the form: *type variable*`;` is used for definitions.

The **assignment** command is what distinguishes imperative programming languages from other programming languages. The assignment typically has the form:

V `:=` *E*.

The command is read ``assign the name *V* to the value of the expression *E* until the name *V i*s reassigned to another value". The assignment binds a name and a value.

> **Aside.** The word ``assign" is used in accordance with its English meaning; a name is assigned to an object, not the reverse. The name then stands for the object. The name is the assignee. This is in contrast to wide spread programming usage in which a value assigned to a variable.

The assignment is not the same as a constant definition because it permits redefinition. For example, the two assignments:

```
X := 3;
X := X + 1
```

are understood as follows: assign `X` to three and then reassign `X` to the value of the expression `X+1` which is four. Thus, after the sequence of assignments, the value of `X` is four.

Several kinds of assignments are possible. Because of the frequent occurrence of assignments of the form: `X := X` *`op`* `E,` C provides an alternative notation of the form: `X` *`op`*`= E`. A *multiple assignment* of the form:

$$V_0 := V_1 := \ldots := V_n := E$$

causes several names to be assigned to the same value. This form of the assignment is found in C. A *simultaneous assignment* of the form:

$$V_0,\ldots,V_n := E_0,\ldots,E_n$$ c

causes several assignments of names to values to occur simultaneously. The simultaneous assignment

permits the swapping of values without the explicit use of an auxiliary variable.

From the point of view of axiomatic semantics, the assignment is a predicate transformer. It is a function from predicates to predicates. From the point of view of denotational semantics, the assignment is a function from states to states. From the point of view of operational semantics, the assignment changes the state of an abstract machine.

Unstructured Commands

Given the importance of sequence control, it is not surprising that considerable effort has been given to finding appropriate control structures. Figure N.M gives a minimal set of basic control structures.

Figure N.M: A set of unstructured commands

command ::= *identifier* `:=` *expression*
| *command* `;` *command*
| *label* `:` *command*
| `GOTO` *label*
| `IF` *boo_exp* `THEN GOTO` *label*

The unstructured commands contain the assignment command, sequential composition of commands, a provision to identify a command with a label, and unconditional and conditional GOTO commands. The unstructured commands have the advantage, they have direct hardware support and are completely general purpose. However, the programs are flat without hierarchical structure with the result that the code may be difficult to read and understand. The set of unstructured commands contains one of the most powerful commands, the GOTO. It is also the most criticized. The GOTO can make it difficult to understand a program by producing `spaghetti' like code. So named because the control seems to wander around in the code like strands of spaghetti.

The GOTO commands are explicit transfer of control from one point in a program to another program point. These *jump* commads come in unconditional and conditional forms:

goto *label*
if *conditional expression* goto *label*

At the machine level alternation and iteration may be implemented using **labels** and **goto** commands. Goto commands often take two forms:

1. *Unconditional* ***goto***. The unconditional goto command has the form:

   goto $LABEL_i$

   The sequence of instructions next executed begin with the command labeled with `LABEL`$_i$.

2. *Conditional* ***goto***. The conditional goto command has the form:

   if *conditional expression* then goto $LABEL_i$

   If the conditional expression is true then execution transfers to the sequence of commands headed by the command labeled with `LABEL`$_i$ otherwise it continues with the command following the conditional goto.

Structured Programming

The term *structured programming* was coined to describe a style of programming that emphasizes hierarchical program structures in which each command has one entry point and one exit point. The goal of structured programming is to provide control structures that make it easier to reason about imperative programs. Figure M.N gives a minimal set of structured commands.

Figure N.M: A set of structured commands

```
command         ::= SKIP
                  | identifier := expression
                  | IF  guarded_command [ []guarded_command ]+ FI
                  | DO  guarded_command [ []guarded_command ]+ OD
                  | command ;command
guarded_command ::= guard --> command
guard           ::= boolean expression
```

The `IF` and `DO` commands which are defined in terms of guarded commands require some explanation. The IF command allows for a choice between alternatives while the DO command provides for iteration. In their simplest forms, an IF statement corresponds to an If condition then command and a DO statement corresponds to a While condition Do command.

IF *guard* --> *command* FI = if *guard* then *command*

DO *guard* --> *command* OD = while *guard* do *command*

A command proceded by a guard can only be executed if the guard is true. In the general case, the semantics of the IF - FI and DO - OD commands requires that only one command corresponding to a guard that is true be selected for execution. The selection is nondeterministic..

Control structures are syntactic structures that define the order in which assignments are performed. Imperative programming languages provide a rich assortment of sequence control mechanisms. Three control structures are found in traditional imperative langauges: *sequential composition*, *alternation*, and *iteration*.

> **Aside.** Imperative programming languages often call assignments and control structures *commands*, *statements*or *instructions*. In ordinary English, a statement is an expression of some fact or idea and thus is an inappropriate designation. Commands and instructions refer to an action to be performed by a computer. Lacking a more neutral term we will use command to refer to assignment, skip, and control structures.

Sequential Composition. Sequential composition specifies a linear ordering for the execution of commands. It is usually indicated by placing commands in textual sequence and either line separation or a special symbol (such as the semicolon) is used to indicate termination point of a command. In C the semicolon is used as a terminator, in Pascal it is a command separator. At a more abstract level, composition of commands is indicated by using a composition operator such as the semicolon (C_0;C_1).

Selection: Selection permits the specification of a sequence of commands by cases. The selection of a particular sequence is based on the value of an expression. The **if** and **case** commands are the most common representatives of alternation.

Iteration: Iteration specifies that a sequence of commands may be executed zero or more times. At run time the sequence is repeatedly composed with itself. There is an expression whose value at run time determines the number of compositions. The **while**, **repeat** and **for** commands are the most common representatives of iteration.

Abstraction: A sequence of commands may be named and the name used to invoke the sequence of commands. Subprograms, procedures, and functions are the most common representatives of abstration.

Skips

The simplest kind of command is the **skip** command. It has no effect.

Composition

The most common sequence is the **sequential** composition of two or (more) commands (often written `S_0;S_1'). Sequential composition is available in every imperative programming language.

Alternation

An **alternative command** may contain a number of alternative sequences of commands, from which exactly one is chosen to be executed. The nondeterministic IF-FI command is unusual. Traditional programming languages usually have one or more if commands and a case command.

```
-- Ada
if condition then
   commands
{ elsif condition then
   commands }
[ else
   commands]
endif
```

```
case expression is
when choice | choice  => commands
when choice | choice  => commands
[when others => commands]
end case;
```

Iteration

An **iterative command** has a body which is to be executed repeatedly and has an expression which determines when the execution will cease. The three common forms are the while-do, the repeat-until, and the for-do.

while-do while *condition* do
body

repeat-until repeat
body
until *condition*

for-do for *index* := *lowerBound*, *upperBound*, *step* do
body

The while-do command semantics require the testing of the condition before the body is executed. The semantics of the repeat-until command require the testing of the condition after the body is executed. The for-do command semantics require testing of the condition before the body is executed.

The iterative commands are often used to traverse the elements of a data structure - search for a item etc. This insight leads to the concept of generators and iterators.

> **Definition:** A *generator*is an expression which generates a sequence of values contained in a data structure.

The generator concept appears in functional programming languages as *functionals*.

> **Definition:** An *iterator* is a generalized looping structure whose iterations are determined by a generator.

An iterator is used with the an extended form of the **for** loop where the iterator replaces the initial and final values of the loop index. For example, given a binary search tree and a generator which performs inorder tree traversal, an iterator would iterate for each item in the tree following the inorder tree traversal.

FOR Item in Tree DO S;

Sequential Expressions

Imperative programming languages with their emphasis on the sequential evaluation of commands often fail to provide a similar sequentiality to the evaluation of expressions. The following code illustrates a common programming situation where there are two or more conditions which must remain true for iteration to occur.

```
i := 0;
while (i < length) and (list[i] <> value) do
   i := i+1
```

The code implements a sequential search for a value in a table and terminates when either the entire table has been searched or the value is found. Assuming that the subscript range for `list` is 0 to `length` it seems reasonable that the termination of the loop should occur either when the index is out of bounds or when the value is found. That is, the arguments to the `and` should be evaluated sequentually and if the first argument is false the remaining argument need not be evaluated since the value of the expression cannot be true. Such an evaluation scheme is call *short-circuit* evaluation. In languages without short-circuit evaluation, if the value is not in the list, the program aborts with a subscript out of range error.

The Ada language provides the special operators `and then` and `or else` so that the programmer can specify short-circuit evaluation.

Subprograms, procedures, and functions

A procedure is an abstraction of a sequence of commands. The **procedure call** is a reference to the abstraction. The syntax of procedure definition and invocation (call) is simple.

Procedure definition: *name*(*parameter list*) { *body* }

Procedure invocation: *name*(*argument list*)

The semantics of the procedure call is determined by the semantics of the procedure body. For many languages with non-recursive procedures, the semantics may be viewed as simple textual substitution.

> **Terminology:** Parameters are often called *formal parameters*and arugments are often called *actual parameters.*

Parameters and arguments have a simple syntax

Parameter list: t_0 $name_1$, ..., t_{n-1} $name_{n-1}$

Argument list: $expression_1$, ..., $expression_{n-1}$

An **in** parameter designates that the body of the procedure may not modify the value of the argument (often implemented as a copy of the argument). An **out** parameter designates that value of the argument is undefined on entry to the procedure and when the procedure terminates, the argument is assigned to a value (often copied to the argument). An **in-out** parameter designates that the value of the parameter may be defined on entry to the procedure and may be modified when the procedure terminates.

		Parameter	**Argument**
Pascal	in:	*name* : *type*	*name*
	in-out:	var *name* : *type*	*expression*
Ada	in	*name* : in *type*	*expression*
	out	*name* : out *type*	*name*
	in-out	*name* : in out *type*	*name*
C	in:	*type name*	*expression*
	in-out:	*type* **name* (internal reference to the in-out parameter must be **name*)	&*name*

Other Control Structures

Other control structures are possible. In simulations, it is often easier to structure a program as a set cooperating tasks. When the task activities are interdependent, they can be structured as a collection of c*oroutines.* Unlike subroutines where control is passed back to the calling routine from the subroutine when the subroutine is finished, control is passed back and forth between the subroutines

with control resuming where it left off (right after the resume commands in the following).

Coroutine C1	Coroutine C2	Coroutine C3
...	...	...
resume C2	resume C3	resume C1
...	...	...

There is a single thread of control that moves from coroutine to coroutine. The multiple calls to a coroutine do not necessarily require multiple activation records.

In addition to coroutines there are concurrent or parallel processes

$$[|| \text{Process}_o, \ldots, \text{Process}_{n-1}]$$

with multiple threads of control which communicate either through shared variables or message passing. Concurrency and parallel programming languages are considered in a later chapter.

Reasoning about Imperative Programs

Imperative constructs jeopardize many of the fundamental techniques for reasoning about mathematical objects. For example, the assignment axiom of axiomatic semantics is valid only for languages without aliasing and side effects. Much of the work on the theory of programming languages is an attempt to explain the ``referentially opaque'' features of programming languages in terms of well-defined mathematical constructs. By providing descriptions of programming language features in terms of standard mathematical concepts, programming language theory makes it possible to manipulate programs and reason about them using precise and rigorous techniques. Unfortunately, the resulting descriptions are complex and the notational machinery is difficult to use in all but small examples. It is this complexity that provides a strong motivativation to provide functional and logic programming as alternatives to the imperative programming paradigm.

Sequencers

There are several common features of imperative programming languages that tend to make reasoning about the program difficult. The **goto** command \cite{Dijk68} breaks the sequential continuity of the program. When the use of the goto command is undisciplined, the breaks involve abrupt shifts of context.

In Ada, the exit sequencer terminates an enclosing loop. All enclosing loops upto and including the named loop are exited and execution follows with the command following the named loop.

Ada uses the **return** sequencer to terminate the execution of the body of a procedure or function and in the case of a function, to return the result of the computation.

Exception handers are sequencers that take control when an exception is raised.

A sequencer is a construct that allows more general control flows to be programmed.

- Jumps
- Exits
- Exceptions -- propagation, raising, resumption, handler (implicit invocation)
- Coroutines

The machine language of a typical computer includes instructions which allow any instruction to be selected as the next instruction. A *sequencer* is a construct that is provided to give high-level programming languages some of this flexibility. We consider three sequencers, jumps, escapes, and exceptions. The most powerful sequencer (the *goto*) is also the most criticized. Sequencers can make it difficult to understand a program by producing `spaghetti' like code. So named because the control seems to wander around in the code like the strands of spaghetti.

Escape

An *escape* is a command which terminates the execution of a textually enclosing construct. An escape of the form:

return expr

is used in C to exit a function call and return the value computed by the function.

An escape of the form:

exit(n)

is used to exit `n` enclosing constructs. The exit command can be used in conjunction with a general loop command to produce `while` and `repeat` as well as more general looping constructs.

In C a `break` command sends control out of the enclosing loop to the command following the loop while the `continue` command transfers control to the beginning of the enclosing loop.

Exceptions

There are many ``exception'' conditions that can arise in program execution. Some exception conditions are normal for example, the end of an input file marks the end of the input phase of a program. Other exception conditions are genuine errors for example, division by zero. Exception handlers of various forms can be found in PL/1, ML, CLU, Ada, Scheme and other languages.

There are two basic types of exceptions which arise during program execution. They are domain failure, and range failure.

Domain failure
: occurs when the input parameters of an operation do not satisfy the requirements of the

operation. For example, end of file on a read instruction, division by zero.

Range failure
: occurs when an operation is unable to produce a result for values which are in the range. For example, division by numbers within an epsilon of zero.

Definition: An *exception condition* is a condition that prevents the completion of an operation. The recognition of the exception is called *raising* the exception.

Once an exception is raised it must be handled. Handling exceptions is important for the construction of *robust* programs. A program is said to be *robust* if it recovers from exceptional conditions.

Definition: The action to resolve the exception is called *handling* the exception. The *propagation* of an exception is the passing of the exception to the context where it can be handled.

The simplest method of handling exceptions is to ignore it and continue execution with the next instruction. This prevents programmer from learning about the exception and may lead to erroneous results.

The most common method of handling exceptions is to abort execution. This is not exceptable for file I/O but may be acceptable for an array index being out of bounds or for division by zero.

The next level of error handling is to return a value outside the range of the operation. This could be a global variable, a result parameter or a function result. This approach requires explicit checking by the programmer for the error values. For example, the *eof* boolean is set to true when the program has read the last item in a file. The *eof* condition can then be checked before attempting to read from a file. The disadvantage of this approach is that a program tends to get cluttered with code to test the results. A more serious consequence is that a programmer may forget to include a test with the result that the exception is ignored.

Responses to an Exception

Return a label and execute a goto -- Fortran

Issues

Resumption of Program Execution

Once an exception has been detected, control is passed to the handler that defines the action to be taken when the exception is raised. The question remains, what happens after handling the exception?

One approach is to treat exception handlers as subroutines to which control is passed and after the execution of the handler control returns to the point following the call to the handler. This is the approach taken in PL/1. It implies that the handler ``fixed'' the state that raised the condition.

Another approach is that the exception handler's function is to provide a clean-up operation prior to termination. This is the approach taken in Ada. The unit in which the exception occurred terminates and control passes to the calling unit. Exceptions are propagated until an exception handler is found.

Suppression of the Exception

Some exceptions are inefficient to implement (for example, run time range checks on array bounds). The such exceptions are usually implemented in software and may require considerable implementation overhead. Some languages give the programmer control over whether such checks and the raising of the corresponding exception will be performed. This permits the checks to be turned on during program development and testing and then turned off for normal execution.

1. Handler Specification
2. Default Handlers

   Propagation of Exception

Side effects

Side effects are a feature of imperative programming languages that make reasoning about the program difficult. Side effects are used to provide communication among program units. When undisciplined access to global variables is permitted, the program becomes difficult to understand. The entire program must be scanned to determine which program units access and modify the global variables since the call command does not reveal what variables may be affected by the call.

At the root of differences between mathematical notations and imperative programs is the notion of *referential transparency* (substitutivity of equals for equals). Manipulation of formulas in algebra, arithmetic, and logic rely on the principle of referential transparency. Imperative programming languages violate the principle. For example:

```
integer f(x:integer)
  {
     y := y+1;
     f := y + x
   }
```

This ``function'' in addition to computing a value also changes the value of the global variable `y`. This change to a global variable is called a *side effect*. In addition to modifying a global variable, it is difficult to reason with the function itself. For example, if at some point in the program it is known that `y = z = 0` then `f(z) = 1` in the sense that after the call `f(z)` will return 1. But, should the following expression occur at that point in the program, it will be false.

$$1 + f(z) = f(z) + f(z)$$

I/O functions of necessity involve side effects. The following expressions involving the C function

getint may return different values even though algebraically they appear to have the same value.

```
2 * getint()
getint() + getint()
```

The first multiplies the next integer read from the input file by two while the second expression denotes the sum of the next two successive integers read from the input file.

Aliasing and dangling references

Two names are *aliases* if they denote (*share*) the same data object during a unit activation. Aliasing is another feature of imperative programming languages that makes programs harder to understand and harder to reason about.

One way aliases occur is when two or more arguments to a subprogram are the same. When a data object is passed by ``reference'' it is referenced both by its name in the calling environment and its parameter name in the called environment. In the following subprogram, the parameters are in-out parameters.

```
aliasingExample (m, n : in out integer);
   {
      n := 1;
      n := m + n
   }
```

The two parameters are used as different names for the same object in the call `aliasingExample( i, i)`. In this example, the result is that `i` is set to 2. In the call `aliasingExample( a[i], a[j] )` the result depends on the values of `i` and `j` with aliasing occuring when they are equal. This second call illustrates that aliasing can occur at run time so the detection of aliasing may be delayed until run time and so compilers cannot be relied on to detect aliasing.

Aliasing interferes with optimizing phase of a compiler. Optimization sometimes requires the reordering of steps or the deletion of unnecessary steps. The following assignments which appear to be independent of each other illustrate an order depencency.

```
x := a + b
y := c + d
```

If `x` and `c` are aliases for the same object, the assignments are interdependent and the order of evaluation is important.

The purpose of the equivalence command in FORTRAN is the creation of aliases. It permits the efficient use of memory (historically a scarce commodity) and can be used as a crude form of a variant record. Another way in which aliasing can occur is when a data object may be a component of several data objects (referenced through pointer linkages).

- Formal and actual parameters share the same data object.
- Procedure calls have overlapping actual parameters.
- A formal parameter and a global variable denote the same data object.

Pointers are intrinsically generators of aliasing.

```
var p, q : ^T;
...
new(p);
q := p
```

When a programming language requires programmers to manage memory for dynamically allocated objects and the language permits aliasing, an object returned to memory may still be accessible though an alias and the value may be changed if the memory manager allocates the same storage area to another object. In the following code,

```
type pointer = ^Integer
var p : Pointer;

procedure Dangling;
var q : Pointer;
begin;
   new(q);  q^ := 23;  p := q;  dispose(q)
end;

begin
   new(p); Dangling(p)
end;
```

the pointer p is left pointing to a non-existent value.

The problem of aliasing arises as soon as a language supports variables and assignment. If more than one assignment is permitted on the same variable *x*, the fact that *x=a* cannot be used at any other point in the program to infer a property of *x* from a property of *a*. Aliasing and global variables only magnify the issue.

Historical Perspectives and Further Reading

Imperative languages have a rich and varied history. The first imperative programming languages were machine languages. Machine instructions were soon replaced with assembly languages which

are essentially transliterations of machine code.

Early Imperative Languages

FORTRAN (FORmula TRANslation) was the first high level language to gain wide acceptance. It was designed for scientific applications and featured an algebraic notation, types, subprograms, and formatted input/output. It was implemented in 1956 by John Backus at IBM specifically for the IBM 704 machine. Efficient execution was a major concern consequently, its structure and commands have much in common with assembly languages. FORTRAN won wide acceptance and continues to be in wide use in the scientific computing community.

COBOL (COmmon Business Oriented Language) was designed (by a committee of representatives of computer manufactures and the Department of Defense) at the initiative of the U. S. Department of Defense in 1959 and implemented in 1960 to meet the need for business data processing applications. COBOL featured records, files and fixed decimal data. It also provided a ``natural language'' like syntax so that programs would be able to be read and understood by non-programmers. COBOL won wide acceptance in the business data processing community and continues to be in wide use.

ALGOL 60 (ALGorithmic Oriented Language) was designed in 1960 by an international committee for use in scientific problem solving. Unlike FORTRAN it was designed independently of an implementation, a choice which lead to an elegant language. The description of ALGOL 60 introduced the BNF notation for the definition of syntax and is a model of clarity and completeness. Although ALGOL 60 failed to win wide acceptance, it introduced block structure, structured control statements and recursive procedures into the imperative programming paradigm.

Evolutionary Developments

PL/I (Programming Language I) was developed at IBM in the mid 1960s. It was designed as a general purpose language to replace the specific purpose languages like FORTRAN, ALGOL 60, COBOL, LISP, and APL (APL and LISP were considered in the chapter on functional programming. PL/I incorporated block structure, structured control statements, and recursion from ALGOL 60, subprograms and formatted input/output from FORTRAN, file manipulation and the record structure from COBOL, dynamic storage allocation and linked structures from LISP, and some array operations from APL. PL/I introduced exception handling and multitasking for concurrent programming. PL/I was complex, difficult to learn, and difficult to implement. For these and other reasons PL/I failed to win wide acceptance.

ALGOL 68 was designed to be a general purpose language which remedied PL/I's defects by using a small number of constructs and rules for combining any of the constructs with predictable results--orthogonality. The description of ALGOL 68 issued in 1969 was difficult to understand since it introduced a notation and terminology that was foreign to the computing community. ALGOL 68 introduced orthogonality and data extensibility as a way to produce a compact but powerful language. The ``ALGOL 68 Report'' considered to be one of the most unreadable documents ever printed and implementation difficulties prevented ALGOL 68's acceptance.

Pascal was developed by Nicklaus Wirth partly as a reaction to the problems encountered with ALGOL 68 and as an attempt to provide a small and efficient implementation of a language suitable for teaching good programming style. C, which was developed about the same time, was an attempt to provide an efficient language for systems programming.

Modula-2 and Ada extended Pascal with features to support module based program development and abstract data types. Ada was developed as the result of a Department of Defense initiative while Modula-2 was developed by Nicklaus Wirth. Like PL/1 and Algol-68, Ada represents an attempt to produce a complete language representing the full range of programming tasks.

Simula 67 added coroutines and classes to ALGOL 60 to provide a language more suited to solving simulation problems. The concept of classes in object-oriented programming can be traced back to Simula's classes. Small-talk combined classes, inheritance, and ease of use to provide an integrated object-oriented development environment. C++ is an object-oriented programming language derived from C. Java, a simplified C++, is an object-oriented languages designed to dynamically load modules at runtime and to reduce programming errors.

Expression-oriented languages

Expression-oriented languages achieve simplicity and regularity by eliminating the distinction between expressions and commands. This permits a simplification in the syntax in that the language does not need both procedure and function abstractions nor conditional expressions and commands since the evaluation of an expression may both yield a value and have a side effect of updating variables. Since the assignment expression V `:=` E can be defined to both yield the value of the expression E and assign V to the value of E, expressions of the form V_0 `:= ... :=` V_n `:=` E are possible giving multiple assignment for free. Algol-68, C, Scheme, and ML are examples of expression oriented languages.

Exercises

1. [Time/Difficulty](section)Give all possible forms of assignment found in the programming language C.
2. Give axiomatic, denotational and operational semantics for the simultaneous assignment command.
3. Discuss the relationship between the assignment command and input and output commands.
4. Give axiomatic, denotational and operational semantics for the `goto` command.
5. Find an algorithm which transforms a program containing `goto`s into an equivalent program without `goto`s.
6. Give axiomatic, denotational and operational semantics for the `skip` command.
7. What is used to indicate sequential composition in
    a. the Pascal family of languages?
    b. the C family of languages?
8. Show how to implement the if-then-else command using unstructured commands.
9. Show how to implement a structured while-do and if-then-else commands using unstructured

commands.

10. Show that case and if commands are equivalent by implementing a case command using if commands and an if command using a case command.
11. Compare and contrast the if and case/switch commands of Ada and C.
12. Compare and contrast the looping commands of Ada and C.
13. Show how to implement the repeat-until and for-do commands in terms of a while-do command.
14. Show that while and repeat until control structures are equivalent.
15. Design a generalized looping command and give its axiomatic semantics.
16. Give axiomatic semantics for the IF-FI and DO-OD commands.
17. Give axiomatic semantics for the C/C++/Java `for` command.
18. Provide recursive definitions of the iterative control structures while, repeat, and for.
19. Alternative control structures
20. What is the effect on the semantic descriptions if expressions are permitted to have side effects?
21. Axiomatic semantics
22. Denotational semantics
23. Operational semantics
24. Classify the following common error/exception conditions as either domain or range errors.
    a. overflow -- value exceeds representational capabilities
    b. undefined value -- variable value is undefined
    c. subscript error -- array subscript out of range
    d. end of input -- attempt to read past end of input file
    e. data error -- data of the wrong type

Object-Oriented Programming

Object-oriented programming is characterized by programming with objects, messages, and hierarchies of objects.

The surest way to improve programming productivity is so obvious that many programmers miss it. Simply write less code.
-- Samuel P. Harbison

Keywords and phrases: Abstract Data Type, object-based, object-oriented, Inheritance, Object, sub-type, super-type, sub-range, sub-class, super-class, polymorphism, overloading, dynamic type checking, Class, Instance, method, message

---

Object-oriented programming shifts the emphasis from data as passive elements defined by relations or acted on by functions and procedures to active elements interacting with their environment. In the context of imperative programming, the emphasis shifts from describing control flow to describing interacting objects.

Object-oriented programming developed out of simulation programs. The conceptual model used is that the structure of the simulation should reflect the environement that is being simulated. For example, if an industrial process is to be simulated, then there should be an object for each entity involved in the process. The objects interact by sending messages.

Each object is designed around a data invariant.

Object-oriented programming is an abstraction and generalization of imperative programming. Imperative programming involves a state and a set of operations to transform the state. Object-oriented programming involves collections of objects each with a state and a set of operations to transform the state. Thus, object-oriented programming focuses on data rather than on control. As in the real world, objects interact so object-oriented programming uses the metaphor of message passing to capture the interaction of objects.

Functional objects are like values, imperative objects are like variables, active objects are like processes.

Aternatively, OOP, an object is a parameter (function and logic), an object is a mutable self (imperative).

Programming in an imperative programming language requires the programmer to think in terms of

data structures and algorithms for manipulating the data structure. That is, data is placed in a data structure and the data structure is manipulated by various procedures.

Programming in an object-oriented language requires the programmer to think in terms of a hierarchy of objects and the properties possessed by the objects. The emphasis is on generality and reusability.

Procedures and functions are the focus of programming in an imperative language. Object-oriented programming focuses on data, the objects and the operations required to satisfy a particular task.

Object-oriented programming, as typified by the Small-talk model, views the programming task as dealing with objects which interact by sending messages to each other. Concurrency is not necessarily implied by this model and destructive assignment is provided. In particular, to the notion of an abstract data type, OOP adds the notion of inheritance, a hierarchy of relationships among types. The idea of data is generalized from simple items in a domain to data type (a domain and associated operations) to an abstract data type (the addition of information hiding) to OOP \& inheritance.

Here are some definitions to enable us to speak the language of object-oriented programming.

- Object: Collection of private data and public operations.
- Class: Description of a set of objects. (encapsulated type: partitioned into private and public)
- Instance: An instance of a class is an object of that class.
- Method: A procedure body implementing an operation.
- Message: A procedure call. Request to execute a method.
- Inheritance: Extension of previously defined class. Single inheritance, multiple inheritance
- Subtype principle: a subtype can appear wherever an object of a supertype is expected.

I think a classification which helps is to classify languages as object-based and object-oriented. A report we recently prepared on OO technology trends reported that object-based languages support to varying degrees: object-based modularity, data abstraction (ADTs) encapsulation and garbage collection. Object-oriented languages additionally include to varying degrees: grouping objects into classes, relating those classes by inheritance, polymorphism and dynamic binding, and genericity.

Dr. Bertrand Meyer in his book 'Object-oriented Software Construction' (Prentice Hall) gives his 'seven steps to object-based (oriented) happiness'

1. Object-based modular structure
2. Data abstraction
3. Automatic memory management
4. Classes
5. Inheritance
6. Polymorphism and Dynamic Binding
7. Multiple and Repeated Inheritance

Subtypes (subranges)

The subtype principle states that a subtype may appear wherever an element of the super type is expected.

Objects

Objects are collections of operations that share a state. The operations determine the messages (calls) to which the object can respond, while the shared state is hidden from the outside world and is accessible only to the object's operations. Variables representing the internal state of an object are called *instance variables* and its operations are called *methods*. Its collection of methods determines its *interface* and its *behavior*.

Objects which are collections of functions but which do not have a state are functional objects. Functional objects are like values, they have the object-like interface but no identity that persists between changes of state. Functional objects arise in logic and functional programming languages.

Syntactically, a functional object can be represented as:

```
name : object
   methods
   ...
```

For example,

Objects which have an updateable state are imperative objects. Imperative objects are like variables. They are the objects of Simula, Smalltalk and C++. They have a name, a collection of methods which are activated by the receipt of messages from other objects, and instance variables which are shared by the methods of the object but inaccessible to other objects.

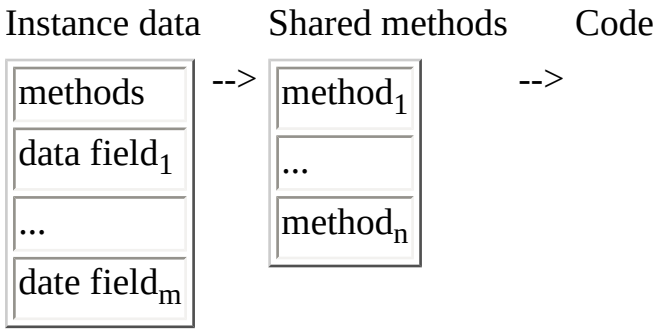
Syntactically, an imperative object can be represented as:

```
name : object
   variables
   ...

   methods
   ...
```

Objects which may be active when a message arrives are active objects. In contrast, functional and imperative objects are passive unless activated by a message. Active objects have three modes: when

there is nothing to do the object is *dormant*, when the agent is executing it is *active*, and when an object is waiting for a resource or the completion of subtasks it is *waiting*. Messages sent to an active object may have to wait in queue until the object finishes a task. Message passing among objects may be synchronous or asynchronous.

Figure M.N: Object implementation



Classes

Classes serve as templates from which objects can be created. Classes have the same instance variables and operations as corresponding objects but their interpretation is different. Instance variables in an object represent *actual* variables while class instance variables are *potential*, being instantiated only when an object is created.

We may think of a class as specifying a behavior common to all objects of the class. The instance variables specify a structure (data structure) for realizing the behavior. The public operations of a class determine its behavior while the private instance variables determine its structure.

Private copies of a class can be created by a make-instance operation, which creates a copy of the class instance variables that may be acted on by the class operations.

Syntactically, a class can be represented as:

```
name : class

   instance variables
   ...

   class variables
   ...

   instance methods
```

```
...
class methods
...
```

Classes specify the behavior common to all elements of the class. The operations of a class determine the behavior while the instance variables determine the structure.

Algebraic semantics

Many sorted algebras may be used to model classes.

Inheritance

Inheritance allows us to reuse the behavior of a class in the definition of new classes. Subclasses of a class inherit the operations of their parent class and may add new operations and new instance variables.

Inheritance captures a form of abstraction called *super-abstraction*, that complements data abstraction. Inheritance can express relations among behaviors such as classification, specialization, generalization, approximation, and evolution.

Inheritance classifies classes in much the way classes classify values. The ability to classify classes provides greater classification power and conceptual modeling power. Classification of classes may be referred to as second-order classification. Inheritance provides second-order sharing, management, and manipulation of behavior that complements first-order management of objects by classes.

Syntactically, inheritance may be specified in a class as:

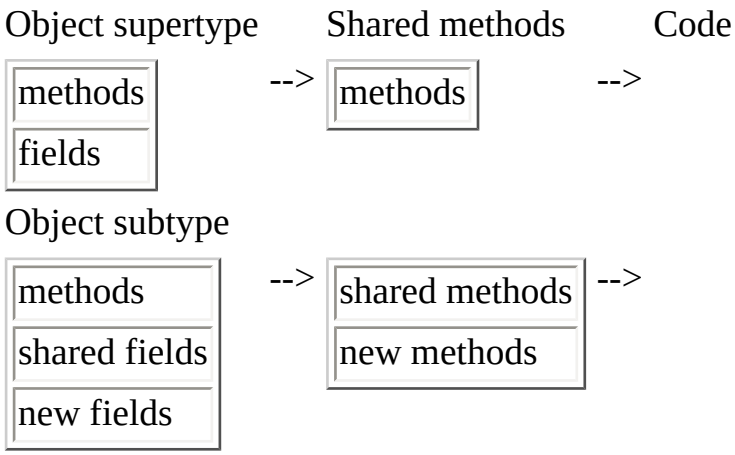
```
name : class

   super class
   ...
   instance variables
   { as before }
```

What should be inherited? Should it be behavior or code: specification or implementation? Behavior and code hierarchies are rarely compatible with each other and are often negatively correlated because shared behavior and shared code have different requirements.

```
Representation, Behavior, Code

DYNAMIC/STATIC/INHERITANCE

Inheritance and OOP
```

```
Type hierarchy

Semantics of inheritance in the functional paradigm.


     type op params = case op of
        f0 : f0 params
     ...
        fn : fn params
        otherwise : supertype op params
        where
        f0 params = def0
     ...
        fn params = defn



inheritance in the logic programming paradigm.


object(structure,methodslist).

isa(type1,type2).

object(rectangle(Length,Width),[area(A is Length*Width)]).
```

Figure M.N: Implementation of inheritance



Algebraic semantics

Order-sorted algebras are required to capture the ordering relations among sorts that arise in subtypes and inheritance.

Types and Classes

The concept of a type and the concept of a class have much in common and depending on the point of view, they may be indistinguishable. The distinction between types and classes may be seen when we examine the compare the inheritance relationship between types and subtypes with the inheritance relationship between classes and subclasses.

Example: The natural numbers are a subtype of the integers but while subtraction is defined for all pairs of integers it is not defined for all pairs of natural numbers.

This is an example of subtype inheritance. Subtypes are defined by additional constraints imposed on a type. The set of values satisfying a subtype is a subset of the set of values satisfying the type and subtypes inherit a subset of the behaviors of the type.

Example: The integers are a subclass of the natural numbers since, the subtraction operation of the natural numbers can be extended to subtraction for integers.

Example: The rational numbers are a subclass of the integers since, they can be defined as pairs of natural numbers and the arithmetic operations on the rational numbers can be defined in terms of the arithmetic operations on natural numbers.

These are examples of subclass inheritance. Subclasses are defined by extending the class behavior. This means that subclasses are more loosely related to their parent than a subtype to a type. Both state and methods may be extended.

Subtyping strongly contrains behavior while subclassing is an unconstrained mechanism. It is the inheritance mechanism of OOP that distingushes between types and classes.

These examples illustrate that subtype inheritance is different from subclass inheritance. Subclasses may define behavior completely unrelated to the parent class.

Types are used for type checking while classes are used for generating and managing objects with uniform properties and behaviors. Every class is a type. However, not every type is a class, since predicates do not necessaryly determine object templates. We will use the term type to refer to structure and values while the term class will be used to refer to behavior.

Examples

```
Queue -- insert_rear, delete_front
Deque -- insert_front, delete_front, insert_rear, delete_rear
```

```
Stack -- push, pop
List -- cons, head, tail
Binary tree -- insert, remove, traverse
Doublely linked list --
Graph -- linkto, path,
Natural numbers -- Ds
Integers -- (=-,Ds)
Rationals
Reals -- (+-,Ds,Ds)
Complex (a,b) or (r,$\theta$)
```

Historical Perspectives and Further Reading

- History
  - Simula
  - ADT
  - Small-Talk
  - Modula-2, C++, Eiffel
  - Flavors, CLOS
- Subtypes (subranges)
- Generic types
- Inheritance -- Scope generalization
- OOP
  - Objects--state + operations
  - Object Classes-- Class, Subclass

- Objects--state + operations
- Object Classes-- Class, Subclass
- Inheritance mechanism

Much of this section follows Peter Wegner\cite{Wegner90}.

Exercises

1. [time/difficulty] (section) Problem statement
2. Stack
3. Queue
4. Tree
5. Construct a ``turtle graphics''
6. Construct a table handler
7. Grammar
8. Prime number sieve
9. Account, Checking, Savings
10. Point, circle

Evaluation of Programming Languages

Models of Computation

The first requirement for a general purpose programming languge is that its computational model must be *universal*. That is, every problem that has an algorithmic solution must be solvable in the computational model. This requirement is easily met as the lambda calculus and the imperative model show. The computational model must be *implementatable* on a computer.

Syntax

Principle

Simplicity: The language should be based upon as few ``basic concepts'' as possible.

Orthogonality: Independent functions should be controlled by independent mechanisms.

Regularity: A set of objects is said to be regular with respect to some condition if, and only if, the condition is applicable to each element of the set. The basic concepts of the language should be applied consistently and universally.

Type Completeness: There should be no *arbitrary* restriction on the use of the types of values. All types have equal status. For example, functions and procedures should be able to have any type as parameter and result. This is also called the principle of regularity.

Parameterization: A formal parameter to an abstract may be from any syntactic class.

Analogy: An analogy is a conformation in pattern between unrelated objects. Analogies are generalizations which are formed when constants are replaced with variables resulting in similarities in structure. Analogous operations should be performed by the same code parameterized by the type of the objects.

Correspondence: For each form of definition there exists a corresponding parameter mechanism and *vice versa*.

Semantics

Principle

Clarity: The mechanisms used by the language should be well defined, and the outcome of a particular section of code easily predicted.

Referential Transparency: Any part of a syntactic class may be replaced with an equal part without changing the meaning of the syntactic class (substitutivity of equals for equals).

Sub-types: A sub-type may appear wherever an element of the super-type is expected.

Pragmatics

- Naturalness for the application (relations, functions, objects, processes)
- Support for abstraction
- Ease of program verification
- Programming environment (editors, debuggers, verifiers, test data generators, pretty printers, version control)
- Operating Environment (batch, interactive, embedded-system)
- Portability
- Cost of use (execution, translation, programming, maintenance)

Applicability

Principle

Expressivity: The language should allow us to construct as wide a variety of programs as possible.

Extensibility: New objects of each syntactic class may be constructed (defined) from the basic and defined constructs in a systematic way.

Example: user defined data types, functions and procedures. Binding, Scope, Lifetime,

Safety

Principle

Safety: Mechanisms should be available to allow errors to be detected.

Type checking-static and dynamic, range checking

Principle

the Data Invariant: A *data invariant* is a property of an object that holds whenever control is not in the object. Objects should be designed around a data invariant.

Information Hiding: Each ``basic program unit'' should only have access to the information that it requires.

Explicit Interfaces: Interfaces between basic program units should be stated explicitly.

Privacy: The private members of a class are inaccessible from code outside the class.

Abstraction

Principle

Abstraction: Abstraction is an emphasis on the idea, qualities and properties rather than the particulars (a suppression of detail). An abstract is a named syntactic construct which may be invoked by mentioning the name. Each syntactic class may be referenced as an abstraction. Functions and procedures are abstractions of expressions and commands respectively and there should be abstractions over declarations (generics) and types (parameterized types). Abstractions permit the suppression of detail by encapsulation or naming. Mechanisms should be available to allow recurring patterns in the code to be factored out.

Qualification: A block may be introduced in each syntactic class for the purpose of admitting local declarations. For example, block commands, block expressions, block definitions.

Representation Independence: A program should be designed so that the representation of an object can be changed without affecting the rest of the program.

Generalization

Principle

Generalization: Generalization is a broadening of application to encompass a larger domain of objects of the same or different type. Each syntactic class may be generalized by replacing a constituent element with a variable. The idea is to enlarge of domain of applicability of a construct. Mechanisms should be available to allow analogous operations to be performed by the same code.

polymorphism, overloading, generics

Implementation

Principle

Efficiency: The language should not preclude the production of efficient code. It should allow the programmer to provide the compiler with any information which may improve the resultant code.

Modularity: Objects of each syntactic class may be compiled separately from the rest of the program.

Novice users of a programming language require *language tutorials* which provide examples and intuitive explanations. More sophisticated users require *reference manuals* which catalogue all the features of a programming language. Even more sophisticated students of a programming language require complete and formal descriptions which eliminate all ambiguity from the language description.

Trends in Programming Language Design

streams, lazy evaluation, reactive systems, knowledge based systems, concurrency, efficient logic and functional languages, OOP.

Hankin, Chris and Nielson, Hanne R. eds (1996)
Symposium on Models of Programming Languages and Computation *ACM Computing Surveys*, *28*, 2, (June 1996), 293-359.

Stack Machine

Objectives
To introduce the machine organization and programming of a stack machine.
Concepts
Lab Techniques
Prerequisites

Background

In a stack machine most instructions obtain their arguments from the stack and place their results on the stack.

Machine Organization
The stack machine consists of a code segment **C** for the program, a data segment **D** for data and a stack, a program counter **PC** to contain the address of the next instruction, a stack top **T** pointer to point to the top of the expression stack (also part of the Store), an instruction register **I** to hold the current instruction, an input device **Input** and an output device **Output**.

Instruction Set
An instruction consists of an operation code and at most one parameter. The action of the instruction is described using a mixture of English language description and mathematical formalism. The mathematical formalism is used to note changes in values that occur to the registers, the data store, the program counter and the input and output devices.

Instruction	Operands	Semantics	Comments
add		D[T-1] := D[T-1] + D[T] T := T-1	Integer add
sub		D[T-1] := D[T-1] - D[T] T := T-1	Integer subtract
mult		D[T-1] := D[T-1] * D[T] T := T-1	Integer multiply
div		D[T-1] := D[T-1] / D[T] T := T-1	Integer divide
lit	X	D[T+1] := X; T := T+1	Push X on the Stack
load	*src*	D[T+1] := D[src]; T := T+1	Push from Store to Stack
store	*dst*	D[dst] := D[T]; T := T-1	Copy top of stack to Store

st	X	T := X	Set stack top pointer to X
jmp		PC := D[T]; T := T-1	Unconditional jump
jmpz		if D[T] = 0 then PC := D[T-1]; T := T-2 else T := T-2	Jump on zero
jmpn		if D[T] < 0 then PC := D[T-1]; T := T-2 else T := T-2	Jump on negative
halt			Halt
read		D[T+1] := Input; T := T+1	Push input item on the Stack
write		Output := D[T]; T := T-1	Put top of Stack to Output

src and *dst* designate source and destination respectively. Division by zero results in unpredictable results.

Operation
Most operations find their arguments on the expression stack. The program counter is initialized to the location of the first instruction. The machine repeatedly fetches the instruction at the address in the PC, increments the PC and executes the instruction and stops when the the halt instruction encountered.

```
PC := 0;                {Initialize the program counter}
repeat
        I := C[PC];     {Fetch instruction}
        PC := PC+1;     {Increment program counter}
        execute(I);     {Execute instruction}
until I = halt
```

Activitities

Assignment

1. Write a program to read two numbers from the input, compute and print their sum on the output.
2. Write a program to read two numbers from the input and print the value of the largest to the output.
3. Use a sentinal controlled loop to read non-negative numbers, compute and print their sum.
4. Use a counter controlled loop to read seven numbers, some positive and some negative and compute and print their average.
5. Read a series of numbers and determine and print the largest number. The first number read indicates how many numbers should be processed.
6. Implement the stack machine.

Hand in

Extra Credit

Multiparadigm Programming Language 0.9

The goal of this work is

- to design an abstract grammar for those elements that programming languages have in common in particular, for abstraction, generalization, and modules and
- to integrate the grammar with abstract grammars for a variety of programming paradigms.

This work is supports ideas developing in *Introduction to Programming Languages* where abstraction, generalization and computational models are used as unifying concepts for understanding programming languages. The goal in that document is to provide a top-down description of the language design process - idea, abstract sysntax, semantics, concrete syntax, formal semantics, and implementation

Bibliography

AbSus85
Abelson, H., Sussman, G.J., and Sussman, J.
Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Massachusetts, 1985.

Backus78
Backus, J. W., ``Can Programming Be Liberated from the von Neumann Style?" CACM, vol. 21, no. 8, pp. 613-614.

Bare84
Barendregt, H. P., *The Lambda Calculus: Its Syntax and Semantics*. 2d ed. North-Holland, 1984.

BirdWad88
Bird, R.A. and Wadler, P.L.,
Introduction to Functional Programming. Prentice/Hall International, 1988.

Boehm66
Boehm, C. and Jacopini, G., ``Flow Diagrams, Turnign Machines, and Languages with Only Two Foramation Rules." CACM, vol. 9, no. 5, pp. 366-371.

CF68
Curry. H. B. and Feys, R., *Combinatory Logic*, Vol. I. North-Holland, 1968.

CHS72
Curry. H. B., Hindley, J. R., and Seldin, J. P., *Combinatory Logic*, Vol. II. North-Holland, 1972.

DJL88
Deransart, P., Jourdan, M., and Lorho, B., *Attribute Grammars: Definitions, Systems and Bibliography*. Lecture Notes in Computer Science 323. Springer-Verlag, 1988.

Dijk68
Dijkstra, E. W., ``Goto Statement Considered Harmful." Communications of the ACM vol. 11 no. 5 (May 1968): pp. 147-149.

Foster96
Foster, I., Compositional Parallel Programming Languages *TOPLAS* Vol 18 No. 4 (July 1996): pp. 454-476.

Gries81
Gries, D., *The Science of Programming* Springer-Verlag, New York, 1981.

Hehner84
Hehner, E. C. R.,
The Logic of Programming. Prentice/Hall International, 1984.

Hend80
Henderson, Peter,
Functional Programming: Application and Implementation. Prentice/Hall International, 1980.

HS86

Hindley, J. R., and Seldin, J. P., *Introduction to Combinators and λ Calculus*, Cambridge University Press, London, 1986.

HU79

Hopcroft, J. E. and Ullman, J. D., *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

Knuth68

Knuth, D. E., ``Semantics of context-free languages." Mathematical Systems Theory, vol. 2, 1968, pp. 127-145. Correction in Mathematical Systems Theory, vol. 5, 1971, p. 95.

Kowalski79

Kowalski, R. A., ``Algorithm = Logic + Control". CACM vol. 22 no. 7, pp. 424-436, 1979.

Landin66

Landin, P. J., The next 700 programming languages, *Communications of the ACM* 9, 157-64 1966.

McCarthy60

McCarthy, J., ``Recursive functions of symbolic expressions and their computation by machine, Part I." CACM vol. 3 no. 4, pp. 184-195, 10, 1960.

McCarthy65

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M., *LISP 1.5 Programmer's Manual*. 2d ed. MIT Press, Cambridge, MA. 1965.

MLennan90

MacLennan, Bruce J.,
Functional programming: practice and theory. Addison-Wesley Publishing Company, Inc. 1990.

Miller67

Miller, G. A.,
The Psychology of Communication. Basic Books, New York, 1967.

PJones87

Peyton Jones, Simon L.,
The Implementation of Functional Programming Languages. Prentice/Hall International, 1987.

PittPet92

Pittman, T. and Peters, J.,
The Art of Compiler Design: Theory and Practice. Prentice-Hall, 1992.

Pratt84

Pratt, T. W.,
Programming Languages: Design and Implementation. Printice-Hall, 1984.

Revesz88

Révész, G. E., *Lambda-Calculus, Combinators, and Functional Programming*. Cambridge University Press, Cambridge, 1988.

Schmidt88

Schmidt, D. A., *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown, Dubuque, Iowa, 1988.

SchFre85

Schreiner, A. T. and Freidman, H. G., *Introduction to Compiler Construction with Unix* Prentice-Hall, 1985.

Scott87

Scott, D. S., *Denotational Semantics: The Scott-Strachey Approach to Programming Language*

Theory. MIT Press, 1987.

Steele84
Steele, G. L., Jr., *Common Lisp.* Digital Press, Burlington, MA. 1984.

Tenn81
Tennent, R. D., *Principles of Programming Languages*, Prentice-Hall International, 1981.

Wegner90
Wegner, Peter, ``Concepts and Paradigms of Object-Oriented Programming." OOPS Messenger vol. 1 no. 1 (August 1990): pp. 7-87.

Worf56
Worf, Benjamin,
Language thought and reality, MIT Press, Cambridge Mass., 1956.

Definitions

abstract type
: An *abstract data type* ---

abstraction
: An abstraction is the

actual parameter

aliasing
: *Aliasing* occurs whenever a given object becomes accessible through more than one name.

actual parameter

argument

assembly language

assertion

backtrack

binding
: *Binding* is an association between two objects.

block

class

clause

coertion

composite type
: A *composite type* is a type whose values are compose of simpler values.

computation
: A *computation* is the application of a sequence of operations to a set of values to yield a value.

computational model
: A *computational model* is a collection of values and operations.

concurrent programming
: *Concurrent programming is characterized by programming with more than one process.*

context

context-sensitive
: A syntactic element is *context-sensitive* if its value depends on the context in which it appears.

coroutine

deadlock

domain

environment

exception

formal parameter

functional programming
: *Functional programming* is characterized by programming with values, functions and functional forms.

generator

imperative programming
: *Imperative programming* is characterized by programming sequential modifications to a state.

inheritance

instance

iterator

lexical analyzer
: a scanner

live-lock

liveness

logic programming
: *Logic programming* is characterized by programming with relations and deduction.

machine language
: 1's and 0's.

method

module

object

object-oriented programming
: *Object-oriented programming* is characterized by programming with objects, messages, and hierarchies of objects.

overloading

parameter

partition

polymorphism

pragmatics
: The *pragmatics* of a programming language describe the degree of success with which a programming language meets its goals both in its faithfulness to the underlying model of computation and in its utility for human programmers.

primitive type
: A *primitive type* is a type whose values cannot be decomposed. The values are atomic.

process

program
: A *program* is a specification of a computation.

programming language
: A *programming language* is a notation for writing programs.

recursive type
: A *recursive type* is a type whose values may be composed of other values of the same type.

safety

scanner
: A *scanner* is a program which groups characters in an input stream into a sequence of tokens.

scope

semantics
: The *semantics* of a programming language describe the relationship between the syntactical elements and the model of computation.

semantic algebra
: A *semantic algebra* is set of values and operations defined on those values. A semantic algebra is distinguished from a type in that semantic algebras are the objects denoted in denotational

semantics while types are the syntactic objects

semantic domain

A *semantic domain* is a set of values.

side effect

A *side effect* is a modification of a non-local environment.

starvation

state

static semantics

The *static semantics* is the description of the structural constraints (context-sensitive aspects) that cannot be adequately described by context-free grammars.

symbol table

syntax

The *syntax* of a programming language describes the structure of programs.

type

A *type* is a set of values and a set of operations (see semantic algebra).

value

A *value* is any thing that may be evaluated, stored, incorporated in a data structure, passed as an argument to a procedure or function, returned as a function result, and so on.

variable

virtual machine

Index

Code

Chapter 1: Introduction

Chapter 2: Syntax

1. Recursive descent parser in Prolog

Chapter N: Translation

1. Sample compiler in Prolog

Answers

Chapter 1: Introduction

Chapter 2: Syntax

Chapter :

Chapter :

Chapter :

Chapter :

Chapter :

Chapter :

Chapter :

Chapter :

Chapter :

Chapter :

Chapter :

Chapter :

Chapter :

Appendix : Logic

\appendix %\input{definitions} \input{fol}

Disclamer and Copyright

This information is provided in good faith but no warranty can be made for its accuracy. Opinions expressed are entirely those of myself and cannot be taken to represent views of past, present or future employers.

If you have questions or comments feel free to send mail to me.
Anthony A. Aaby

Last update:

Send comments to: webmaster@cs.wwc.edu

Gödel Tutorial

Introduction

Example Programs

Mathematics

Fibonacci Function

```
MODULE    Fibonacci.

IMPORT    Integers.


PREDICATE Fib : Integer * Integer.

% Fib(k,n) <-> n is the Fibonacci number F_{k} of rank k.

Fib(0,0).
Fib(1,1).
Fib(k,n) <-
          k > 1 &
          FibIt(k-2,1,1,n).


PREDICATE FibIt : Integer * Integer * Integer * Integer.

% FibIt(k,f,g,n) <-> n  =  F_{k} * f  +  F_{k+1} * g.

FibIt(0,_,g,g).
FibIt(k,f,g,n) <-
          k > 0 &
          g < n &
```

```
            FibIt(k-1,g,f+g,n).
```

GCD Function

```
MODULE      GCD.

IMPORT      Integers.

PREDICATE   Gcd : Integer * Integer * Integer.

Gcd(i,j,d) <-
            CommonDivisor(i,j,d) &
            ~ SOME [e] (CommonDivisor(i,j,e) & e > d).

PREDICATE   CommonDivisor : Integer * Integer * Integer.

CommonDivisor(i,j,d) <-
            IF (i = 0 \/ j = 0)
            THEN
              d = Max(Abs(i),Abs(j))
            ELSE
              1 =< d =< Min(Abs(i),Abs(j)) &
              i Mod d = 0 &
              j Mod d = 0.
```

Database

Family Tree

```
MODULE        DB.

BASE          Person.

CONSTANT      Fred, Mary, George, James, Jane, Sue : Person.

PREDICATE     Ancestor, Parent, Mother, Father : Person * Person.

Ancestor(x,y) <-
              Parent(x,y).
Ancestor(x,y) <-
              Parent(x,z) &
              Ancestor(z,y).

Parent(x,y) <-
              Mother(x,y).
Parent(x,y) <-
              Father(x,y).
```

```
Father(Fred, Mary).
Father(George, James).

Mother(Sue, Mary).
Mother(Jane, Sue).
```

Sports Database

```
MODULE      Sports.

IMPORT      Sets.

BASE        Person, Sport, PersonSports.

CONSTANT    Mary, Bill, Joe, Fred : Person;
            Cricket, Football, Tennis : Sport.

FUNCTION    Pair : Person * Set(Sport) -> PersonSports.

PREDICATE   Likes : Person * Sport.

Likes(Mary, Cricket).
Likes(Mary, Tennis).
Likes(Bill, Cricket).
Likes(Bill, Tennis).
Likes(Joe, Tennis).
Likes(Joe, Football).
```

List/Set Processing

```
MODULE     SetProcessing.

IMPORT     Sets.

PREDICATE  Sum : Set(Integer) * Integer.

Sum(s,y) <-
        x In s &
        Sum1(s\{x},x,y).

PREDICATE  Sum1 : Set(Integer) * Integer * Integer.

Sum1({},x,x).
Sum1(s,x,x+w) <-
        z In s &
        Sum1(s\{z},z,w).

PREDICATE  Max : Set(Integer) * Integer.
```

```
Max(s,y) <-
        x In s &
        Max1(s\{x},x,y).

PREDICATE  Max1 : Set(Integer) * Integer * Integer.

Max1({},x,x).
Max1(s,x,y) <-
        z In s &
        IF z>x THEN Max1(s\{z},z,y) ELSE Max1(s\{z},x,y).
```

Last update:

Send comments to: webmaster@cs.wwc.edu

Scheme Tutorial

Introduction

Scheme is an imperative language with a functional core. The functional core is based on the lambda calculus. In this chapter only the functional core and some simple I/O is presented.

In functional programming, parameters play the same role that assignments do in imperative programming. Scheme is an applicative programming language. By applicative, we mean that a Scheme function is applied to its arguments and returns the answer. Scheme is a descendent of LISP. It shares most of its syntax with LISP but it provides lexical rather than dynamic scope rules. LISP and Scheme have found their main application in the field of artificial intelligence.

The purely functional part of Scheme has the semantics we expect of mathematical expressions. One word of caution: Scheme evaluates the arguments of functions prior to entering the body of the function (eager evaluation). This causes no difficulty when the arguments are numeric values. However, non-numeric arguments must be preceded with a single quote to prevent evaluation of the arguments. The examples in the following sections should clarify this issue.

Scheme is a weakly typed language with dynamic type checking and lexical scope rules.

The Structure of Scheme Programs

A Scheme program consists of a set of function definitions. There is no structure imposed on the program and there is no main function. Function definition may be nested. A Scheme program is executed by submitting an expression for evaluation. Functions and expressions are written in the form

```
(function_name arguments)
```

This syntax differs from the usual mathematical syntax in that the function name is moved inside the parentheses and the arguments are separated by spaces rather than commas. For example, the mathematical expression 3 + 4 * 5 is written in Scheme as

```
(+ 3 (* 4 5))
```

Syntax

The programming language Scheme is syntactically close to the lambda calculus.

Scheme Syntax

E in Expressions
I in Identifiers (variables)
K in Constants

E ::= K | I | (E_0 E^*) | (lambda (I^*) E_2) | (define I E')

The star `*' following a syntactic category indicates zero or more repetitions of elements of that category thus Scheme permits lambda abstractions of more than one parameter. Scheme departs from standard mathematical notation for functions in that functions are written in the form (*Function-name Arguments*...) and the *arguments are separated by spaces and not commas*.

For example,

```
(+ 3 5)
(fac 6)
(append '(a b c) '(1 2 3 4))
```

The first expression is the sum of 3 and 5, the second presupposes the existence of a function `fac` to which the argument of 6 is presented and the third presupposes the existence of the function append to which two lists are presented. Note that the quote is required to prevent the (eager) evaluation of the lists. Note uniform use of the standard prefix notation for functions.

Types

Among the constants (atoms) provided in Scheme are numbers, the boolean constants #T and #F, the empty list (), and strings. Here are some examples of atoms and a string:

```
A, abcd, THISISANATOM, AB12, 123, 9Ai3n, "A string"
```

Atoms are used for variable names and names of functions. A list is an ordered set of elements consisting of atoms or other lists. Lists are enclosed by parenthesis in Scheme as in LISP. Here are some examples of lists.

```
(A B C)

(138 abcde 54 18)

(SOMETIMES (PARENTHESIS (GET MORE)) COMPLICATED)

()
```

Lists are can be represented in functional notation. There is the empty list represented by () and the list *cons*truction function `cons` which constructs lists from elements and lists as follows: a list of one element is (cons X ()) and a list of two elements is (cons X (cons Y ())).

Simple Types

The simple types provided in Scheme are summarized in this table.

TYPE & VALUES
boolean & #T, #F
number & integers and floating point
symbol & character sequences
pair & lists and dotted pairs
procedure & functions and procedures

Composite Types

The composite types provided in Scheme are summarized in this table.

TYPE & REPRESENTATION & VALUES
list & (*space separated sequence of items*) & any
in function& defined in a later section &
in

Type Predicates

A predicate is a boolean function which is used to determine membership. Since Scheme is weakly typed, Scheme provides a wide variety of type checking predicates. Here are some of them.

PREDICATE & CHECKS IF
(boolean? arg) & arg is a boolean
in (number? arg) & arg is a number
in (pair? arg) & arg is a pair
in (symbol? arg) & arg is a symbol
in (procedure? arg) & arg is a function
in (null? arg) & arg is empty list
in (zero? arg) & arg is zero
in (odd? arg) & arg is odd
in (even? arg) & arg is even

in

Numbers, Arithmetic Operations, and Functions

Scheme provides the data type, `number`, which includes the integer, rational, real, and complex numbers.

Some Examples:

```
(+ 4 5 2 7 5 2) -  is equivalent to \( 4 + 5 + 2 + 7 + 5 + 2 \)
(/ 36 6 2) -  is equivalent to \(\frac{\frac{36}{6}}{2} \)
(+ (* 2 2 2 2 2) (* 5 5)) - is equivalent to \( 2^{5} + 5^{2} \)
```

Arithmetic Operators

SYMBOL & OPERATION
+ & addition
in - & subtraction
in * & multiplication
in / & real division
in quotient & integer division
in modulo & modulus
in

Lists

Lists are the basic structured data type in Scheme. Note that in the following examples the parameters are quoted. The quote prevents Scheme from evaluating the arguments. Here are examples of some of the built in list handling functions in Scheme.

cons
takes two arguments and returns a pair (list).

```
(cons '1 '2)          is   (1 . 2)
(cons '1 '(2 3 4))      is   (1 2 3 4)
(cons '(1 2 3) '(4 5 6))  is   ((1 2 3) 4 5 6)
```

The first example is a dotted pair and the others are lists. \marginpar{expand} Either lists or dotted pairs can be used to implement records.

car
returns the first member of a list or dotted pair.

```
(car '(123 245 564 898))             is   123
(car '(first second third))           is   first
(car '(this (is no) more difficult))   is  this
```

cdr
returns the list without its first item, or the second member of a dotted pair.

```
(cdr '(7 6 5))              is  (6 5)
```

```
(cdr '(it rains every day))  is  (rains every day)
(cdr (cdr '(a b c d e f)))   is   (c d e f)
(car (cdr '(a b c d e f)))   is   b
```

null?
returns \#t if the {\bf obj}ect is the null list, (). It returns the null list, (), if the object is anything else.

list
returns a list constructed from its arguments.

```
(list 'a)                        is  (a)
(list 'a 'b 'c 'd 'e 'f)      is  (a b c d e f)
(list '(a b c))              is  ((a b c))
(list '(a b c) '(d e f) '(g h i)) is  ((a b c)(d e f)(g h i))
```

length
returns the length of a list.

```
(length '(1 3 5 9 11))                 is  5
```

reverse
returns the list reversed.

```
(reverse '(1 3 5 9 11)) is  (11 9 5 3 1)
```

append
returns the concatenation of two lists.

```
(append '(1 3 5)  '(9 11))  is  (1 3 5 9 11)
```

Boolean Expressions

The standard boolean objects for true and false are written \verb+#t+ and \verb+#f+. However, Scheme treats any value other than \verb+#f+ and the empty list \verb+()+ as true and both \verb+#f+ and \verb+()+ as false. Scheme provides {\bf not, and, or} and several tests for equality among objects.

Logical Operators

SYMBOL & OPERATION not & negation
and & logical conjunction
or & logical disjunction

Relational Operators

SYMBOL & OPERATION
= & equal (numbers)
(<) & less than
(<=) & less or equal
(>) & greater than
(>=) & greater or equal

eq? & args are identical
eqv? & args are operationally equivalent
equal? & args have same structure and contents

Conditional Expressions

Conditional expressions are of the form:

```
(if test-exp then-exp)

(if test-exp then-exp else-exp).
```

The `test-exp` is a boolean expression while the `then-exp` and `else-exp` are expressions. If the value of the `test-exp` is true then the `then-exp` is returned else the `else-exp` is returned. Some examples include:

```
(if (> n 0) (= n 10))
(if (null? list) list (cdr list))
```

The **list** is the then-exp while **(cdr list)** is the else-exp. Scheme has an alternative conditional expression which is much like a case statement in that several test-result pairs may be listed. It takes one of two forms:

```
(cond
   (test-exp1 exp ...)
    (test-exp2 exp ...)
    ...)
(cond
   (test-exp exp ...)
    ...
   (else exp ...))
```

The following conditional expressions are equivalent.

```
(cond
        ((= n 10) (= m 1))
        ((> n 10) (= m 2) (= n (* n m)))
        ((< n 10) (= n 0)))

(cond
        ((= n 10) (=m 1))
        ((> n 10) (= m 2) (= n (* n m)))
        (else (= n 0)))
```

Functions

Definition expressions bind names and values and are of the form:

```
(define id exp)
```

Here is an example of a definition.

```
(define pi 3.14)
```

This defines `pi` to have the value 3.14. This is not an assignment statement since it cannot be used to rebind a name to a new value.

Lambda Expressions

User defined functions are defined using lambda expressions. Lambda expressions are unnamed functions of the form:

```
(lambda (id...) exp )
```

The expression `(id...)` is the list of formal parameters and *exp* represents the body of the lambda expression. Here are two examples the application of lambda expressions.

```
((lambda (x) (* x x)) 3)      is 9
((lambda (x y) (+ x y)) 3 4) is  7
```

Here is a definition of a squaring function.

```
(define square (lambda (x)  (* x x)))
```

Here is an example of an application of the function.

```
1 ]=> (square 3)
;Value: 9
```

Here are function definitions for the factorial function, gcd function, Fibonacci function and Ackerman's function.

```
(define fac
          (lambda (n)
                  (if (= n 0)
                         1
                         (* n (fac (- n 1))))))
%

%

(define fib
          (lambda (n)
                  (if (= n 0)
                         0
                         (if (= n 1)
                                    1
                                   (+ (fib (- n 1)) (fib (- n 2)))))))
%

%

(define ack
```

```
            (lambda (m n)
                    (if (= m 0)
                        (+ n 1)
                        (if (= n 0)
                                    (ack (- m 1) 1)
                                   (ack (- m 1) (ack m (- n 1)))))))
    %

%

    (define gcd
            (lambda (a b)
                    (if (= a b)
                        a
                        (if (> a b)
                                    (gcd (- a b) b)
                                   (gcd a (- b a))))))
```

Here are definitions of the list processing functions, sum, product, length and reverse.

```
    (define sum
            (lambda (l)
                    (if (null? l)
                        0
                        (+ (car l) (sum (cdr l))))))
    %

%

    (define product
            (lambda (l)
                    (if (null? l)
                        1
                        (* (car l) (sum (cdr l))))))
    %

%

    (define length
            (lambda (l)
                    (if (null? l)
                        0
                        (+ 1 (length (cdr l))))))
    (define reverse
            (lambda (l)
                    (if (null? l)
                        nil
                        (append (reverse (cdr l)) (list (car l))))))
```

Nested Definitions

Scheme provides for local definitions by permitting definitions to be nested. Local definitions are introduced using the functions `let`, `let*` and `letrec`. The syntax for the `define` function is expanded to permit local definitions. The syntax of the `define` function and the `let` functions is

Scheme Syntax

E in Expressions
I in Identifier(variable)
...
B in Bindings
...

E ::= ...| (lambda (I...) E...) |
(let B_0 E_0) | (let* B_1 E_1) | (letrec B_2 E_2) |...
B ::= ((I E)...)

Note that there may be a sequence of bindings. For purposes of efficiency the bindings are interpreted differently in each of the ``let'' functions. The `let` values are computed and bindings are done in parallel, this means that the definitions are independent. The `let*` values and bindings are computed sequentially, this means that later definitions may be dependant on the earlier ones. The `letrec` bindings are in effect while values are being computed to permit mutually recursive definitions. As an illustration of local definitions here is a definition of insertion sort definition with the insert function defined locally. Note that the body of `isort` contains two expressions, the first is a `letrec` expression and the second is the expression whose value is to be returned.

```
(define isort (lambda (l)
           (letrec
              ((insert  (lambda (x l)
                 (if (null? l)
                    (list x)
                    (if (<= x (car l))
                       (cons x l)
                       (cons (car l) (insert x (cdr l))))))))

           (if (null? l)
                  nil
                  (insert (car l) (isort (cdr l)))))))
```

{`letrec` is used since `insert` is recursively defined. Here are some additional examples:

```
; this binds x to 5 and yields  10
(let ((x 5)) (* x 2))
; this bind x to 10, z to 5 and yields 50.
(let ((x 10) (z 5)) (* x z))
```

`Lets may be nested. For example,` the expression

```
(let ((a 3) (b 4)
     (let ((double (* 2 a))
          (triple (* 3 b)))
        (+ double triple))))
```

is 18.

Input and Output Expressions

Scheme does not readily support the functional style of interactive programming since input is not passed as a parameter but obtained by successive evaluations of the builtin function **read**. For example,

```
(+ 3 (read))
```

returns the sum of 3 and the next item from the input. A succeeding call to {\tt read} will return the next item from the standard input, thus, {\tt read} is not a true function. The function {\bf display} prints its argument to the standard output. For example,

```
(display (+ 3 (read)))
```

displays the result of the previous function. The following is an example of an interactive program. It displays a prompt and returns the next value from the standard input.

```
(define prompt-read (lambda (Prompt)
   (display Prompt)
   (read)))
```

Higher Order Functions

A higher order function (or functional) is a function that either expects a function as a parameter or returns a function as a result. In Scheme functions may be passed as parameters and returned as results. Scheme does not have an extensive repertoire of higher order functions but {\tt apply} and {\tt map} are two builtin higher order functions.

- The function **apply** returns the result of applying its first argument to its second argument.

```
1 ]=>  (apply + '(7 5))

;Value:   12

1 ]=>  (apply max '(3 7 2 9))

;Value:   9
```

- The function **map** returns a list which is the result of applying its first argument to each element of its second argument.

```
1 ]=>   (map odd? '(2 3 4 5 6))

;Value: (() #T () #T ())
```

- Here is an example of a ``curried'' function passed as a parameter. **dbl** is a dubbling function.

```
1 ]=> (define dbl (lambda (x) (* 2 x)))
```

```
;Value: dbl

1 ]=> (map dbl '(1 2 3 4))

;Value: (2 4 6 8)

1 ]=>
```

- The previous example could also be written as:

```
1 ]=> (map (* 2) '(1 2 3 4))

;Value: (2 4 6 8)

1 ]=>
```

An Example Program

The purpose of the following function is to help balance a checkbook. The function prompts the user for an initial balance. Then it enters the loop in which it requests a number from the user, subtracts it from the current balance, and keeps track of the new balance. Deposits are entered by inputting a negative number. Entering zero (0) causes the procedure to terminate and print the final balance.

```
(define checkbook (lambda ()

; This check book balancing program was written to illustrate
; i/o in Scheme. It uses the purely functional part of Scheme.

        ; These definitions are local to checkbook
        (letrec

            ; These strings are used as prompts

           ((IB "Enter initial balance: ")
            (AT "Enter transaction (- for withdrawal): ")
            (FB "Your final balance is: ")

            ; This function displays a prompt then returns
            ; a value read.

            (prompt-read (lambda (Prompt)

                  (display Prompt)
                  (read)))

            ; This function recursively computes the new
            ; balance given an initial balance init and
            ; a new value t.  Termination occurs when the
            ; new value is 0.

            (newbal (lambda (Init t)
                  (if (= t 0)
                      (list FB Init)
```

```
                    (transaction (+ Init t)))))

            ; This function prompts for and reads the next
            ; transaction and passes the information to newbal

            (transaction (lambda (Init)
                      (newbal Init (prompt-read AT)))))

; This is the body of checkbook;  it prompts for the
; starting balance

  (transaction (prompt-read IB)))))
```

Appendix

DERIVED EXPRESSIONS

(cond (test1 exp1) (test2 exp2) ...)
a generalization of the conditional expression.

ARITHMETIC EXPRESSIONS

(exp x)
which returns the value of \(e^{x}\)

(log x)
which returns the value of the natural logarithm of {\bf x}

(sin x)
which returns the value of the sine of {\bf x}

(cos x)

(tan x)

(asin x)
which returns the value of the arcsine of {\bf x}

(acos x)

(atan x)

(sqrt x)
which returns the principle square root of x

(max x_1 x_2...)
which returns the largest number from the list of given {\bf num}bers

(min x_1 x_2...)

(quotient x_1 x_2)
which returns the quotient of \(\frac {x_1}{x_2}\)

(remainder x_1 x_2)
which returns the integer remainder of \(\frac {x_1}{x_2}\)

(modulo x_1 x_2)
returns x_1 modulo x_2

(gcd num1 num2 ...)
which returns the greatest common divider from the list of given {\bf num}bers

(lcm num1 num2 ...)
which returns the least common multiple from the list of given {\bf num}bers

(expt base power)
which returns the value of {\bf base} raised to {\bf power}

{\bf note: For all the trigonometric functions above, the x value should be in radians}

LIST EXPRESSIONS

(list obj)

returns a list given any number of {\bf obj}ects.

(make-list n)

returns a list of length {\bf n} and every atom is an empty list ().

HIGHER ORDER FUNCTIONS

(apply procedure obj ... list)

returns the result of applying {\it procedure} to {\it object} and returns the elements of {\it list}. It passes the first obj as the first parameter to procedure, the second obj as the second and so on. List is the remag arguments into a list to procedure. This is useful when some or all of the arguments are in a list.

(map procedure list)

returns a list which is the result of applying procedure to each element of {\bf list}.

I/O

(read)

returns the next item from the standard input file.

(write obj)

prints {\bf obj} to the screen.

(display obj)

prints {\bf obj} to the screen. Display is mainly for printing messages that do not have to show the type of object that is being printed. Thus, it is better for standard output.

(newline)

sends a newline character to the screen.

(transcript-on filename)

opens the file filename and takes all input and pipes the output to this file. An error is displayed if the file cannot be opened.

(transcript-off)

```
ends transcription and closes the file.
```

References

Abelson, Harold.
Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Mass. 1985.

Dybvig, R. Kent.
The Scheme Programming Language. Prentice Hall, Inc. Englewood Cliffs, New Jersey, 1987.

Springer, G. and Friedman, D.,
Scheme and the Art of Programming. The MIT Press, 1989.

Last update:

Send comments to: webmaster@cs.wwc.edu

ML Tutorial

In functional programming:

- Programs are collections of definitions.
- The basic mode of computation is the construction and application of functions. *Higher-order functions* take functions as arguments and return functions as results i.e., functions are first-class values.
- Functions are free from *side effects* (operations that permanently change the value of a variable).
- Recursion is the only method of repetion.
- rule-based programming
- Pattern matching
- *polymorphism* permits functions to take arguments of various types
- Typing: ML has a type inference system which which permits strong type checking without requiring declaration of the type of each variable.

- ML is case sensitive!
- To start up sml enter: /app4/sml/sml at the unix prompt
- To leave sml enter CTRL-D.
- To run an sml program foo enter: /app4/sml/sml < foo at the unix prompt
- To run a program interactively in sml, start up sml and type the sml expression: use("foo");
- To interact with a program in sml, type in an expression in response to the ML prompt (-). ML responds with the value and type of the expression.

Expressions

1. Constants: Integers, Reals, Booleans, Strings (encolosed in double quotes). Characters are single strings of length 1. Newline, Tab, Backslash, Quote mark, Control character
2. Arithmetic Operators (in order of precedence): +, -; *, /, mod, div; ~ (unary minus).
3. String Operators: concatenation A^B, empty string "".
4. Comparison Operators: =, <, >, <=, >=, <> (as in Pascal) with lower precedence then arithmetic operators.
5. Logical Operators: not, andalso, orelse. The latter two are short-circuit operators.
6. Selection Operator: if E then F else G where type of F and G must be the same.

Type Consistency

When operators are given arguments of the incorrect type, ML displays an error message indicating a type constructor mismatch (tycon) and displays the expected and actual argument types.

The arithmetic operators do not permit mixed types.

1. real(I), ceiling, floor, truncate, ord, chr

are used convert between values of one to another type.

Variables and Environments

In imparative programming languages an *environment* consists of a collection of *variables* that have a names *identifiers* and hold values. Associated with each variable is a type. The set of variable constitutes the *store*. Computation proceeds by side-effects, i.e., by changing the store.

In ML, computation proceeds by adding new *value bindings* to the store not by side-effects.

1. Identifiers: are character strings with certain restrictions.

    - alphanumeric begin with upper or lower case letter or an apostrophe followed by zero or more letters or digits. Type variables must begin with the apostrophe
    - symbolic identifiers are formed from the set of characters + - / * < > = ! @ # $ % ^ & ' ~ \ | ? :. It is recommended that symbolic identifiers be used for user defined operators.
2. Top-Level environment is the ML system environment
3. Value bindings: an assignment-Like statement used to extend the environment.
    - **Name** Variable description
    - **Example**

```
val identifier = value
```

    - **Context**
    - **Problem**
    - **Solution** Semicolons (;) to terminate or separate are optional
4. ML Program: programs are sequences of definitions
    - **Name**
    - **Example** programs are sequences of definitions
    - **Context**
    - **Problem**
    - **Solution** Semicolons (;) to terminate or separate definitions are optional

Tuples and Lists

- Tuples: (*item_0, ..., item_n*) -- two or more expressions of any type
  Accessing typles: #*i*T -- elements are indexed beginning at one
- Lists: [*item_0, ..., item_n*] -- zero or more expressions of one type
  List Notation and Operators: [ ], hd(L), tl(L), x::xs, L1@L2, nil
- Strings and Lists: L=explode(S), S=implode(L), i.e., S = implode(explode(S))

Functions

Function definitions:

- **Name**
- **Example**

```
fun identifier( parameter list ) = expression
```

- **Context**
- **Problem**
- **Solution**

Parameters: variables to which a function is applied in its definition
Arguments: expressions to which a function is applied.
Function type:

```
val name :  domain type -> range type
```

Examples

```
fun upper(c) = chr( ord(c) - 32 );

fun square( x ) = x*x;
Error: unbound type constructor: x

fun square( x:real ) = x*x;
```

Function application
pi*square(radius) or pi*square radius (function application has higher precedence than arithmetic operators)
Functions with more than one parameter
max3
Comments
(* ... *) nesting permitted
Recursive functions

```
fun reverse(L) =
        if L = nil then nil
        else reverse(tl(L)) @ [hd(L)]
```

Function execution: call by value with eager evaluation
Non-linear recursion: \[\left(\begin{array}{c}nm \end{array}\right) = n!/((n-m)!m!)\]
Mutual recursion:

```
fun
        definition of first function
and
        ...
and
        definition of last function;
```

Type Inference

Patterns

Patterns are expressions w/o variables and if it contains variables the variables are given values if they match.

```
fun identifier ( first pattern ) = first expression
|   identifier ( second pattern ) = second expression
...
|   identifier ( last pattern ) = last expression;
```

the identifiers must all be the same; a variable may appear just once in a pattern; `as' may not be used in a pattern; common patterns include: nil, x::xs ...

1. AS:

```
identifier as pattern
```

2. Anonymous variables: _
3. Pattern matching problems: explicit declaration detects spelling errors

ASSIGNMENT

do seven problems from the problem set beginning on page 60,

Local environments

- **Name**
- **Example**

```
let
   declarations
in
```

```
        expressions
    end
```

- **Context**
- **Problem**
- **Solution** the expressions must be separated by semicolons (;)

```
let
        declarations
in
        expressions
end
```

Examples:

```
fun merge(nil,M) = M
|   merge(L,nil) = L
| merge(L as x::xs, M as y::ys) =
        if (x:int) $<$ y then x::merge(xs,M)
        else y::merge(L,ys);

fun split(nil) = (nil,nil)
|   split([a]) = ([a],nil)
|   split(a::b::cs) =
        let
                val (M,N) = split(cs)
        in
                (a::M, b::N)
        end;


fun mergeSort(nil) = nil
|   mergeSort([a]) = [a]
|   mergeSort(L) =
        let
                val (M,N) = split(L);
                val M = mergeSort(M);
                val N = mergeSort(N);
        in
                merge(M,N)
end;
```

Exceptions

THEORY: partial functions -- need to report on improper use.

1. User-defined Exceptions.
   - **Name**
   - **Example**

     ```
     exception Foo;
     ...
     raise Foo
     ```

   - **Context**
   - **Problem**
   - **Solution**

```
exception BadN;
exception BadM;

fun comb(n,m) = if n < 0 then raise BadN
                else if m < 0 orelse m > n then raise BadM
                else if m = 0 orelse m = n then 1
                else comb(n-1,m) + comb(n-1,m-1)
```

2. Local exceptions.

```
fun comb(n,m) = let
                   exception BadN;
                   exception BadM;
                in
                   if n < 0 then raise BadN
                   else if m < 0 orelse m > n then raise BadM
                   else if m = 0 orelse m = n then 1
                   else comb(n-1,m) + comb(n-1,m-1)
                end
```

Side effiects

1. The Print Function.
   - **Name**
   - **Example** print(x)
   - **Context** x must be of type integer, real, Boolean, or string
   - **Problem**
   - **Solution**
2. Statement lists.
   - **Name**
   - **Example** (e$_0$; ...; e$_n$)
   - **Context**
   - **Problem**
   - **Solution**

```
fun printList(nil) = ()
|   printList(x::xs) = (print(x:int); print("\n"); printList(xs))
```

3. Simple input.
   - **Name**
   - **Example**

     ```
     open_in(``filename'') -- returns pointer to open file
     end_of_stream( file ) -- corresponds to eof in Pascal
     input( file, n ) -- return n characters of input in a string
     ```

   - **Context**
   - **Problem**
   - **Solution**

   READ REST OF CHAPTER

Polymorphic functions

```
fun identity(x) = x;
```

1. Operators with restricted polymorphism

- Arithmetic operators: +, -, *, and ~
- Division and remainder operators: /, div, and mod.
- Inequality operators: <, <=, >=, and >.
- Boolean operators: andalso, orelse and not.
- String operator: ^
- Type conversion operators: ord, chr, real, floor, ceiling, and truncate.

2. Operators with polymorphism
   - Tuple operators: (,,...,), #1, #2 ...
   - List operators: ::, @, hd, and tl, nil and [...].
   - equality operators: = and <>

Implementation of lists using cons cells.

Equality operator vs pattern matching: implications for polymorphism

```
fun reverse(L) =
        if L = nil then nil
        else reverse(tl(L)) @ [hd(L)];
VS

fun reverse(nil) = nil
|   reverse(x::xs) = reverse(xs) @ [x];
```

Higher-Order Functions

Functions that can take functions as arguments and/or produce functions as values are called *higher-order functions*.

Examples

1. Identity function

```
fun identity(x) = x;
```

2. Reverse

```
fun reverse(nil) = nil
|   reverse(x::xs) = reverse(xs) @ [x];
```

3. Trapezoidal rule

```
fun trap( a, b, n, F ) =
        if n <= 0 orelse b-a <= 0.0 then 0.0
        else
           let
              val delta = (b-a)/real(n)
           in
              delta*(F(a)+F(a+delta))/2.0 +
                  trap(a+delta,b,n-1,F)
           end;
```

4. map (applies its first argument to each element of a list)

```
fun map(F,nil) = nil
| map(F,x::xs) = F(x)::map(F,xs)

map(square,[1,2,3,4,5])
```

```
(*Anonymous function definition*)

map(fn x => x*x) [1,2,3];
```

5. reduce

```
exception EmptyList;

fun reduce(F,nil)   = raise EmptyList
|   reduce(F,[a])   = a
|   reduce(F,x::xs) = F(x,reduce(F,xs));

...
reduce(fn (x,y) => x+y,[1,2,3,4,5])
```

   Note: fold is built-in version of reduce

6. Example: variance -- average of squares minus the square of the average
7. Op: convert infix to function name -- op + (2,3)
8. filter
9. Composition of functions.

ASSIGNMENTdo 2 on pages 112-114

Defining New Types

Basic types: int, real, string, bool, unit, exn, instream; type variables 'a (any type), ''a (any equality type) Induction: $T_1 * T_2$, $T_1 -> T_2$, T_1 list, T_1 array, T_1 ref

1. Type definitions
   - **Name**
   - **Example**

     ```
     type identifier = type expression
     ```

   - **Context** Introduce an abreviation for a type
   - **Problem**
   - **Solution**
2. Polymorphic Type Definitions
   - **Name**
   - **Example**

     ```
     type (list of type parameters) identifier = type expression
     ```

   - **Context**
   - **Problem**
   - **Solution**
3. Data type declarations and data constructors
   - **Name** Data type declaration
   - **Example**

     ```
     datatype \=(list of type parameters) identifier \==
     first constructor expression |
     ...
     last constructor expression}
     ```

   - ***Context***

- ***Problem***
- ***Solution***
- ***Name*** *Enumerated type example.*
- ***Example***

```
datatype fruit = Apple | Pear | Grape;
```

- ***Context*** *Apple .. Grape are values*
- ***Problem***
- ***Solution***
- ***Name*** *Union type example.*
- ***Example***

```
datatype ('a, 'b) element =
P of 'a * 'b |
S of 'a;
```

- ***Context*** *P("hello", 7), S("this") are possible values*
- ***Problem***
- ***Solution***

4. *Recursively defined datatypes.*
   - ***Name*** *Binary Tree*
   - ***Example***

```
datatype 'label btree =
                  Empty |
Node of 'label * 'label btree * 'label btree
```

   - ***Context*** *leaves have the value {\sf Node(label item, Empty, Empty)} etc*
   - ***Problem***
   - ***Solution***

The ML Module System

1. *Modules. A module is a separately compilable unit, typically a file.*
   1. *Structures -- collections of types, datatypes, functions, exceptions and other elements*
   2. *Signatures -- collections of information describing the types an other specifications for some of the elements of a structure.*
   3. *Functors. -- operations that take one or more structures and produce another structure.*
2. *Information hiding. Information that is not usable outside a cluster. Informantion that is usable is said to be exported.*
3. *Structure.*
   - ***Name*** *Structure*
   - ***Example***

```
mmm\=mmm\=mmmmmmm\kill
structure identifier =
struct
elements of the structure
end
```

   - ***Context***
   - ***Problem***
   - ***Solution***
4. *Signature.*
   - ***Name*** *Signature*
   - ***Example***

```
signature identifier = usesig specifications end
```

- ***Context***
- ***Problem***
- ***Solution***

5. *EXAMPLE: stack*

```
structure STACK = struct

   exception EmptyStack

   datatype 'item stack =
       Empty |
       Node of 'item * 'item stack

   fun isEmpty( S ) = S = Empty

   fun create() = Empty

   fun push( x, S ) = Node( x, S )

   fun pop( Empty ) = raise EmptyStack
   |   pop( Node( x, S) ) = S

   fun top( Empty ) = raise EmptyStack
   |   top( Node( x, S ) ) = x

end


signature INT\_STACK = sig

    type 'item stack
    val create : unit $->$ int stack
    val pop : int stack $->$ int stack
    val push : int * int stack $->$ int stack
    val top : int stack $->$ int

  end

structure IntStack : INT\_STACK = STACK

open IntStack
```

More about Exceptions

```
exception OutOfRange of int*int;

fun comb1(n,m) = if n <= 0 then raise OutOfRange(n,m)
                 else if m < 0 orelse m > n then raise OutOfRange(n,m)
                 else if m = 0 orelse m = n then 1
                 else comb(n-1,m) + comb(n-1,m-1)

fun comb(n,m) = comb1(n,m) handle
                        OutOfRange(0,0) => 1 |
                        OutOfRange(n,m) => (print("Out of Range: n=");
                                            print(n);
                                            print("m=");
                                            print(m);
                                            print("\n");
```

```
                                                    0)
```

Computing with functions as values

1. *Function Composition.*
   - ***Name***
   - ***Example*** *F o G*
   - ***Context***
   - ***Problem***
   - ***Solution***
2. *Curried Functions.*

```
fun comb n m = if m = 0 orelse m = n then 1
               else comb(n-1,m) + comb(n-1,m-1)

fun exponent x 0 = 1.0
| exponent x y = x * exponent x (y-1)
```

3. *Partially instantiated functions.*
4. *Folding Lists. (compare with reduce)*

```
fun fold F nil y = y
| fold F (x::xs) y = F(x, (fold F xs y))
...
fold (op +) L 0

fold (op *) L 1

fun length L = fold (fn(a,x) => x+1) L 0
```

Last update:

Send comments to: webmaster@cs.wwc.edu

Haskell Tutorial

Introduction

Haskell is a general purpose, purely functional programming language named after the logician Haskell B. Curry. It was designed in 1988 by a 15-member committee to satisfy, among others, the following constraints.

- It should be suitable for teaching, research, and applications, including building large systems.
- It should be freely available.
- It should be based on ideas that enjoy a wide consensus.
- It should reduce unnecessary diversity in functional programming languages.

It's features include higher-order functions, non-strict(lazy) semantics, static polymorphic typing, user-defined algebraic datatypes, type-safe modules, stream and continuation I/O, lexical, recursive scoping, curried functions, pattern-matching, list comprehensions, extensible operators and a rich set of primitive data types.

The Structure of Haskell Programs

A module defines a collection of values, datatypes, type synonyms, classes, etc. and *exports* some of these resources, making them available to other modules.

A Haskell *program* is a collection of modules, one of which, by convention, must be called `Main` and must export the value `main`. The *value* of the program is the value of the identifier `main` in module `Main`, and `main` must have type `Dialogue`.

Modules may reference other modules via explicit `import` declarations, each giving the name of a module to be imported, specifying its entities to be imported, and optionally renaming some or all of them. Modules may be mutully recursive.

The name space for modules is flat, with each module being associated with a unique module name.

There are no mandatory type declarations, although Haskell programs often contain type declarations. The language is strongly typed. No delimiters (such as semicolons) are required at the end of definitions - the parsing algorithm makes intelligent use of layout. Note that the notation for function application is simply juxtaposition, as in `sq n`.

Single line comments are preceded by ``--" and continue to the end of the line. For example:

```
succ n = n + 1  -- this is a successor function
```

Multiline and nested comments begin with **{-** and end with **-}**. Thus

```
{- this is a
     multiline
        comment -}
```

Lexical Issues

Haskell code will be written in ``typewriter font'' as in ```f (x+y) (a-b)`''. Case matters. Bound variables and type variables are denoted by identifiers beginning with a lowercase letter; types, constructors, modules, and classes are denoted by identifiers beginning with an uppercase letter.

Haskell provides two different methods for enclosing declaration lists. Declarations may be explicitly enclosed between braces { } or by the layout of the code.

For example, instead of writing:

```
f a + f b where { a = 5; b = 4; f x = x + 1 }
```

one may write:

```
f a + f b = where a = 5; b = 4
                f x = x + 1
```

Function application is curried, associates to the left, and always has higher precedence than infix operators. Thus ``f x y + g a b'' parses as ``((f x) y) + ((g a) b)''

Values and Types

All computation is done via the evaluation of *expressions*(syntactic terms) to yield *values*. Values are divided into disjoint sets called *types* --- integers, functions, lists, etc. Values are *first-class* objects. First-class values may be passed as arguments to functions, returned as results, placed in data structures, etc. Every value has a *type* (intuitively a type is a set of values). *Type expressions* are syntactic terms which denote *type values* (or just *types*). Types are not first-class in Haskell.

Expressions are syntactic terms that denote *values* and thus have an associated *type*.

Type System

Haskell is *strongly typed* --- every expression has exactly one ``most general'' type (called the principle type.

Types may be *polymorphic* --- i.e. they may contain *type variables* which are universally quantified over all types. Furthermore, it is always possible to statically infer this type. User supplied type declarations are *optional*

Pre-defined datatypes

Haskell provides several pre-defined data types: Integer, Int, Float, Double, Bool, and Char.

Pre-defined structured datatypes

Haskell provides for structuring of data through *tuples* and {\sl lists. Tuples have the form:

```
(e1, e2, ..., en) n >=  2
```

If e_i has type t_i then the tuple has type $(t_1, t_2, ..., t_n)$

Lists have the form: $[e_1, e_2, ..., e_n]$ where n >= 0 and every element e_i must have the same type, say t, and the type of the list is then [t]. The above list is equivalent to: $e_1:e_2:...:e_n:[]$ that is, ``:" is the infix operator for ``cons".

User Defined Types

User defined datatypes are done via a ```data`" declaration having the general form:

data *T* u_1 ... u_n = C_1 t_{11} ... t_{1k1}
| ...
| C_n t_{n1} ... t_{nkn}

where *T* is a *type constructor*; the u_i are *type variables*; the C_i are *(data) constructors*; and the t_{ij} are the *constituent types* (possibly containing some u_i).

The presence of the u_i implies that the type is *polymorphic* --- it may be instantiated by substituting specific types for the u_i.

Here are some examples:

```
data Bool = True | False
data Color = Red | Green | Blue | Indigo | Violet
data Point a = Pt a a
data Tree a = Branch (Tree a) (Tree a) | Leaf a
```

`Bool` and `Color` are *nullary type constructors* because they have no arguments. `True, False, Red,` etc are *nullary data constructors*. `Bool` and `Color` are *enumerations* because all of their data constructors are nullary. `Point` is a *product* or *tuple* type constructor because it has only one constructor; `Tree` is a *union* types; often called an algebraic data type.

Functions

Functions are first-class and therefore ``higher-order." They may be defined via declarations, or ``anonymously" via *lambda abstractions*. For example,

```
\x -> x+1
```

is a lambda abstraction and is equivalent to the function `succ` defined by:

```
succ x = x + 1
```

If ``x'' has type t_1 and ```exp`'' has type t_2 then `` `\x -> exp`'' has type $t_1 \to t_2$. Function definitions and lambda abstractions are ``curried'', thus facilitating the use of higher-order functions. For example, given the definition

```
add x y = x + y
```

the function `succ` defined earlier might be redefined as:

```
succ = add 1
```

The curried form is useful in conjunction with the function `map` which applies a function to each member of a list. In this case,

```
map (add 1) [1, 2, 3] => [2,3,4]
```

`map` applies the curried function `add 1` to each member of the list `[1,2,3]` and returns the list `[2,3,4]`.

Functions are defined by using one or more equations. To illustrate the variety of forms that function definitions can take are are several definitions of the factorial function. The first definition is based on the traditional recursive definition.

```
fac n = if n == 0 then 1
        else n*fac( n - 1)
```

The second definition uses two equations and pattern matching of the arguments to define the factorial function.

```
fac 0     = 1
fac (n+1) = (n+1)*fac(n)
```

The next definition uses two equations, pattern matching of the arguments and uses the library function `product` which returns the product of the elements of a list. It is more efficient then the traditional recursive factorial function.

```
fac 0     = 1
fac (n+1) = product [1..(n+1)]
```

The final definition uses a more sophisticated pattern matching scheme and provides error handling.

```
fac n | n <  0    = error "input to fac is negative"
      | n == 0    = 1
```

```
        | n >  0    = product [1..n]
```

The infix operators are really just functions. For example, the list concatenation operator is defined in the Prelude as:

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)
```

Since infix operators are just functions, they may be curried. Curried operators are called *sections*. For example, the first two functions add three and the third is used when passing the addition function as a parameter.

```
(3+)
(+3)
(+)
```

Block structure

It is also permitted to introduce local definitions on the right hand side of a definition, by means of a ``where" clause. Consider for example the following definition of a function for solving quadratic equations (it either fails or returns a list of one or two real roots):

```
quadsolve a b c | delta < 0  = error "complex roots"
                | delta == 0 = [-b/(2*a)]
                | delta > 0  = [-b/(2*a) + radix/(2*a),
                                -b/(2*a) - radix/(2*a)]
                  where
                  delta = b*b - 4*a*c
                  radix = sqrt delta
```

The first equation uses the builtin error function, which causes program termination and printing of the string as a diagnostic.

Where clauses may occur nested, to arbitrary depth, allowing Haskell programs to be organized with a nested block structure. Indentation of inner blocks is compulsory, as layout information is used by the parser.

Polymorphism

Functions and datatypes may be *polymorphic*; i.e., universally quantified in certain ways over all types. For example, the ``Tree" datatype is polymorphic:

```
data Tree a = Branch (Tree a) (Tree a) | Leaf a
```

``Tree Int" is type of trees of fixnums; ``Tree (Char -> Bool)" is the type of trees of functions mapping characters to Booleans, etc. Furthermore:

```
fringe (Leaf x)            = [x]
fringe (Branch left right) = finge left ++ fringe right
```

``fringe'' has type ``Tree a -> [a]'', i.e. ``for all types a, fringe maps trees of a into lists of a.

Here

```
id x = x
[] ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)
map f [] = []
map f (x:xs) = f x : map f xs
```

id has type a->a, (++) (append) has type: [a]->[a]->[a], and map has type (a->b)->[a]->[b]. These types are inferred *automatically*, but may optionally be supplied as *type signatures*:

```
id   :: a -> a
(++) :: [a] -> [a] -> [a]
map  :: (a->b) -> [a] -> [b]
```

Type synonyms

For convenience, Haskell provides a way to define *type synonyms* --- i.e. names for commonly used types. Type synonyms are created using *type declarations*. Examples include:

```
type String = [Char]
type Person = (Name, Address)
type Name   = String
data Address = None | Addr String
```

This definition of String is part of Haskell, and in fact the literal syntax "hello" is shorthand for:

```
['h','e','l','l','o']
```

Pattern Matching

We have already seen examples of pattern-matching in functions (fringe, ++, etc.); it is the primary way that elements of a datatype are distinguished.

Functions may be defined by giving several alternative equations, provided the formal parameters have different patterns. This provides another method of doing case analysis which is often more elegant than the use of guards. We here give some simple examples of pattern matching on natural numbers, lists, and tuples. Here is (another) definition of the factorial function, and a definition of Ackerman's function:

Accessing the elements of a tuple is also done by pattern matching. For example the selection functions on 2-tuples can be defined thus

```
fst (a,b) = a
snd (a,b) = b
```

Here are some simple examples of functions defined by pattern matching on lists:

```
sum [] = 0
sum (a:x) = a + sum x

product [] = 0
product (a:x) = a * product x

reverse [] = []
reverse (a:x) = reverse x ++ [a]
```

n+k -- patterns are useful when writing inductive definitions over integers. For example:

```
x ^ 0     = 1
x ^ (n+1) = x*(x^n)

fac 0 = 1
fac (n+1) = (n+1)*fac n

ack 0 n = n+1
ack (m+1) 0 = ack m 1
ack (m+1) (n+1) = ack m(ack (m+1) n)
```

As-patterns are used to name a pattern for use on the right-hand side. For example, the function which duplicates the first element in a list might be written as:

```
f (x:xs) = x:x:xs
```

but using an *as-pattern* as follows:

```
f s@(x:xs) = x:s
```

Wild-cards. A wild-card will match anything and is used where we don't care what a certain part of the input is. For example:

```
head (x:_)  = x
tail (_:xs) = xs
```

Case Expressions

Pattern matching is specified in the Report in terms of case expressions. A function definition of the form:

```
f p11 ... p1k = e1
...
f pn1 ... pnk = en
```

is semantically equivalent to:

```
f x1 ... xk = case (x1, ..., xk) of (p11, ..., p1k) -> e1
                                    ...
                                    (pn1, ..., pnk) -> en
```

Lists

Lists are pervasive in Haskell and Haskell provides a powerful set of list operators. Lists may be appended by the '++' operator. The operator '**' does list subtraction. Other useful operations on lists include the infix operator `:' which prefixes an element to the front of a list, and infix `!!' which does subscripting. Here are some examples

```
["Mon","Tue","Wed","Thur","Fri"] ++ ["Sat","Sun"] is
     ["Mon","Tue","Wed","Thur","Fri","Sat","Sun"]
[1,2,3,4,5] [2,4] is [1,3,5]
0:[1,2,3] is [0,1,2,3]
[0,1,2,3]!!2 is 2
```

Note that lists are subscripted beginning with 0. The following table summarizes the list operators.

Symbol	Operation
x:List	prefix an element to a list
List ++ List	concatenate two lists
List \\ List	list difference
List !! n	n-th element of a list n = 0..

Arithmetic sequences

There is a shorthand notation for lists whose elements form an arithmetic series.

```
[1..5]    -- yields [1,2,3,4,5]
[1,3..10] -- yields [1,3,5,7,9]
```

In the second list, the difference between the first two elements is used to compute the remaining elements in the series.

List Comprehensions

List comprehensions give a concise syntax for a rather general class of iterations over lists. The syntax is adapted from an analogous notation used in set theory (called ``set comprehension''). A simple example of a list comprehension is:

```
[ n*n | n <- [1..100] ]
```

This is a list containing (in order) the squares of all the numbers from 1 to 100. The above expression would be read aloud as ``list of all n*n such that n is drawn from the list 1 to 100". Note that ``n" is a local variable of the above expression. The variable-binding construct to the right of the bar is called a ``generator" - the ``<-" sign denotes that the variable introduced on its left ranges over all the elements of the list on its right. The general form of a list comprehension in Haskell is:

```
[ body | qualifiers ]
```

where each qualifier is either a generator, of the form: *var* `<-` exp, or else a filter, which is a boolean expression used to restrict the ranges of the variables introduced by the generators. When two or more qualifiers are present they are separated by commas. An example of a list comprehension with two generators is given by the following definition of a function for returning a list of all the permutations of a given list,

```
perms [] = [[]]
perms x  = [ a:y | a <- x; y <- perms (x
[a]) ]
```

The use of a filter is shown by the following definition of a function which takes a number and returns a list of all its factors,

```
factors n = [ i | i <- [1..n]; n `mod` i = 0 ]
```

List comprehensions often allow remarkable conciseness of expression. We give two examples. Here is a Haskell statement of Hoare's ``Quicksort" algorithm, as a method of sorting a list,

```
quicksort :: [a] -> [a]
quicksort []     = []
quicksort (p:xs) = quicksort [ x | x <- xs, x <= p ]
                   ++ [ p ] ++
                   quicksort [ x | x <- xs, x >  p ]
```

Here is a Haskell solution to the eight queens problem. We have to place eight queens on chess board so that no queen gives check to any other. Since any solution must have exactly one queen in each column, a suitable representation for a board is a list of integers giving the row number of the queen in each successive column. In the following program the function "queens n" returns all safe ways to place queens on the first n columns. A list of all solutions to the eight queens problem is therefore obtained by printing the value of (queens 8)

```
queens 0 = [[]]
queens (n+1) = [ q:b | b <- queens n; q <- [0..7]; safe q b ]
safe q b = and [ not checks q b i | i <- [0..(b-1)] ]
checks q b i = q=b!!i || abs(q - b!!i)=i+1
```

Lazy Evaluation and Infinite Lists

Haskell's evaluation mechanism is ``lazy", in the sense that no subexpression is evaluated until its value is required. One consequence of this is that is possible to define functions which are non-strict (meaning that they are capable of returning an answer even if one of their arguments is undefined). For example we can define a conditional function as follows,

```
cond True x y = x
cond False x y = y
```

and then use it in such situations as ``cond (x=0) 0 (1/x)".

The other main consequence of lazy evaluation is that it makes it possible to write down definitions of infinite data structures. Here are some examples of Haskell definitions of infinite lists (note that there is a modified form of the ``.." notation for endless arithmetic progressions)

```
nats = [0..]
odds = [1,3..]
ones = 1 : ones
nums_from n = n : nums_from (n+1)
squares = [ x**2 | x <- nums_from 0 ]
odd_squares xs = [ x**2 | x <- xs, odd x ]
cp xs ys = [ ( x, y ) | x <- xs, y <- ys ]     -- Cartesian Product
pyth n = [ ( a, b, c ) | a <- [1..n],          -- Pythagorean Triples
                         b <- [1..n],
                         c <- [1..n],
                         a + b + c <= n,
                         a^2 + b^2 == c^2 ]
squares = [ n*n | n <- [0..] ]
fib = 1:1:[ a+b | (a,b) <- zip fib ( tail fib ) ]
primes = sieve [ 2.. ]
         where
         sieve (p:x) = p : sieve [ n | n <- x, n `mod` p > 0 ]
repeat a = x
           where x = a : x
perfects = [ n | n <- [1..]; sum(factors n) = n ]
primes = sieve [ 2.. ]
         where
         sieve (p:x) = p : sieve [ n | n <- x; n mod p > 0 ]
```

The elements of an infinite list are computed ``on demand", thus relieving the programmer of specifying ``consumer-producer" control flow.

One interesting application of infinite lists is to act as lookup tables for caching the values of a function. For example here is a (naive) definition of a function for computing the n'th Fibonacci number:

```
fib 0 = 0
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

This naive definition of ``fib" can be improved from exponential to linear complexity by changing the recursion to use a lookup table, thus

```
fib 0 = 1
fib 1 = 1
```

```
fib (n+2) = flist!!(n+1) + flist!!n
            where
            flist = map fib [ 0.. ]
```

alternatively,

```
fib n = fiblist !! n
        where
           fiblist = 1:1:[a+b| (a,b) <- zip fiblist (tail fiblist) ]
```

Another important use of infinite lists is that they enable us to write functional programs representing networks of communicating processes. Consider for example the Hamming numbers problem - we have to print in ascending order all numbers of the form 2^a*3^b*5^c, for a,b,c>=0. There is a nice solution to this problem in terms of communicating processes, which can be expressed in Haskell as follows

```
hamming = 1 : merge (f 2) (merge (f 3) (f 5))
        where
        f a = [ n*a | n <- hamming ]
        merge (a:x) (b:y) = a : merge x (b:y), if a<b
                          = b : merge (a:x) y, if a>b
                          = a : merge x y,     otherwise
```

Abstraction and Generalization

Haskell supports abstraction in several ways:

- where expressions
- function definitions
- data abstraction
- higher-order functions
- lazy evaluation

Data Abstraction

Haskell permits the definition of abstract types, whose implementation is hidden from the rest of the program. To show how this works we give the standard example of defining stack as an abstract data type (here based on lists):

```
module Stack (StackType, push, pop, top, empty)
where
data StackType a = Empty | Stk a (StackType a)
push x s = Stk x s
pop (Stk _ s) = s
top (Stk x _) = x
empty = Empty
```

The constructors `Empty` and `Stk`, which comprise ``the implementation'' are not exported, and thus hidden

outside of the module. To make the datatype *concrete*, one would write:

```
module Stack (StackType(Empty,Stk), push, ...)
...
```

Higher-Order Functions

Haskell is a fully higher order language --- functions are first class citizens and can be both passed as parameters and returned as results. Function application is left associative, so `f x y` it is parsed as `(f x) y`, meaning that the result of applying f to x is a function, which is then applied to y.

In Haskell every function of two or more arguments is actually a higher order function. This permits partial parameterization. For example `member` is a library function such that `member x a` tests if the list `x` contains the element a (returning True or False as appropriate). By partially parameterizing member we can derive many useful predicates, such as

```
vowel = member ['a','e','i','o','u']
digit = member ['0','1','2','3','4','5','6','7','8','9']
month = member ["Jan","Feb","Mar","Apr","May","Jun",
                "Jul","Aug","Sep","Oct","Nov","Dec"]
```

As another example of higher order programming consider the function `foldr`, defined by

```
foldr op k [] = k
foldr op k (a:x) = op a (foldr op k x)
```

All the standard list processing functions can be obtained by partially parameterizing `foldr`. Here are some examples.

```
sum = foldr (+) 0
product = foldr (*) 1
reverse = foldr postfix []
          where postfix a x = x ++ [a]
```

Abstract data types

Overloading

Type Classes

I/O

Arrays

Types

Simple Types

Haskell provides three simple types, boolean, character and number.

Types	Values
Bool	True, False
Char	the ASCII character set
Int	minInt, ..., maxInt
Integer	arbitrary precision integers
Float	floating point, single precision
Double	floating point, double precision
Bin	binary numbers
String	list of characters
Funtions	lambda abstractions and definitions
Lists	lists of objects of type T
Tuples	Algebraic data types
Numbers	integers and floating point numbers

Composite Types

Haskell provides two composite types, lists and tuples. The most commonly used data structure is the list. The elements of a list must all be of the same type. In Haskell lists are written with square brackets and commas. The elements of a tuple may be of mixed type and tuples are written with parentheses and commas. Tuples are analogous to records in Pascal (whereas lists are analogous to arrays). Tuples cannot be subscripted - their elements are accessed by pattern matching.

Type	Representation	Values
list	[*comma separated list*]	user defined
tuple	(*comma separated list*)	user defined

Here are several examples of lists and a tuple:

```
[]
["Mon","Tue","Wed","Thur","Fri"]
[1,2,3,4,5]
("Jones",True,False ,39)
```

Type Declarations

While Haskell does not require explicit type declarations (the type inference system provides static type checking), it is good programming practice to provide explicit type declarations. Type declarations are of the form:

e :: t

where *e* is an expression and *t* is a type. For example, the factorial function has type

```
fac :: Integer -> Integer
```

while the function `length` which returns the length of a list has type

```
length :: [a] -> Integer
```

where `[a]` denotes a list whose elements may be any type.

Type Predicates

Since Haskell provides a flexible type system it also provides type predicates check on the type of an object. Haskell provides three type predicates.

Predicate	Checks if
digit	argument is a digit
letter	argument is a letter
integer	argument is an integer

Expressions

Arithmetic Operators

Haskell provides the standard arithmetic operators.

Symbol	Operation
+	addition
-	subtraction
*	multiplication
/	real division
div	integer division
mod	modulus
^	to the power of

Tuples (records)

The elements of a tuple are accessed by pattern matching. An example is given in a later section.

Logical Operators

The following table summarizes the logical operators.

Symbol	Operation
not	negation
&&	logical conjunction
||	logical disjunction

Boolean Predicates

The following table summarizes the boolean operators.

Symbol	Operation
==	equal
/=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Modules

At the top level, a Haskell program consists of a collection of *modules*. A Module is really just one big declaration which begins with the keyword `module`. Here is an example:

```
module Tree ( Tree(Leaf,Branch), fringe ) where

data Tree a                 = Leaf a | Branch ( Tree a ) ( Tree a )

fringe :: Tree a -> [a]
fringe ( Leaf x )           = [x]
fringe ( Branch left right ) = fringe left ++ fringe right

y
```

Appendix

The following functions are part of the Haskell standard prelude.

BOOLEAN FUNCTIONS

&&
: and

||
: or

not
: and

otherwise
: is equavalent to true.

CHARACTER FUNCTIONS

ord

chr

isAscii, **isControl, isPrint, isSpace, isUpper, isLower, isAlpha, isDigit, isAlphanum, toUpper, toLower**

NUMERIC FUNCTIONS

subtract

gcd, lcm

x^n
: positive exponents only

x^^n
: positive and negative exponents

truncate, round, ceiling, floor

SOME STANDARD FUNCTIONS

fst (x, y)
: = x

snd (x, y)
: = y

(f.g) x
: = f(g x) -- function composition

flip f x y
: = f y x

until p f x
: yields the result of applying f until p holds

==, /=, <, <=, >=, >

max x y, min x y

+, - , *

negate,abs, signum,fromInteger

toRational

`div`, `rem`, `mod`

even, odd

divRem

toInteger

Operators: +,-,*,/,^

minInt, maxInt

subtract

gcd

lcm

truncate, round, ceiling, floor

pi

exp,log,sqrt

****,logBase**
sin,cos,tan
asin,acos,atan
sinh,cosh,tanh
asinh,acosh,atanh
Prelude `PreludeList` Haskell provides a number of operations on lists. Haskell treats strings as lists of characters so that the list operations and functions also apply to strings.

head, tail
: extract the first element and remaining elements (respectively) of a non-empty list.

last, init
: are the duals of head and tail, working from the end of a finite list rather than the beginning.

null, (++), (\ \):
: test for the null list, list concatenation (right-associative), and list difference (non-associative) respectively.

length
: returns the length of a list

!!
: is the infix list subscript operator; returns the element subscripted by the index; the first element of the list has subscript 0.

length
: returns the length of the list.

map
: applies its first argument to each element of a list (the second argument); `map (+2) [1,2,3]` is `[3,4,5]`

filter
: returns the list of elements of its second argument which satisfy the first argument; `filter (<5) [6,2,5,3]` is `[2,3]`

partition
: takes a predicate and a list and returns a pair of lists, those elements of the argument list that satisfy and do not satisfy the predicate

foldl, foldl1
scanl, scanl1
foldr, foldr1
scanr, scanr1
iterate
repeat x
: is the infinite list xs = x:xs

cycle xs
: is the infinite list xs' = xs ++ xs'

take n xs
: is the list of the first n elements of xs

drop n xs
: is the list xs less the first n elements

splitAt n xs
: is the pair of lists obtained from xs by spliting it in two after the n^{th} element

takeWhile
dropWhile
span
break
lines, unlines

words, unwords
nub
reverse
and, or
any, all
x elem xs, x notElem xs
are the tests for list membership
sum, product
sums, products
maximum, minimum
concat
transpose
zip, zip3--zip7
zipWidth, zipWidth3--zipWidth7
Prelude
PreludeArray
Prelude
PreludeText
Prelude
PreludeIO

References

Bird and Wadler
Introduction to Functional Programming Prentice Hall, New York, 1988.
Field and Harrison
Functional Programming Addison-Wesley, Workingham, England, 1988.
The Yale Haskell Group
The Yale Haskell Users Manual Beta Release 1.1-0. May 1991.
Hudak, Paul et al.
Report on the Programming Language Haskell Version 1.1 August 1991.
Peyton Jones, S. L.
The Implementation of Functional Programming Languages Prentice-Hall, englewood Cliffs, NJ, 1987.

Last update:

Send comments to: webmaster@cs.wwc.edu

PCN Tutorial

Overview

- Principles
  - First-Class Concurrency
  - Controlled nondeterminism
  - Compositionality
  - Mapping Independence
- PCN realization: The execution of a parallel program forms a set of concurrently executing lightweight processes (threads) which communicate and synchronize by reading and writing shared definitional variables. Individual threads may apply the usual sequential programming techniques of state change and sequencing. Execution is deterministic, unless specialized operators are invoked to make nondeterministic choices.
  - Definitional variables -- untyped, used for communication and synchronization
  - Mutable variables -- typed, used in squential threads for state information and for communication.
  - Concurrent composition -- for specification of concurrency and when combined with recursion, dynamic process creation.
  - Nondeterministic choice -- specification of nondeterministic choice
  - Encapsulation of state change -- restriction to a single thread

PCN Syntax

- Constants: PCN uses the same ANSI C conventions for character, integer, double precision floating point and string constants. (Strings: "-------")
- Data Types: character, integer, and double precision floating point (`char, int, double`) are as in C. One dimensional arrays of these data types are supported. There is a complex data type -- the `tuple`.
- Expressions: Arithmetic expressions as in C. User defined functions may not be called in guards.
- Variable Names: (as in C)
- Comments: (as in C)
- Procedures: *heading declarations block* All arguments are passed by reference.
- Functions: `function` *heading declarations block* the block must contain calls to the primitive `return(r)` to specify a return value `r`. The return value of a function must be a definitional variable.
- Delimiters: The blocks within a composition must be separated by either a comma or a semicolon.
- Declarations: *type variable_names;* Declarations are used only for mutable variables -- definitional variables are not declared.

Sequential Composition and Mutable Variables

Example.

`{; block_0, ..., block_n }`

variable := *expression*

Context.

Sequential Composition is used to provide sequencing of updates to mutable variables and sequencing I/O operations.

Problem.

Synchronization with mutable variables cannot be achieved without complex locking mechanisms.

Solution.

Mutable variables should be snapshot when passed and only modified in one procedure.

Examples

```
swap( array, i, j )
int array[], i, j, temp;
{; temp := array[i],
array[i]:= array[j],
array[j] := temp
}

swaptest()
int a[3], i, j;
{; a[0] := 0, a[1] := 1, a[2] := 2,
    i := 1, j := 2,
    stdio:printf("Before: %d %d %d\n",{a[0],a[1],a[2]},_),
    swap( a, i, j ),
    stdio:printf("After: %d %d %d\n",{a[0],a[1],a[2]},_),
}
```

Parallel Composition and Definitional Variables

Name.

Parallel Composition

Example.

{|| *block_0, ..., block_n* }

variable = *expression*

"_" represents a unique anonymous definition variable.

Context.

Parallel Composition is used to expose *opportunities* for concurrent execution while sequential composition constrains execution order.

Problem.

Synchronization with mutable variables cannot be achieved without complex locking mechanisms.

Solution.

Name.

Definitional Variable

Example.

variable = *expression*

Context.

Definitional Variables are used for communication and synchronization and have the following properties:

- Have an initial value -- a special "undefined" value
- "Read" operations block until the variable is given a value
- Are defined "written" by the definition operator =
- Once defined, cannot be modified
- Can be shared by procedures in a parallel composition
- Are not explicitly declared
- Can take on values of type `char, int, double, tuple`

Problem.

Solution.

Choice Composition

Example.

`{? guard_0 -> block_0, ..., guard_n -> block_n }` -- each guard is a sequence of one or more tests which include

- a < b, a > b, a <= b, a >= b, a == b, a != b: arithmetic comparison tests
- `int(a), char(a), double(a), tuple(a)` : type tests
- `data(a)` : synchronization test
- tuple1 ?= tuple2 : tuple match
- `default` : default action

`guard_i -> block_i` is called an *implication*

Context.

Choice Composition is used

- to choose between alternatives
- to synchronize processes
- to provide nondeterministic choice

Problem.

Solution.

The operational semantics of Choice composition are:

- Evaluate each guard left to right
- If any test suspends/fails, guard suspends/fails
- If all tests succeed, guard succeeds
- If all guards fail, process terminates
- If no guards succeed and some suspend, process suspends.
- If some guards succeed, execute one implication body

Examples

```
Choosing between disjoint alternatives.
max(x, y, z)
{? x >= y -> z = x,
   x < y  -> z = y
}

Synchronization is required if x or y is undefined.
max(x, y, z)
{? x >= y -> z = x,
  default -> z = y
}

Nondeterministic choice (alternatives are non-disjoint).
max(x, y, z)
{? x >= y -> z = x,
   y >= x -> z = y
}

switch(sensor1, sensor2, alarm)
{? data(sensor1) -> alarm = 1,
   data(sensor2) -> alarm = 2
}
```

Repetitive Actions

Name.
Quantification
Example.
{ op i over low .. high :: block }
where block is executed once for each i in the range low..high either concurrently (if op = ||) or sequentially (if op = ;).
Context.
Quantification is useful when specifying iterative computation involving mutable variables or ports
Problem.
Solution.

Name.
Recursion
Example.
Context.
Problem.
Solution.

Tuples

Example.
{ term_0, ..., term_{k-1} }, (k <= 0) where term_i are definitional data structures.
Context.
guard tests: ==, ?=, !=; access: t[i] to access the i-th element; make_tuple(n, tuple) makes a definitional tuple of arity n
Problem.
Tuple match does *not* perform unification

Solution.

Lists

Name.
List (a two-tuple with special notation)
Example.
[], [x_0, ..., x_n], [x_0, ..., x_i | R]
Context.
Problem.
Solution.

programs: list length, buildlist, listadd,

Stream Communication

Name.
Producer-Consumer
Example.

```
 { || Producer(Stream), Consumer(Stream) }

Producer( Stream )
 {|| Produce(Item), Stream = [Item|StreamP], Producer(StreamP)}

Consumer( Stream )
 {? Stream ?= [Item|StreamP] -> {|| Consume(Item), Consumer( StreamP ) }
}
```

Context.
Problem.
Stream communication terminates when the stream closes i.e. the stream = [].
Solution.

Name.
Broadcast communication (one to many)
Example.

```
{ producer( S ), consumer(s), ..., consumer(s) }
```

Context.
Problem.
Solution.

Name.
Many to one
Example.

```
 {|| producer( s1 ), producer( s2 ), consumer( stream ),
instream = [{"merge", s1}, {"merge",s2}],
sys:merger(instream, stream)}
```

Context.
Problem.
Solution.

Name.
Two way communication
Example.

```
{|| query( qr ), response( qr ) }
query( qr )
   { || qr = { theQuery, theResponse }, ... }
query( qr )
   { ? qr ?= { theQuery, theResponse } -> ... theResponse = ... }
```

Context.
Problem.
Solution.
Two streams or query-response pair

Name.
Bounded-Communication
Example.

```
{|| Buffer = [S1,S2,S3|End],
             producer( Buffer ), consumer( Buffer, End )}

producer( Buffer )
 {? Buffer ?= [Slot|B1] -> {|| Slot = ..., producer( B1 ) }
 ...
 }

consumer( Buffer, End )
 {? Buffer ?= [Item|B1] -> {|| ..., End = [Slot|E1], consumer( B1, E1 ) }
 ...
 }
```

Context.
Problem.
Solution.

Examples

Hello World

```
main(argc, argv, exit_code)
{; stdio:printf("Hello world.\n", {}, d),
   exit_code = 0
}
```

Script to Compile, Link, & Execute

```
pcncomp -c hello.pcn

pcncomp hello.pam -o hello -mm hello -mp main

hello
```

Input Process Output

```
#include <pcn_stdio.h>

output(C)
   {; stdio:printf("The circumference is: %d\n", {C}, d),
      exit_code = 0
   }

input(R)
   {; stdio:printf("Enter a radius: ", {}, _),
      stdio:scanf("%d%d", {R}, _)
   }

main(argc, argv, exit_code)
   {|| C=2*3.14159*R, input(R), output(C)}
```

Script to Compile, Link, & Execute

```
pcncomp -c circle.pcn

pcncomp circle.pam -o circle -mm circle -mp main

circle
```

Arithmetic and Lists

```
#include <pcn_stdio.h>

/*****************************************************
     Arithmetic
*****************************************************/

minimum(x,y,result)
{? x >= y -> result = y,
   x <= y -> result = x
}

sum(x,y,result)
{? y==0 -> result = x,
   y>0  -> sum(x+1,y-1,result)
}

power(x,y,result)    /* result = x to the y power */
{? y==0 -> result = 1,
   y>0  -> {|| result = x*r1, power(x,y-1,r1)}
}
```

```
factorial(n,result)  /* result = n! */
{? n==0 -> result = 1,
   n>0  -> {|| result=n*r1, factorial(n-1,r1)}
}

function f(n)
{? n == 0 -> {|| R = 1, return(R)},
   n > 0  -> {|| R = n*f(n-1), return(R)}
}

/*****************************************************
     Lists
*****************************************************/

generator(n,L)   /* L = [n, n-1,...,1] */
{? n == 0 -> L = [],
   n > 0  -> {|| L = [n|L1], generator(n-1,L1)}
}

count(Ls,cnt)    /* cnt = the length of list Ls */
{? Ls ?= []      -> cnt = 0,
   Ls ?= [_|Ls1] -> {|| cnt = cnt1+1, count(Ls1,cnt1)}
}

list_sum(Ls,result) {|| sumlist(Ls,0,result)}
/* result = sum of the elements in Ls */

sumlist(Ls,n,result)
{? Ls?=[]      -> result = n,
   Ls?=[x|Ls1] -> sumlist(Ls1,n+x,result)
}

/*****************************************************
     Test Harness
*****************************************************/

main(argc, argv, exit_code)
{; stdio:printf("Chapter 3.\n", {}, _),
   stdio:printf("Enter two numbers: ", {}, _),
   stdio:scanf("%d%d", {a,b}, _),


/* Arithmetic */
   minimum( a,b, rmin ),
   stdio:printf("The minimum of %d and %d is: %d\n", {a,b,rmin}, _),

   sum(a,b,rsum),
   stdio:printf("The sum of %d and %d is: %d\n", {a, b, rsum}, _),

   power( a,b, rpower ),
   stdio:printf("%d to the %d power is: %d\n", {a,b,rpower}, _),

   factorial(a,rfac),
   stdio:printf("%d! is: %d\n", {a, rfac}, _),
```

```
   stdio:printf("%d! is: %d\n", {4,f(4)},  _),


/* Lists */
   {|| generator(a,Lst), count(Lst,n), list_sum(Lst,rsum)},

   stdio:printf("The list is: %t\n", {Lst}, _),
   stdio:printf("The list is of length: %d\n", {n}, _),
   stdio:printf("and the sum of its elements is: %d\n", {rsum}, _),

   exit_code = 0
}
```

Compile, Link, Execute

```
pcncomp -c demo.pcn

pcncomp demo.pam -o demo -mm demo -mp main

demo
```

Assembler

```
#include <pcn_stdio.h>

/*****************************************************
     An Assembler
     Ls: a list of assembly code
     As: an intermediate list of
*****************************************************/

assemble(Ls,Os) {|| asm(Ls,As), resolve(0,As,Os) }

asm(Ls,Cb)
{? Ls ?= [store(a,v)|Ls1] -> {|| Cb = [{_,1,a,v,0}|Cm], asm(Ls1,Cm)},
   Ls ?= [load(v,b) |Ls1] -> {|| Cb = [{_,2,v,b,0}|Cm], asm(Ls1,Cm)},
   Ls ?= ["halt"    |Ls1] -> {|| Cb = [{_,3,0,0,0}|Cm], asm(Ls1,Cm)},
   Ls ?= [jump(a)   |Ls1] -> {|| Cb = [{_,4,a,0,0}|Cm], asm(Ls1,Cm)},
   Ls ?= [label(a)  |Ls1] -> {|| {? Cb ?= [{na,_,_,_,_}|_] ->  a=na},
                                 asm(Ls1,Cb)
                              },
   Ls ?= []               -> Cb = []
}

resolve(n,Ls,Os)
{? Ls ?= [{a,p,q,r,s}|Ls1] -> {|| a=n,
                                  Os=[p,q,r,s|Os1],
                                  resolve(n+1,Ls1,Os1)
                               },
   Ls ?= []                 -> Os = []
}

/*****************************************************
     Test Harness
```

```
*****************************************************/

main(argc, argv, exit_code)
{; stdio:printf("Assembler Demo.\n", {}, _),

   As = [load(1,2),label(x),store(3,4),jump(x),"halt"],
   stdio:printf("Assembly Code: %lt\n", {As}, _),
   assemble(As,Os),
   stdio:printf("Machine Code: %t\n", {Os}, _),

   exit_code = 0
}
```

Compile, Link, Execute

```
pcncomp -c asm.pcn

pcncomp asm.pam -o asm -mm asm -mp main

asm
```

Static Pipeline Processing

```
#include <pcn_stdio.h>

/*****************************************************
     Pipeline -- Static Process Set

        Input, Process, Output

*****************************************************/

input(L) {|| generator(7,L)}

generator(n,L)   /* L = [n, n-1,...,1] */
{? n == 0 -> L = [],
   n > 0  -> {|| L = [n|L1], generator(n-1,L1)}
}

process(L,O)
{? L ?= [x|L1] -> {|| O = [2*x|O1], process(L1,O1)},
   L == []     -> O = L
}

output(L)
{? L ?= [x|L1] -> {; stdio:printf("%d ", {x}, _),
                     output(L1)
                  },
   default     -> stdio:printf("%\n", {}, _)
}


/*****************************************************
     Test Harness
*****************************************************/
```

```
main(argc, argv, exit_code)
{; stdio:printf("Pipeline Demo: Input, Process, Output.\n", {}, _),

   {|| input(L), process(L,O), output(O) },

   exit_code = 0
}
```

Merge Sort

```
merge(A, i,j,k,l, B)
{? i<=j, k<=l -> {? A[i]<=A[k] -> {|| B[m]=A[i], merge(A,i+1,j,k,l,m+1,B),
                     A[i]>=A[k] -> {|| B[m]=A[k], merge(A,i,j,k+1,l,m+1,B)}
   i> j, k<=l -> {|| B[m]=A[k], merge(A,i,j,k+1,l,m+1,B)}
   i<=j, k> l -> {|| B[m]=A[i], merge(A,i+1,j,k,l,m+1,B)}
   default    -> skip
}

sort(A, i, j, B)
{|| m = (i+j) div 2,
    sort(A, i,   m, B),
    sort(A, m+1, j, B),
    merge(B, i, m, m+1, j A)
}
```

Pipeline Sort

```
#include <pcn_stdio.h>

/*****************************************************************

                  Pipeline -- Dynamic Process Set

          Pipeline Sort  -- values flow through the processes

*****************************************************************/

generator(n,L)   /* L = [n, n-1,...,1] */
{? n == 0 -> L = [],
   n > 0  -> {|| L = [n|L1], generator(n-1,L1)}
}

/*****************************************************************
                             Sort
*****************************************************************/

sort(In,Sorted)
{|| pipe_end(In, Sorted )}

pipe_end( In, Out )
{? In ?= [] -> Out = [],
   In ?= [y|In1] -> {|| cell( y, Lin, Lout, Rin, Rout ),
                        pipe_end( In2, Out1 ),
```

```
                        Lin = In1, Lout = Out,
                        In2 = Rout, Out1 = Rin
                    }
    }

    cell(x,Lin,Lout,Rin,Rout)
    {? Lin ?= [y|Lin1], y < x  -> {|| Rout = [x|Rout1],
                                      cell( y, Lin1,  Lout, Rin, Rout1 )
                                   },
       Lin ?= [y|Lin1], y >= x -> {|| Rout = [y|Rout1],
                                      cell( x, Lin1,  Lout, Rin, Rout1 )
                                   },
       Lin ?= []               -> {|| Lout = [x|Rin], Rout = []},
    }



    /*****************************************************************
                            Test Harness
    *****************************************************************/

    main(argc, argv, exit_code)
    {; stdio:printf("Pipeline Sort.\n", {}, _),

    /*   {|| generator(7,L), sort(L,SL)}, */
       {|| L=[5,7,3,6], sort(L,SL)},
       stdio:printf("%t is %t sorted\n", {SL,L}, _),

       exit_code = 0
    }
```

Process-Machine Mapping

```
    function node(i) {|| return ( i%nodes() ) }

    work()
    char str[30]; int k;
    {; host(str,k),
       stdio:printf("Node %s reporting.\n", {str}, _)
    }


    main(argc, argv, exit_code)
    {; stdio:printf("Machine topology/nodes/location. %lt %d %d\n",
       {topology(), nodes(), location()}, _),

       {|| i over 0 .. nodes()-1 :: work()@node(i)},
       exit_code = 0
    }
```

Foreign Program

```
    #include <stdlib.h>
```

```
/* C code */

void host(str,k)
   char *str; int *k;
{ int i;
  i = gethostname(str,k);
}
```

Compilation, Linking, & Execution

```
pncomp -c net.pcn

pncomp -c host.c

pncomp net.pam host.o -o net -mm net -mp main

net -pcn -nodes adams:baker:glacier:grandcoolie:hood:jefferson:\
johnday:polaris:radar:rainier:shasta:sthelens
```

Operating Systems

Single Processor Kernel

```
#include <pcn_stdio.h>

main(argc, argv, exit_code)
{; stdio:printf("\n\nSingle processor kernel demo\n\n", {}, _),
   JobQueue = [{0,100},{1,300},{2,75},{3,400},{5,30},{6,176}],
   kernel( JobQueue ),
   exit_code = 0
}


/*****************************************************************
               Kernel - single processor kernel

    JobQueue  - Queue of new jobs
*****************************************************************/

kernel( JobQueue )
{|| FreeList = [_,_,_,_|_],
    scheduler( ReadyList, RunList, JobQueue, FreeList ),
    dispatcher(ReadyList, RunList),
}


/*****************************************************************
               Scheduler - submit jobs to the dispatcher

    ReadyList - Queue of jobs for the dispatcher
    RunList   - Queue of jobs that have been run
    JobQueue  - Queue of new jobs
    FreeList  - Available discriptors for new jobs
*****************************************************************/
```

```
scheduler( ReadyList, RunList, JobQueue )
{|| scheduler1( ReadyList, RunList, JobQueue, FreeList )}

scheduler( ReadyList, RunList, JobQueue, FreeList )
{? JobQueue ?= [J|JQ], FreeList ?= [_|FL] ->
                              {|| ReadyList = [J|RdyL],
                                  scheduler( RdyL, RunList, JQ, FL)
                              },
   RunList ?= [{Id,n}|RnL] -> {? n >  0 -> {|| ReadyList = [{Id,n}|RdyL],
                                               scheduler( RdyL,
                                                          RnL,
                                                          JobQueue,
                                                          FreeList)
                                           },
                                 n <= 0 -> {|| FL = [_|FreeList],
                                               scheduler( ReadyList,
                                                          RnL,
                                                          JobQueue,
                                                          FL )
                                           }
                              },

}


/*****************************************************************
               Dispatcher - Prepare processes for execution

    InQ - Queue of jobs to be run
    OutQ - Queue of jobs that have been run
*****************************************************************/

dispatcher( InQ, OutQ )
{? InQ ?= [P|IQ] -> {;  cpu( P, 15, R ),
                        OutQ = [R|OQ],
                        dispatcher( IQ, OQ )
                    }
}


/*****************************************************************
               CPU Process - execute a job for a time slice

    JobIn    - Discriptor of job to be run
    Quantum  - Time quantum for this job
    JobOut   - Discriptor of job after running for time <= Slice
*****************************************************************/

cpu( JobIn, Quantum, JobOut )
{? JobIn ?= {Id, n} -> {; JobOut = {Id, n - Quantum},
                          stdio:printf("Running Job: %d\n", {Id}, _)
                       }
}
```

References

Chandy, K. Mani and Taylor, Stephen (1992)
An Introduction to Parallel Programming Jones and Bartlett, Boston, MA.
Foster, Ian and Tuecke, Steven (1993)
Parallel Programming with PCN Argonne National Laboratory, Chicago, IL.

Last update:

Send comments to: webmaster@cs.wwc.edu

Introduction to Software Engineering

Anthony A. Aaby

Goal: to provide reference material for the introductory sequence based on SE1-SE5

Preface

Table of Contents

Verification and Validation

1. Correctness
2. Debugging
3. Testing

Miscellaneous

1. Logic Programming Systems
2. Logic Programming Schemata
3. Parallel Systems
4. A Formal Methodology
5. Systems Development
6. Tools

Appendix

- Errors
- Examples
- Software Components
- An Imperative Langauge
- A Unified Pradigm Langauge

Preface

Why we wrote this book.

Graduates of a program in computer science should be able to prove their programs correct and we believe that students should begin to learn how to do this from their first computer science course. Skill in proving programs correct (like most complex skills) are best developed gradually by proceeding from a simple and intuitive level to a more complex and explicit level. We wanted to lay a foundation for this skill beginning with the first computer science class; to incorporate assertions and axiomatic semantics in a natural and systematic way into our introductory course. In particular, we wanted to firmly ground our student in three key concepts.

- The notion of state and that imperative programs are developed by carefully planning changes to the state.
- The semantics of programming language constructs reflect the relationships among the program variables.
- Assertions are an important aid in the program development process.

We leave for later courses the activity of formal proofs of correctness. Finding no text to support our needs (most CS 1 texts don't mention this formal side), we decided do develop a supplementary text which could support our goals reguardless of what language or main text we chose. In addition, we want to encourage others to view programming in a more formal light.

Who should use this book?

Instructors and students in introductory computer science courses and practicing programmers who want a gentle, intuitive yet more formal basis for programming.

What is this book about?

This book is about programming as an intellectual activity. We begin with the notion of state and how programming commands change the state. Assertions are then introduced to provide documentation and are related to the semantics of the program. Later assertions are used to assist in understanding and in verifying the correctness of programs. Chapter 1 introduces the notion of state and programming commands as state modifiers. Chapter 2 introduces the notion of an assertion and relates it to the semantics of programming commands. Straight line programs are used for illustration. Chapter 3 introduces pre- and post-conditions as assertions for procedures and functions. Chapter 4 introduces the semantics of selection. Chapter 5 introduces the semantics of repetition and variant and

invariant assertions. Chapter 6 introduces methods for dealing with data types such as arrays and lists. Chapter 7 shows how programs may be developed from specifications in the form of pre- and post-conditions.

How to use this book.

To the Instructor

We believe that the concepts of assertions, pre- and post-conditions, variants, and invariants should be introduced informally and as a natural part of the programming process. Since students learn largely through examples and by example, it is necessary for the the instructor to consistently use assertions when presenting and discussing algorithms and programs. It is our experience that it is not necessary to make a big deal of assertions, students will begin using assertions if they see their instructors use them. And that it is not necessary to explicitly devote lecture time to introducing these concepts in an introductory level course.

To the Student

Students should use this book to gain additional insight into the programming process.

{AA, JM}

Key ideas

- The imperative paradigm -- a computation is characterized by a finite sequence of states. Objects have a state (name-value association). The assignment operator is used to change an objects state (associate a different value with the name of the object).
- Programming constructs:
    - Control structures
    - Data structures
- Software Engineering
    - Design
        - Specification
        - Top Down vs Bottom up design
        - Object-oriented design
        - Stepwise Refinement
        - Correctness and Verification
            - assertions
            - Procedure, function, and constuct --- pre- \& post-conditions
            - Loop --- invariant, \& variant
    - Implementation
        - Stubs
        - Animation

        - Harnesses
        - Regression testing
    - Tool kit -- standard algorithms and techniques

Chapter layout:

- Standard Practices
- Examples
- Software Engineering

Send comments to: webmaster@cs.wwc.edu

Introduction

The Imperative Model of Computation

- State Sequence [$S_{initial} = S_0, S_1, \ldots S_n = S_{final}$]
- data types
- expressions
- assignment, read, write
- control structures --- sequence; selection; repetition
- abstraction and generalization

History of Imperative Programming

Dates	Language	Programming Paradigm
	Machine Language	
	Assembly Language	
1950-1970	Fortran	Imperative Programming
1970-1980	Pascal	Structured Programming
1980-1990	Small-Talk	Object-Oriented Programming
1990-??	Linda, SR	Parallel and distributed programming

\part{Control}

PR: intro prog lang

Overall program structure

\bex Simple pascal program: Variable declarations; Assignment command; Expressions; input; output \eex

SE: software engineering

- Top-down design -- bottom-up implementation
- Comments
- Programming Template: Prompted Input, Labeled output

Straight Line Programs

PR: intro prog lang

Data types:
boolean, character, integer, real
Language:
declarations, literals, constants, variables, expressions, built-in functions, assignment, input, output, sequencial composition
Algorithms and techniques:
Interfaces

Basic type declarations

Arithmetic Operators and assignment

\bex Stick man \eex \bex Circle -- radius, circumference, area \eex

SE: software engineering

\bdf Actions occuring in a linear order illustrate the *sequence* control structure. \edf

When to use sequential composition

Template: Prompt--Read

Template: Input--Process--Output

Programs with Abstraction

PR: intro prog lang

Procedures, functions and parameters

The standard functions

units

SE: software engineering

\bdf *Abstraction* is the use of a name in place of the actual object. \edf \bdf *Generalization* is the use of parameters to widen the scope of applicability. \edf \bdf *Scope* is the range of visibility of an object. \edf \bex The {\sf StickPerson} program illustrates the concepts of abstraction and scope.

```
program StickPerson ( input, output );

procedure DrawHead;
...
procedure DrawArms;
...
```

```
procedure DrawBody;
...
procedure DrawLegs;
...
procedure DrawPerson;
   begin
      DrawHead;
      DrawArms;
      DrawBody;
      DrawLegs
   end;

begin
   DrawPerson;
end.
```

\eex The {\sf Circle} program illustrates abstraction and generalization.

```
program Circle ( input, output );

const
   pi = 3.14;

var
   Radius : real;

Procedure GetRadius ( var R : real );
...
Function Area ( R : real; var A : real );
...
Function Circumference ( R : real; var C : real );
...
Procedure PrintResults ( R, A, C : real );
...

begin
   GetRadius ( Radius );
   PrintResults ( Radius, Area (  Radius ), Circumference ( Radius ) )
end.
```

Functions: d = rt, c = 2pi*r, a = pi*r*r

When to use abstraction

When to use Procedures

When to use Functions

Design

Top-down design, stepwise refinement

Correctness

Pre- and Post-conditions

Implementation

stubs, animation, dummy values

Programs with Iteration I

PR: intro prog lang

Loops and recursion

```
For Index := lower bound to {upper bound} Do
   something

For Index := upper bound downto {lower bound Do
   something
```

where bounds and index are expressions and variable of an enumerated type

SE: software engineering

Definite Iteration

\bex Circle program \eex \bex Average \eex \bex Fibonacci numbers \eex \bex Multiplication table \eex

When to use the definite *Iteration* control structure

- Generate a sequence of values --- the alphabet, series of fractions, counting numbers etc.
- Obtaing a sequence of input values from the program user --- numbers to use in calculations etc.
- Modeling dynamical, `feedback' systems by returning to particular data items again and again.
- Generating patterns based on counts.
- Creating tables of values.

Template---Input One, Output One

```
{ one at-a-time processing }
repeat the following steps:
   read a value;
   process it
```

Template---Do Something A Specified Number Of Times

```
do the following a given number of times:
   perform some action
```

Programs with Selection

HU: human computer communication

Menu driven interfaces

PR: intro prog lang

Conditional statements

Language:
relational and boolean expressions
Data:
Boolean

```
if boolean expression then
   {do something

if boolean expression then
   {do something
else
   {do something else

case  expression of some enumerated type  of
   sequence of expressions : do something
   range expression : do something
end
```

SE: software engineering

When to use the *Select* control structures

- *Selecting* a particular value or kind of value from a flow of data --- a dollar sign, a number etc.
- *Selectively updating* counters or other variables --- counting the number of items in each category.
- *Routing* data to the right part of a program --- menu driven user interface.
- *Error-checking* data --- making sure that if fits within certain boundaries, and fixing or rejecting values that don't.

Programs with Iteration II

PR: intro prog lang

Loops and recursion

```
While some condition is true Do
   something

Repeat
   something
Until some contition is true
```

SE: software engineering

SE1: Fundamental Problem Solving Concepts

Control structures: selection, *iteration*, recursion

When to use the indefinite *Iteration* control structure

Correctness I

- loop variant
- loop invariant

Correctness II

Goal
- overall purpose

Bound
- reason loop will end

Plan
- action loop is going to take

Kinds of bounds

Sentinel bounds
- scanning etc Example: count \# char in a sentence; ends in a ., ?, or !

Count bounds
- for loop or counting loop Example: arith drill; exit when three errors have occured

Limit bounds
- exact, boundary, relative Example: Newton's method for computing square roots x := (n/x + x)/2; numerical integeration.

Data bounds
- data used up Example: binary search

Programs with Procedures and Functions

\part{Data}

Text Files

Computing with characters as opposed to numbers is called *text processing*.

PR: intro prog lang

eoln, eof

SE: software engineering

Examples:

- count number of chars, words and lines in a file.
- number of characters in a sentence
- pretty printing of output (number of lines per page)
- reduce to lower case
- copy files
- translation
- data validation

Streams and Filters

```
01234546789\kill
while not eof do begin
   read(Ch);
   write(Ch);
end
```

Enumerations and other types

PR: intro prog lang

Basic type declarations

Turbo Pascal's built in types

SE: software engineering

SE1: Fundamental Problem-Solving

\subsubsection{Data Types} Enumerations, subranges, etc

Arrays

PR: intro prog lang

Arrays and records

SE: software engineering

Count bounds
- array *traversal*
  - initialize
  - print, retrieve
  - find largest, smallest etc
  - shift items
  - vector and matrix arithmetic

Sentinel bounds
- array *searching*
  - finding the location of a particular value
  - text processing (stored in an array)
  - locating a subsequence

Data bounds
- *divide and conquer* the array
  - describe the array in terms of relative borders
  - divide a large array into smaller pieces to be processed separately
  - place barriers between portions of an array
  - gradually shrink the unexplored portion of an array.

Limit bounds
- simulations
  - Modeling diffusion or flow
  - Modeling population growth
  - Modeling cycles

Data:

Arrays -- 1D, 2D

Algorithms and Techniques:

~
- Validation Loops
- Auxiliary variables (boolean flags)
- Searching and Sorting
  - Searching -- iterative, recursive, binary
  - Sorting -- bubble, insertion, selection
- Sequential Files
- Text
- Binary
  - Searching -- iterative, recursive, binary
  - Sorting -- bubble, insertion, selection

Records

PR: intro prog lang

Arrays and records

Binary Files

PR: intro prog lang

```
{Binary Files in Turbo Pascal}
program FileDemo ( input, output );

type
   ItemType = item type definition
   FileType = file of ItemType

var
   Internal : FileType;
   External : string;
...
   { To read from a file-- }
   readln( External );
   SYSTEM.assert( Internal, External );
   reset( Internal );
   ...
   read( Internal, variable list );
   ...
   SYSTEM.close( Internal );

   { To write to a file-- }
   readln( External );
   SYSTEM.assert( Internal, External );
   rewrite( Internal );
   ...
   write( Internal, variable list );
   ...
   SYSTEM.close( Internal );
```

SE: software engineering

Characteristics of files

- Files are controlled by the computer's operating system and they exist independently of any individual program.
- Files are stored sequentially -- they have to be read, component by component, from beginning to end, items may be added only at the end.
- File length is not limited.
- Files have a *state* that depends on whether they are being inspected (read) or generate (written).

Last update:

Send comments to: webmaster@cs.wwc.edu

State

To the beginner the programming process can be missleadingly simple. This can be attributed, in part, to two factors. First, the beginning programmer's earliest programming problems can usually be solved by writing an intuitively obvious sequence of program statements. Unfortunately, this intuitive approach to programming does not hold up to more complex problems. In fact the complexity of programs increases quickly. Second, it is common among both beginning and experienced programmers to use the ``successive approximation'' technique of programming. You write a program to solve a specified problem. If the program does what it is supposed to do then you are done. Otherwise, you make a change in the program which will cause the new version to more closely approximate what the program is supposed to do. Especially for beginning students, the modifications made at each step are determined more by (misguided) intuition than by logic. Using this method students can often produce a program which satisfies its specification, but the student cannont explain how the program works. These two factors can be summarized in

---

The First Principle of Programming

It is easy to write the statements of a correct program; what is difficult is getting these statements in the correct order.

The goal of a programming course is to help students evolve their own programming style and methodology which is guided by --expandlater. A sound programming methodology should be based on, among other things, an understanding of the basic characteristics of programs. Every program can be seen as a static or a dynamic object. The static object is actually a description, in some high level language, which will eventually be translated to an equivalent machine language form. Static characteristics include syntax, algorithm description and data storage description. The dynamic characteristics of a program can be seen when the program executes, and include correctness (does the program execute without error) and validity (does the program satisfy its specification). The static characteristics

```
statically
        syntax
        algorithm
        data
dynamically
        validation (does it do the specified things)
        verification (does it run to completion)
```

To gain an understanding of how to properly order program statements requires a more scientific approach to programming, an approach which is based on a thorough understanding of the basic charactersitics of a program.

Send comments to: webmaster@cs.wwc.edu

Assertions and Semantics

Send comments to: webmaster@cs.wwc.edu

Abstraction

Assertions for procedures and functions: pre- and post-conditions

In this chapter, we focus on using procedures and functions to make programs easier to read and understand.

It is a law of the mind that we can retain only five to nine things at a time in short term memory. The implication for programmers is that a well written program should resemble and read like a good outline. Just as the levels of an outline are levels of abstraction, a program should be structured into levels. The main level of a program should read like the outermost levels of an outline; a very general list of steps to be taken. Additional detail is provided at successively lower levels until the level of simple statements is reached.

For example, consider an interactive program which prompts a user for the radius of a circle and then prints its circumference and area.

```
Pi = 3.1415;
write ( 'Enter the radius $>>$ ' );
read ( Radius );
Circumference, Area := 2*Pi*Radius, Pi*sqr( Radius );
write ( 'For a circle of radius ',  Radius );
write ( 'The circumference is: ',Circumference );
write ( 'The area is: ', Area );
```

Even though this program is small enough to be completely understood as it stands, it is complex enough to serve as an illustration of how levels of abstraction may be used in programming. The program consists of three sections, an input section, a processing section and an output section. Rewritten in the form of an outline and using comments to provide the levels of abstraction, the program becomes:

```
Pi = 3.1415;
-- Get Data
   -- Print Prompt
      write ( 'Enter the radius $>>$ ' );
   -- Read Radius
      read ( Radius );
-- Compute Results
   Circumference, Area := 2*Pi*Radius, Pi*sqr( Radius );
-- Print Results
   write ( 'For a circle of radius ',  Radius );
```

```
   write ( 'The circumference is: ', Circumference );
   write ( 'The area is: ', Area );
```

The comments are an explanation of what the program does. A better solution is to use procedures and functions where each comment is replaced with a procedure. The resulting program is:

```
Pi = 3.1415;
get_data ( Radius );
compute_results ( Radius, Circumference, Area );
print_results ( Radius, Circumference, Area );

where

procedure get_data ( R : number );
   print_prompt;
   read_radius ( R );
end.

procedure print_prompt;
   write ( 'Enter the radius $>>$ ' );
end.

procedure read_radius ( R : number );
   read_radius ( R );
end.


procedure compute_results ( R, C, A : number );

   C, A := circum ( R ), circle_area( R );

end.



function circle_area ( R : number ) : number;

   area := Pi*sqr( R );

end.



function circum ( R : number ) : number;

   circum := 2*Pi*R;

end.
```

```
procedure print_results ( R, C, A : number );

   write ( 'For a circle of radius ',  R );

   write ( 'The circumference is: ', C );

   write ( 'The area is: ', A );

end.
```

The number of procedures and functions used in this example are excessive but they serve to illustrate the first principle of abstraction.

The First Principle of Abstraction

A procedure or function is a named sequence of statements which is called by mentioning its name.

Or simply, procedures and functions are sequences of statements which are named and the name is used wherever the sequence of statements is needed.

For short programs (< 1000 lines), procedures and functions are the primary means by which abstraction is obtained.

While abstraction can be introduced into a program by grouping sequences of statements into a procedures or functions as in the previous example, it is more appropriate to design programs ``top-down''.

```
procedure compute_results ( R, C, A : number );
PRE: R $\ge$ 0
POST: C = 2*Pi*R, A = Pi*R*R
   C, A := circum ( R ), circle_area( R );
end.
```

An assertion placed prior to the body of an abstraction is called a {\em pre-condition} and an assertion placed after the body of an abstraction is called a {\em post-condition}. \expandlater

---

{The Second Principle of Abstraction

A procedure or function should be called only if its pre-condition is satisfied and its post-condition is required.

Post-condition

In this example, there is no pre-condition, the function {\sf get_positive} may be used any where it is required that the input be scanned and the first positive number returned.

```
function get_positive : number;
POST: a positive number is returned
...
```

Pre-condition

In these examples, the pre-condition requires that the parameter be non-negative, if the user passes a negative value, the result is unspecified. The function may abort, causing the user program to abort or it may return an arbitrary value misleading the user program.

```
function sqrt ( X : real ) : real;
PRE: X $\ge$ 0
POST: the square root of X ($\sqrt{\mbox{\sf X}}$) is returned
...
end.

function factorial ( N : number ) : number;
PRE: N $\ge$ 0
POST:factorial ( N! ) is returned
...
end.
```

\expandlater

Send comments to: webmaster@cs.wwc.edu

Selection

An example here is the square root function. The selection statement permits the distinction between an assertion and a comment since the selection statement may be used to verify the correctness of the input. Without the selection statement, the assertion is no more than a comment hoping that the input is correct

```
read( X );
Assert: X is a number (otherwise read statement would fail)
compute {\sf X} factorial
Y := X + 5;

read( X );
if X $\ge$ 0 then
   Assert: X $\ge 0$
   compute factorial
else Assert: X $< 0$
   print error message
```

The principle of selection

Regardless of the branch taken, the selection statement establishes the post-condition.

Send comments to: webmaster@cs.wwc.edu

Repetion

The first principle of repetition

Every loop has an invariant.

The second principle of repetition

Every loop has a variant.

---

while *condition* do *statement*

Three Questions

1. What task is to be performed in a single loop iteration? (the answer becomes the body of the loop)
2. Under what conditions should repetition continue? (the answer becomes the loop condition)
3. What is required for the loop condition to be tested and the loop body execute the first time? (the answer becomes the initilization code for the loop)

```
initialization code
   { assert: Loop Invariant; Loop Variant $\ge$ 0 }
while loop condition do begin
      { assert: Loop Invariant; Loop Condition; Loop Variant $>$ 0}
   loop body
      { assert: Loop Invariant; Loop Variant $\ge$ 0}
end
   { assert: Loop Invariant; Not Loop Condition}
```

Counter Loops

```
Counter := 1;
   { assert: Loop Invariant; Limit-Counter $\ge$ 0 }
while Counter $<$ Limit do begin
      { assert: Loop Invariant; Limit-Counter $>$ 0 }
      { Perform some task }
```

```
      Counter := Counter+1
      { assert: Loop Invariant; Limit-Counter $\ge$ 0 }
end
   { assert: Loop Invariant; Limit-Counter $=$ 0 }

   {Pre: Counter $\ge$ 0}
procedure Loop( Counter : integer );
begin
      if Counter = 0 then
           finished
      else begin
           Perform the task
           Loop( Counter - 1 )
      end
end
...
Loop( Limit )
```

Example: Summing n-items (sum=$\sum_{i=1}^{Count}x_i$); Factorial (ans=$n!/(i!)$) Consider n! = if n = 0 then 1 else n*(n-1)*...*1 and n! = if n = 0 then 1 else n*(n-1)!

Trailer Loops

```
Get( Something );
   { assert: Loop invariant; \# items to get - \# items got $\ge$ 0 }
while not TrailerValue(Something) do begin
      { assert: Loop invariant; \# items to get - \# items got $>$ 0 }
      { Perform some task }
      Get( Something )
      { assert: Loop invariant; \# items to get - \# items got $\ge$ 0 }
end
   { assert: Loop invariant; Something $=$ TrailerValue }
```

Example: summing n?-items (sum=$\sum_{i=1}^{\mbox{\# items got}}x_i$); \# of Words in a sentence (\# words = \# blanks read)

Correctness

- Safety --- Nothing bad will happen (e.g. Invariant property of a loop)
- Liveness --- Something good will happen (e.g. Variant property of a loop providing progress toward termination)

The process of verifying the correctness of a loop can be reduced to the verification of four loop properties.

Initialization

The loop must be properly intitialized. Preservation
Each iteration must perform the desired task. Finialization
Upon loop exit the desired results are true. Termination

The loop will eventually terminate execution.

The Loop Invariant

- Begin with: *What has happened so far?*
- What is supposed to be true as a result of the completion of the loop?

The Loop Variant

- What is the termination condition?
- If the termination condition is not met the loop is entered. Will the loop body insure progress toward termination?

Format

```
{ Establish Invariant and Variant $\ge$ 0 }
while BooleanCondition do
   {Variant $>$ 0}
   Command
   {Maintain invariant}
end
{Invariant and Variant = 0 implies Post Condition}

repeat
   Command
until BooleanCondition
```

Examples

1. Skip blanks {Inv: Previous chars where blanks} {Var: Length of line to be processed}

   ```
   repeat
       Read ( SomeChar )
      {Inv: Previous chars were blank}
   until SomeChar $\ne$ Blank
   ```

2. Number of lines and average number of characters per line {Inv: l is \# lines seen; c is \# chars seen} {Var: length remaining}

   ```
   Suma := 0
   CountByTwo := 0
   while CountByTwo $<$ 20 do
      Suma := Suma + CountByTwo
      CountByTwo := CountByTwo + 2
   end

   Sumb := 0
   Counter := 3
   repeat
   ```

```
        Sumb := Suma + Counter
        Counter := Counter + 1
    until Counter $>$ 4
```

Send comments to: webmaster@cs.wwc.edu

Data Types

Program Construction using Assertions

Introduction

The software process consists of the activities and associated information that are required to develop a software system.

Specification-Based Models

In the specification-based model, the task of the programmer is:

> Given a specification, develop a program that satisfies the specification.

Specification based models are most applicable to large system-engineering projects where there is a clear goal and the system is developed in parallel by different individuals or teams.

The prototypical specification-based model is the *waterfall model:*

1. *Specification*
2. *Design and implementation*
3. *Integration and testing*
4. *Operation and maintenance*

The problem with this model is the lack of feedback from one stage to another. In practice, there is always some interaction between phases of the model.

Incremental models are a further development of the waterfall model. The system functionality is partitioned into a series of increments and theses are developed one by one.

Evolutionary Development Models

1. Formulate an outline of the system requirements.
2. Develop a system as rapidly as possible.
3. Evaluate this system with users and modify the system until the system functionality meets the users' needs.

Evolutionary development is the most appropriate approach for interactive systems with significant user interface component and for innovative systems.

Miscellaneous Material

Our goal is to develop programs methodically by using programming theory. It is possible to reason about programs in a mathematically precise manner. It is also possible to incrementally refine the *program specification* and *mechanically translate* the specification to an equivalent program.

Program composition:

> The method by which programs are put together to form larger programs.
>
> There are three basic ways to *compose* programs: in parallel, by choice or sequentially. Program components share information.

Types of programs

- Transformational programs
- Reactive programs
- Object-Oriented programs
- Realtime programs

GET MATERIAL FROM HOME

References

Sommerville, Ian (1996)
> Software Process Models, *ACM Computing Surveys*, Vol. 28, No. 1, March 1996.

Chandy & Taylor

Last update:

Send comments to: webmaster@cs.wwc.edu

Software Engineering: The Software Life Cycle

The goal of software engineering is to develop software that satisfies and possibly exceeds the customer's expectations, is developed in a timely and economical fashion, and is resilient to change and adaptation. Such software

- exhibits a strong architectural vision and
- is the result of a well-managed iterative and incremental development life cycle.

This document provides an overview of the software engineering process and although the process is described in terms of phases, the phases are usually, iterative, incremental and to some extent, concurrent. When the phases are sequential, the life-cycle is called the waterfall model, when the phases are sequential and iterated, the life-cycle is called the spiral model. The spiral model is superior to the waterfall model when resources are limited.

Conceptualization Phase

Conceptualization produces a statement of the *problem and the desired solution*. The output of the conceptualization phase is a *requirements document*.

For most programming exercises, there is no conceptualization phase, the requirements document is the problem assigned as a programming exercise.

Analysis Phase

Analysis starts with the requirements and produces a specification of *what the system does*. The output of the analysis phase is a *specification document*.

For most programming exercises, the analysis phase

- produces a description of the input and output,
- defines the relationship between the input and output,
- generates test cases to be used to demonstrate the correctness of the program, and
- defines input which will cause errors.

Design Phase

Design begins with the specification and produces a description of *how the system will be built* from implementation-oriented components. The output of the design phase is a *design document*.

For most programming exercises, the design phase

- produces a description of the data structures used to organize and store data,
- designs the algorithms to process the data,

- identifies and orders the tasks required to solve the problem and
- designs the user interface.

Implementation Phase

Implementation begins with the design and produces an encoding of the design in a programming language to produce a *working system*. The output of the implementation phase is *code*.

For most programming exercises, the implementation phase should be a straight forward translation of the design into code and the program should be thoroughly tested for compliance with the specifications.

Maintenance Phase

Maintenance begins when the system is put into service and is concerned with managing the evolution of the system in response to changing requirements.

For most programming exercises, there is no maintenance phase.

Example

Requirements
: A program to compute the circumference of a circle given its radius.

Specification
: Users will invoke the program via the command *circle* which then prompts the user for input. Upon receiving the input, the program displays the result and terminates. The data objects required by the program are the radius and the circumference which are related through the formula: Circumference = 2 Pi radius. They should be floating point numbers. The value of Pi will be a constant internal to the program. In addition, a prompt for user input and labeling of the program output is required.

In the following sample run of the program, the system prompt is > and user input is in italics.

```
> circle

Enter the radius: 34.56

The circumference is: 1953.33

>
```

Errors: Given a negative value for a radius the program will compute a negative circumference. Given non-numeric input, the program behavior is unpredictable.

Design

Data Structures
: The constant Pi of value 3.14, the variable radius of type floating point.

Algorithms
: Prompt and read the input, the formula C=2 Pi R, for computing the circumference, implemented as a function, and labeling of the output.

Program Structure

1. GetInput (radius)

    1. Display prompt
    2. Read radius
2. DisplayResult (Circumference(radius))

Code (C++)

```
/*****************************************************************
Description: A program to compute the circumference of a circle
Input: The radius of the circle
Output: Prompt for input, labeled circumference of the circle
Programmer: A. Aaby
Date: January 13, 1993
Revision History:
*****************************************************************/

#include <iostream.h>

/*****************************************************************
The Get Input Function

Description: Prompts for input, input must be a real number
or an integer.  Other input may cause the program to abort.
Precondition: None
Postcondition: The parameter, r, is a number
*****************************************************************/

void GetInput(float &r)
{
    cout << "Enter the radius: ";
    cin >> r;
}

/*****************************************************************
Function implementing the formula C = 2*Pi*R

Precondition: r is a real number
Postcondition: Circumference = 2*Pi*r
*****************************************************************/

void Circumference(float &r, float &C)

{
    float Pi = 3.14;
    C = 2*Pi*r;

}

/*****************************************************************
The Display Result Function
```

```
Precondition: parameter C must be a number
Postcondition: C is printed, labeled as the circumference  of a circle
*****************************************************************/

void DisplayResult(float &C)

{
    cout << endl << "The circumference is: " << C;
}

/*****************************************************************
The Body of the program
*****************************************************************/

void main()

{
    float C = 0,r = 0;
    GetInput(r);
    Circumference(r,C);
    DisplayResult(C);
}
```

Code (Pascal)

```
PROGRAM Circle (Input, Output);

{*****************************************************************
Description: A program to compute the circumference of a circle
Input: The radius of the circle
Output: Prompt for input, labeled circumference of the circle
Programmer: A. Aaby
Date: January 13, 1993
Revision History:
*****************************************************************}

CONST Pi = 3.14; { an approximation to pi }

VAR radius : REAL; { the radius of the circle}

{*****************************************************************
The Get Input Procedure

Description: Prompts for input, input must be a real number
or an integer.  Other input may cause the program to abort.
Precondition: None
Postcondition: The parameter, R, is a number
*****************************************************************}
```

```
PROCEDURE GetInput( VAR r : REAL );

   BEGIN
      Write( 'Enter the radius: ');
      Readln( r )
   END;

{*****************************************************************
Function implementing the formula C = 2*Pi*R

Precondition: R is a real number
Postcondition: Circumference = 2*Pi*R
*****************************************************************}

FUNCTION Circumference( r : REAL ) : REAL;

   BEGIN
      Circumference := 2*Pi*r
   END;

{*****************************************************************
The Display Result Procedure

Precondition: parameter C must be a number
Postcondition: C is printed, labeled as the circumference  of a circle
*****************************************************************}

PROCEDURE DisplayResult( c : REAL );

   BEGIN
      Writeln( 'The circumference is: ', c:5:2 )
   END;

{*****************************************************************
The Body of the program
*****************************************************************}

BEGIN

   GetInput(radius);
   DisplayResult(Circumference(radius))

END.
```

Validation

The program is validated by providing sample runs of the program using positive and negative integers and real numbers which demonstrate the range and precision of the program.

This document borrows from: **Object-oriented analysis and design: with applications** by Grady Booch

---

Maintained by WWC CS Department

Last Modified:

Send comments to: webmaster@cs.wwc.edu

Software Engineering: The Analysis Phase

Analysis starts with the requirements and produces a specification of *what the system does*. The output of the analysis phase is a *specification document*.

The goal is to understand the problem domain and produce a model of the system which describes the classes of objects that exist in the system, the relationships between those classes, the operations that can be performed on the system, and the allowable sequences of those operations.

The resulting specification must be complete, consistent, readable and testable against reality and its purpose is to provide a model of the desired behavior of the system that is to be built.

Methods

There are three categories of methods

- Algorithmic Decomposition
- Data-driven
- Object-oriented

Algorithmic Decomposition

Algorithmic decomposition is often called *Top-down structured design*. In this approach, the problem is decomposed into subproblems, each of which are decomposed into subsubproblems until the level of a trivial problem is reached.

Related to algorithmic decomposition are outlines for talks and papers, assembly instruction, and the structure of a book.

This method is supported by subprogram constructs in traditional programming languages. But, it does not address the issues of data abstraction, information hiding, or concurrency and it does not scale up well for handling complex systems.

Data-Driven Analysis

In data-driven analysis, the structure of the software system is derived by mapping system inputs to outputs. A model of the problem domain is constructed by:

1. Drawing the data flow diagram. (Depict what happens rather than how it happens)
2. Deciding what sections to computerize.
3. Specifying the details of the data flow.

4. Defining the logic of the processes. (Use pseudocode to define each process)
5. Defining the data stores. (Define the exact contents of each data store and its format)
6. Defining the physical resources.
7. Determining the input/output specifications. (Define the user interface)

This method is supported by subprogram constructs and user defined data types in traditional programming languages. But, it does not address the issues of data abstraction, information hiding, or concurrency and it does not scale up well for handling complex systems.

Object-Oriented Analysis

Object-oriented analysis examines the requirements from the perspective of the classes and objects found in the vocabulary of the problem domain. Thus, it is the process of identifying and modeling the essential object classes and the logical relationships and interactions among them.

Object-oriented analysis should discover the

- classes of objects that exist in the system and the relationships between those classes and
- operations that can be performed on the system and the allowable sequences of those operations.

This method is supported by classes, objects, inheritance and polymorphism in object-oriented programming languages such as C++, Eiffel, and Small-Talk since they provide for data abstraction and information hiding.

Specification Document

For most programming exercises the specification document may be developed with the aid of the following outline:

Title:
Description:
Include formulas and relationships required to solve the problem
Input:
Include type and format of input data.
Output:
Include user prompts and other program output.
Errors
What to expect if the input does not conform to the specifications.
Example:
What a sample interaction with the program should look like.
Test Data:
Appropriate test data for customer acceptance and program validation

At this point, a user's manual can be written. It is a guide to using the program. It includes the purpose

of the program and sample runs to show users how to use the program.

This document borrows from: **Object-oriented analysis and design: with applications** by Grady Booch and **Object-Oriented Development** *the Fusion Method* by Coleman et. al.

Maintained by WWC CS Department
Last Modified:
Send comments to: webmaster@cs.wwc.edu

Software Engineering: The Design Phase

Design begins with the specification and produces a description of *how the system will be built* from implementation-oriented components. The output of the design phase is a *design document*.

Program design is an orderly methodical process often involving creative insight. The design process can begin as soon as there is some formal or informal model of the problem to be solved. It progresses through repeated partitioning of the model of the problem to be solved. It should stop when key abstractions are simple enough to require no further decomposition and can be composed from existing reusable software components. Thus, the design document is a `blueprint' for the implementation of the system. A good design adheres to the following principles:

- Object interaction:
  - Minimize object interactions
  - Cleanly separate functionality: each module should perform one action or achieve a single goal.
  - Develop modular systems
- Visibility: Minimize data and functional dependencies

Definitions

Divide and Conquer
: Partition the problem into simpler subproblems, each of which can be considered independently.

Stepwise refinement
: Postpone decisions as to details as late as possible in order to be able to concentrate on the important issues.

Top-down
: Begin with those aspects of the problem that are the most general and which use other program components.

For example, begin with the user interface, then functions which would be invoked by the interface then those called by those functions, etc.

Bottom-up
: Begin with those aspects of the problem that are most basic and used by other program components.

Methods

Process-Oriented Design

Process-Oriented Design is suitable for programs which are defined in terms of their input and output (a program *transforms* its input into its output).

Data Flow Analysis

The basic approach is to identify the tasks to be performed on the data and the order in which they are to be carried out. Programs that are developed are refinements of the sequence: **Input -- Process -- Output** or

```
Repeat
   Input an item and process it
```

Transaction Analysis

The basic approach is to identify the transactions that the system is to perform. Programs that are developed are refinements of

```
Repeat
   Get Transaction
   Choose action from alternatives
```

Examples include: menu driven programs and programs for automated teller machines.

Data-Oriented Design

The basic approach is to design the program according to the structure of the data on which it is to operate. The program is a *model* of the real world that is relevant to the problem. The model is developed in terms of *entities* and *actions* that can be performed on them. Examples include: game playing programs.

Object-Oriented Design

The initial steps are similar to that of data-oriented design but the model is developed in terms of *objects* and *actions* that can be performed on them and objects are viewed as instances of a *class*. From **Object-Oriented Development *the Fusion Method*** by Coleman et. al. Prentice Hall. The design phase delivers models that show the following:

- How system operations are implemented by interacting objects
- How classes refer one to another and how they are related by inheritance
- Attributes of, and operations on, classes

Examples include: windowing environments where a window may be a customization of a more general window design and financial programs where a transaction may be a customization of general transaction operation.

Rule-Based Design

The basic approach is to design the program according to a set of rules that describe the problem domain and use an inference engine to apply the rules to the given input to determine the output. Examples include: database programs, expert systems and compilers.

Documentation

For programming exercises the design document should include

- Definitions of key data structures.
- Interface (parameters) and pseudocode for each function or procedure.
- A hierarchical structure chart indicating dependencies between modules.

A **programming manual**
: is a guide to installation and modification of the program. It includes information concerning the design of the program including data structures and algorithms.

Maintained by WWC CS Department
Last Modified:
Send comments to: webmaster@cs.wwc.edu

Software Engineering: The Implementation Phase

Implementation uses the design document to produce code.

The code is validated by demonstrating that the program satisfies its specifications. Typically, sample runs of the program demonstrating the behavior for expected data values and boundary values are required.

Methods

Small programs are written using the model:

```
write / compile / test
```

It may take several iterations of the model to produce a working program. As programs get more complicated, testing and debugging alone may not be enough to produce reliable code. Instead, we have to write programs in a manner that will help insure that errors are caught or avoided.

Top-Down Implementation

Implementation begins with the user invoked module and works toward the modules that do not call any other modules. The implementation may proceed depth-first or breadth-first.

Bottom-Up Implementation

Implementation begins with modules that do not call any other modules and works toward the main program. Test harnesses (see below) are used to test individual modules. The main module constitutes the final test harness.

Stubs

Stub programming is the implementation analogue of top-down design and stepwise refinement. It supports incremental program development by allowing for error and improvement.

A **stub** program
> is a stripped-down, skeleton version of a final program. It doesn't implement details of the algorithm or fulfill all the job requirements. However, it does contain rough versions of all subprograms and their parameter lists. Furthermore, it can be compiled and run.

Extensive use of procedures and parameters are the difference between stub programs and prototypes. Quick and dirty prototypes should not be improved--they should be rewritten. A stub program helps demonstrate that a program's structure is plausible. Its procedures and functions are unsophisticated versions of their final forms, but they allow limited use of the entire program. In particular, it may work for a limited data set. Often the high-level procedures are ready to call lower-level code, even if the more detailed subprograms haven't even been written. Such sections of code are commented out. The comment brackets can be moved, call by call, as the underlying procedures are actually written.

Incremental Program Development

As programs become more complex, changes have a tendency to introduce unexpected effects. Incremental programming tries to isolate the effect of changes. We add new features in preference to adding new functions, and add new functions rather than writing new programs. The program implementation model becomes:

```
define types / compile / fix;
add load and dump functions/ compile / test;
add first processing function / compile / test / fix;
add features / compile / test / fix;
add second processing function / compile / test / fix;
keep adding features / and compiling / and testing / and fixing.
```

Object-Oriented Programming

Given an object-oriented design,

- inheritance, reference, and class attributes are implemented in programming language classes,
- object interactions are encoded as methods belonging to a selected class, and
- the permitted sequences of operations are recognized by state machines.

From **Object-Oriented Development *the Fusion Method*** by Coleman et. al. Prentice Hall.

Testing

A **trace**
of a program consists of a listing of the values of each variable at each point in the execution of the program.

It is often difficult to determine what causes a program to fail. While program tracing is useful, it is difficult to perform on any but the smallest programs. Stub programs let larger systems be debugged and tested as they are being built, a small portion at a time. Major program connections are tested first, which means that major bugs and shortcomings are detected early in the game. Furthermore, testing and debugging are distributed throughout the entire implementation. Even if a program isn't completely finished by the due date it's a preliminary working version--and not just a useless mess of code. Often procedures are built into programs to assist in testing a program and left behind in case they're needed again. It is also common practice to build certain testing tools that are thrown away.

Walkthrough

Working on a program tends to create a mind set in the programmer that renders obvious mistakes invisible. Merely explaining a program aloud can give a totally new view of it.

A **walkthrough**
is an explanation and defense of the program's algorithm and implementation to an audience.

Program Animation/Instrumentation

Program animation/instrumentation is a way to inspect a program while it is running. It differs from tracing in that the values of selected variables are printed at specific points in the program.

A **program probe**
is an output statement added to a program to print the value of a variable or to indicate the progress of execution.

When the program is executed, the output statements are probes into the program which serve to animate the program. The program can then be compared to the output to isolate the program errors. A program probe often takes the following form:

```
Debug = true;
...
if Debug
{
    label and print the value of the variable;
}
...
```

Often the particular value of a variable is not as important as whether or not it meets some particular constraint (e.g. positive). Statements which express such constraints are called **assertions**. A probe which prints an error message only when variable fails to meet the constraint can be written as follows:

```
void assert(BooleanExpression)
{
    if not BooleanExpression print error message ;
}
```

Then a statement such as: assert(X>=0,"X is negative"), can be inserted into the appropriate point in the code.

An **assertion**
is a boolean expression which is to be true at a particular point in a program.

Test Harnesses

A **test harness**
is a program shell that is used to test procedures in isolation, before they are integrated into a more complex final program.

A program is a delivery system for procedure calls. A test harness is precisely that--a delivery system for procedure tests that contains:

- the type definitions;
- procedures that initialize and/or inspect data structures; and
- the new procedure that is undergoing testing or modification.

The new procedure can be tested without having to deal with a main program that is more complex and finicky than the harness is. Once it works, the new procedure can be transferred to the main program.

Integration Testing

Testing to check that modules combine together correctly. In addition, there should be a final product test and acceptance testing by the client.

Regression Testing

Testing that is performed to insure that modifications to a program have not modified previously correct behavior.

This requires a collection of test data be maintained for the purpose of regression testing.

Documentation

Programs
: must have a header identifying the program, programmer, revision history and other pertinent information.

```
/*
Program file name:
Language:
Operating System:
Programmer:
Date:
Revision History:
Title: program title or name
Purpose: short description or purpose of the program
Input: what the user must supply
Output: what the program prints/produces
Special requirements:
*/

#include <iostream.h>

    void main()
    {
        the main program
    }
```

The following must be included when the code is submitted as part of a class assignment.

```
    Class:
    Section:
    Assignment:
```

Procedures and functions
: should be commented to identify their pre- and post-conditions.

```
    /*
    Title: function title or name
    Purpose: short description or purpose of the program
    Input/Precondition: what the user must supply
    Output/Postcondition: what the function computes
    Data Structures:
    Algorithms:
    */

    type function identifier (parameters)
    {
        body of the function
        return expression
    }
```

Repetitive structures
: should be commented to identify their purpose, termination conditions and progress functions.

```
/*
Goal/Invariant: overall purpose of the loop
Bound/Termination condition: reason loop will end
Plan/Metric: the action the loop will take including approaching the bound
*/
```

Complex algorithms
: should be commented to assist the reader in understanding the code and where appropriate a citation of the source of the algorithm.

Variables and data structures
: should be commented to identify their purpose and organization and access functions where appropriate.

95.6.9 a.aaby

Some portions are adapted from *Oh! Pascal* third edition

Maintained by WWC CS Department
Last Modified:
Send comments to: webmaster@cs.wwc.edu

Programs with Style

The visual appearance of your programs is as important as the solution they represent. Use plenty of white space (blank lines, indentation) to draw attention to the structure of your solution and to improve readability. In addition follow a consistent style in program layout. The suggestions that follow are just that, suggestions. You will develop your own style and employers usually set their own standards.

Reserved Words and Identifiers

Reserved words should be distinguishable from user defined identifers either by writing them in all caps or in all lower case or by judicious use of indentation and white space.

User defined identifiers should be chosen to aid the reader in understanding the program. Long names should be avoided as they tend to reduce the readability of a program. Single letter identifiers should be avoided except for loop indices in which case *i, j,* and *k* are typical indices.

1. Identifiers that name modules, procedures, exceptions, types and constants should begin with an uppercase letter. (e.g. *GetData, Factorial(n)*)
2. Identifiers that name variables, parameters, record and object components should begin with a lowercase letter. (e.g. *radius, score[i], student.name*)
3. The initial letter of each embedded word except the first should be capitalized or separated from the previous word by an underscore. Acronyms should be in a single size. (e.g. *thisLongName, this_long_name, CPUType, cpuType*)

Constants

Define and use symbolic names for constants. Only rarely should numeric literals other than 0 and 1 appear in the program text (e.g. *Pi = 3.14159*).

Types

User defined types should end in a capital *T* to permit the use of the name as a variable (e. g. *type_identifierT = type_definition, identifier = identifierT;*).

Punctuation

A single space usually appears before and after all binary operators including *:=* and = used in assignments and declarations (e.g. *x + y, x*y, first = last + 1, assignable* := *expression*).

A single space or a newline appears after colon, comma, and semicolon, but none before. Unless

required by adjacent tokens, no spaces appear before or after left and right parentheses, square brackets, or curly braces, or the up-arrow (^), period (.), or double period (..). (e.g. *Factorial(n), score[i], {a comment}, list^.tail, name.last, array[0..Max]*)

Statements in sequence are separated by semicolons. For consistency, also place a semicolon after the last statement in a sequence. (e.g. *First; Second; ... Last;*)

Indentation

Indentation should be used to show the structure of the program, declarations and statements. The amount of indentation should be the same -- two to four spaces seem to be good values. Statements or declarations needing more than one line should have subsequent lines indented more than one level.

C++

```
#include <iostream.h>

int variable;

void function()
{
    body
}

void main()
{
    if true
        initialize
    else
        process
}
```

When a declaration or statement can be placed on a single line without appearing to be cramped, consider doing so;

```
if (x < Limit) y = K*p;
```

is better than

```
if (x < Limit)
    y = K*p;
```

Blank Lines

Procedure declarations should be separated by at least one blank line. Long lists of procedure parameter declarations should be written putting each parameter on its own line.

Comments

A space follows the `{' beginning a comment and before the `}' ending a comment. If a comment extends over more than one line, subsequent lines should be indented the same level as the `{'.

A comment that applies to a group of declarations or statements should appear before the group and be preceded by a blank line. Major sections of code should be introduced by comments in boxes.

```
/************************************
  Section name and other infomation
************************************/
```

Adapted from: **Modula-3** by Samual P. Harbison

Maintained by WWC CS Department
Last Modified:
Send comments to: webmaster@cs.wwc.edu

Testing

Testing is the process of determining whether a task has been correctly carried out. Testing can show the presence of faults but cannot verify the correctness of the code. Testing should be performed throughout the software life cycle.

- *Nonexecution-Based Testing*
  - walk-throughs - code reading and inspections with a team
  - Cleanroom - incremental software life-cycle, formal techniques for specification and design, code reading and inspections.
  - correctness proving
- *Execution-Based Testing*
  - testing to specifications (black box testing) - test cases based on specifications
  - testing to code (glass box testing) - test cases based on the code
  - methodology

References

Schach, Stephen R. (1996)
Testing: Principles and Practices, *ACM Computing Surveys*, 28, 1, (March 1996), 277-279.

Parallel Program Design: A Formal Methodolgy

Processes

- Static creation
- Dynamic creation

Static creation

{|| producer(...), consumer(...) }

Dynamic creation

processes(N) {? N > 0 --> {|| process(N,...), processes(N-1,...) }}

Partitioning

Program Scalability: The measure of increased program performance for an increased number of computers. **Hiding Latency:** Use multiprocessing within each computer to keep computers busy during communication.

Domain Decomposition

Domain Decomposition: Divide up the *data* of the problem and operate on the parts concurrently Examples: Matrix operations

Problem Scaling: It is often necessary to scale the problem size with the number of computers to maintain performance improvements.

Functional Decomposition

Functional Decomposition: Divide up the *function* of the problem and operate on the parts concurrently Examples: Numeric integration, compiler

{|| integrate(a,x1...), integrate(x1,x2...), ... integrate(xN1,b...) } {|| scanner(...), parser(...), codegen(...) }

Granularity

Granularity: Group partitions to exploit locality thereby increasing the ratio of computation to communication.

Mapping and Load Balancing

{|| p403">(...)@node423"> ... }

Communication Protocols

Streams

[x425">, x427">, ..., x349">]

Stream Communication

{|| producer(...,S,...), consumer(...,S,...) } {|| scanner(Source,Tokens,...), parser(Tokens,ParseTree,...), codegen(ParseTree,Code,...) }

Bounded Buffer

{|| Buffer = [S1, S2, S3 439"> End], producer(Buffer), consumer(Buffer, End) }

Two-Way

```
Prompter( S )
{||           Prompt = ...,
   S = [{Prompt, Response}439">Ss],
   { ? Response == R1 --> {|| ..., Ss = ..., prompter(Ss) },
   ...
      Response == RM --> {|| ..., Ss = ..., prompter(Ss) }
   }
}

Responder( S )
{ ? S ?= [{Prompt, Response}439">Ss] ->
   { ? Prompt == P1 --> {|| ..., Response = ..., prompter(Ss) },
   ...
```

```
        Prompt == PN --> {|| ..., Response = ..., prompter(Ss) }
    }
}
```

Broadcast (One-to-many

```
{|| producer(S), consumer1(S), ... , consumerN(S) }
```

Many-to-One

```
{||     producer1(S1), producer2(S2),
   S1S2 = [{"merge", S1}, {"merge'',S2}],
   sys:merger(S1S2, S),
   consumer(S)
}
```

Distributors (One-to-Many)

Termination Detection

Detecting Termination: Chain programs together and place a constant on the left of the chain. When a program detects termination it closes a section of the chain. Eventually, the constant appears at the right end of the chain signifying termination.

Transformational Systems

Design Methods: The design method is based on the predicate calculus to permit working with relations between inputs and outputs rather than functions from input to outputs. Programs are developed as follows:

1. Given a sufficiently simple specification, derive a program from it in a simple mechanical fashion.
2. Given a complex specification, show that it is composed of simpler specifications. The composition operators are logical **and**, **or** and **implies**. Derive a program that is the composition of programs satisfying the simpler specifications.

Program design process:

- Given a specification, construct a specification in the predicate calculus.
- Given a specification in the predicate calculus, transform it into a canonical form using the following:
    - Use only logical and, or, and implies.

    - Use both and ``,'' for logical and.
    - Use to denote
- Design a program such that. For a PCN program:
    - Composition of specifications using conjunction corresponds to composition of programs using parallel composition.
    - Composition of specifications using implications corresponds to composition of programs using choice composition.

Programs are derived from specifications written in the predicate calculus

Proof Rules

Reactive Systems

An **invariant** of a program is a predicate that holds in all states of all computations of a program. Example: Bank account - amount = deposits - withdrawals

Predicates about states or state transitions that hold ``eventually'' are called *progress properties* (liveness properties). Example: If a part fails, a warning light will come on.

Predicates that hold for all states or for bounded-length sequences of transitions are called *safety properties*. Example:

Send comments to: webmaster@cs.wwc.edu

Systems Development

A summary of Vessy and Glass

Definitions

Strong problem solving methods
: designed to fit and do an optimal job on one kind of problem

Weak problem solving methods
: designed to adjust to a multiplicity of problems, but solve none of them optimally

Nature of Systems Development

Requires expertise in two disciplinary areas

- the area of the problem being solved (the application domain) and
- the area of constructing a software solution (the systems and software discipline).

Method-Based Approaches to Systems Development

- Unified methodology approach
  - Process oriented: structured analysis, design, and programming
    --used when the process is more stable than the data
  - Data oriented: entity relationship
    --used when data is more stable than process
  - Object-oriented: considers both data and process as a package; an object is a cohesive collection of data coupled with the processes acting on that data
    --used to model real-world objects and the ways they interact
- Technique approach -- a collection of techniques that have been known to work.

Combining Method and Application in Systems Development

- *Theory*. Cognitive fit is the notion that problem solving elements should support the strategies

(or processes) required to perform the task.

- *Matching methodology to application.*
  - Process oriented: scientific-engineering applications, payroll, inventory, accounts receivable, accounts payable
  - Data oriented: record keeping applications
  - Object oriented: applications where data and process are intimately related, real-time systems
- *Matching technique to task.* multiparadigm

References

Vessey, Iris & Blass, Robert
Strong vs. Weak: Approaches to Systems Development. *Commun. ACM 41, 4 (April 1998), 99-102*

Software Tools and Environments

Any system that assists the programmer with some aspect of programming can be considered a programming tool. Software tools include:

- *Program Editors*
- *Compilers*
- *Linkers and Loaders*
- *Preprocessors*
- *Cross Referencers*
- *Source-Level Debuggers*
- *Debugging Aids*

Additional tools that address software engineering issues include:

- *System Builders*
- *Version Managers*
- *Design Editors*
- *Code Generators*
- *Testing Aids*

References

Reiss, Steven P (1996)
Software Tools and Environments, *ACM Computing Surveys*, 28, 1, (March 1996), 281-284.

Send comments to: webmaster@cs.wwc.edu

Examples

Euclid's Algorithm for: x = y*q + r

Finds quotient and remainder by repeated subtraction Note that: dividend = divisor * quotient + remainder

```
q, r := 0, x
while r >= y do
   q, r := q+1, r-y
end
```

Proof of correctness:

```
{ 0 <= x, 0 < y } :
    q, r := 0, x;
{ 0 <= r, 0 < y, x = y*q + r } :
                              if r < y -> done
                              [] r >= y ->
{ 0 < y <= r, x = y*q + r }
{ 0 <= r - y, 0 < y, x = y*(q + 1) + (r - y) }
                                                  q, r := q+1, r-y;
      { 0 <= r, 0 < y, x = y*q + r }
                                             fi
{ 0 <= r < y, x = y*q + r }
```

Notes: Method: Delete a conjunct Variant: r r >= 0 implies x >= 0 r > r - y implies y > 0

Minimum

{Inv: x is smallest seen so far} {Var: length-i}

```
{Pre: SomeFile has just been opened, and contains at least one integer}
Read ( SomeFile, Min )
while not eof( Somefile ) do
   {Inv: Min is the mininum of the values read so far}
    Read ( SomeFile, SomeInt )
    if SomeInt < Min then
      Min := SomeInt
end
{Post: Min is the smallest value in the file SomeFile}
```

Algorithm for greatest divisor of N

{Inv: No value between i and N is a divisor} {Var: i}

```
{Pre: Number is an integer > 1}
Divisor := Number - 1
while Number mod Divisor $\ne$ 0 do
   {Inv: No value between Divisor and Number is a divisor}
   Divisor := Divisor - 1
end
{Post: Divisor is the greatest divisor of Number other than Number}
```

Algorithm for: gcd(a,b)

```
if a < 0 -> x := -a fi
if b < 0 -> y := -b fi
while x $\neq$ y do
   if x > y -> x := x-y
   [] x < y -> y := y-x
   fi
end
G := x
```

Properties of the GCD

```
gcd(x,y) = gcd(y,x)
gcd(x,y) = gcd(x,x+y) = gcd(x+y,y)
gcd(x,y) = gcd(x,y-x) = gcd(x-y,y)
gcd(x,y) = gcd(-x,y) = gcd(x,-y) = gcd(-x,-y)
gcd(x,x) = |x|
gcd(x,0) = gcd(0,x) = |x|
gcd(cx,cy) = c gcd(x,y)
gcd(x$^n$,y$^n$) = gcd(x,y)$^n$ for m >= 0
```

The rules:

```
gcd(x,x) = |x|
gcd(x,0) = |x|
```

are the only rules which give the gcd directly. Proof of correctness:

```
   x,y := a,b
{gcd(a,b) = gcd(x,y) } :
   if x = y -> done
   [] x > y -> {gcd(a,b) = gcd(x-y,y)}
         x := x-y;
      {gcd(a,b) = gcd(x,y)}
   [] x < y -> {gcd(a,b) = gcd(x,y-x)}
         y := y-x;
      {gcd(a,b) = gcd(x,y)}
   fi
{gcd(a,b) = x}
```

```
   G := x
{gcd(a,b) = G}
```

Method: replace a constant with a variable Variant is x + y $x+y > x-y+y \ge 0$ iff $y > 0$ $x+y > x+y-x \ge 0$ iff $x > 0$

Algorithm for: $\sum_{i=1}^n A_i$

```
S,I := 0,0
while I < n do
   S,I := S+A[I+1],I+1
end
```

Proof of correctness:

```
   { 0 = $\sum_{i=1}^0$A[i], 0 < n = |A| }
   S,I := 0,0
    { S = $\sum_{i=1}^I$A[i], I $\leq$ n }
   while I < n do
    { S = $\sum_{i=1}^I$A[i], I < n }
   { S+A[I+1] = $\sum_{i=1}^{I+1}$A[i], I+1 $\leq$ n }
      S,I := S+A[I+1],I+1
    { S = $\sum_{i=1}^{I}$A[i], I $\leq$ n }
   end
   { S = $\sum_{i=1}^I$A[i], I $\leq$ n, I $\geq$ n }
   { S = $\sum_{i=1}^n$A[i] }
```

Method: Replace a constant by a variable I = { S = $\sum_{i=1}^I$A[i], I $\leq$ n } V = n - I OR: {Inv: (sum=$\sum_{i=1}^{Count}x_i$)} {Var: N-i}

```
{Pre: 0 < N}
sum, i := 0, 0
{ Inv: sum=$\sum_{j=1}^ix_i$; Var: N-i >= 0}
while i < N do begin
   { Inv: sum+x$_{i}$=$\sum_{j=1}^{i}x_i$; Var: N-i > 0}
   { Inv: sum+x$_{i+1}$=$\sum_{j=1}^{i+1}x_i$; Var: N-(i+1) >= 0}
      sum, i := x$_{i+1}$, i+1
   { Inv: sum=$\sum_{j=1}^ix_i$; Var: N-i > 0}
end
{ Inv: sum=$\sum_{j=1}^Nx_i$; Var: N-i = 0}
```

First N Fibonacci numbers

{Inv: Fibs < NextFib have been printed} {Var: MaxFib - NextFib}

```
Read ( MaxFib )
{Pre: 0 < MaxFib}
OneFib, NextFib := 1, 1
Writeln ( OneFib )
```

```
{ Inv:  Fibs < NextFib have been printed; Var: MaxFib - NextFib >= 0}
while NextFib < MaxFib do begin
   { Inv:  Fibs < NextFib have been printed; Var: MaxFib - NextFib > 0}
   Writeln( NextFib )
   NextFib, OneFib := OneFib + NextFib, NextFib
   { Inv:  Fibs < NextFib have been printed; Var: MaxFib - NextFib >= 0}
end
{ Inv:  Fibs <= NextFib have been printed; Var: MaxFib - NextFib = 0 }
{ Post:  Fibs <= MaxFib have been printed }
```

The Even Divisors of N

{Inv: Even divisors > i have been printed} {Var: i}

```
{Pre: SomeInt >= 1}
Divisor := SomeInt
repeat
   if SomeInt mod Divisor = 0 then
      writeln ( SomeInt )
   SomeInt := SomeInt - 1
   {Inv: All divisors of SomeInt > Divisor have been output;
    Var: Divisor >= 0}
until Divisor = 0
```

The Factorial Function

{Inv: Factorial = i!} {Var: N-i}

```
Fac, Count := 1, 1
   {Inv: Fac = Count!, Count <= N}
while Count < N do
   {Inv: Fac = Count!; Var: N-Count > 0}
    Fac, Count := Count $*$ Fac, Count + 1
end
   {Inv: Fac = N!}
```

Merging

```
0
I, J, K := 0, 0, 0
Do I < NA and J < NB -->
   if A[I+1] <= B[J+1] --> C[K+1], I, K := A[I+1], I+1, K+1
   [] B[J+1] < A[I+1] --> C[K+1], J, K := B[J+1], J+1, K+1
   fi
[] I < NA and NB <= J -->
   C[K], I, K := A[I], I+1, K+1
[] J < NB and NA <= I -->
    C[K], J, K := B[J], J+1, K+1
od
```

```
I, J, K := 0, 0, 0
while I < NA and J < NB do
   if A[I+1] <= B[J+1] then
      C[K+1], I, K := A[I+1], I+1, K+1
   else C[K+1], J, K := B[J+1], J+1, K+1
end
while I < NA do
      C[K], I, K := A[I], I+1, K+1
end
while J < NB do
       C[K], J, K := B[J], J+1, K+1
end
```

Searching Algorithms

Linear Search

Find first occurrence of t in a

```
i := 0
while a[i] $\ne$ t do
   i := i+1
end
```

Proof of correctness:

```
    i := 0;
{  a[j] $\ne$ t for j=0..i-1 }:
                            if a[i] = t -> done
                            [] a[i] <> t ->} { a[j] $\ne$ t for j=0..i }
      i := i+1
      {  a[j] $\ne$ t for j=0..i-1 }
                                          fi}
{  a[i]=t, a[j] $\ne$ t for j=0..i-1 }
```

Variant: n-i n-i >= 0 implies n >= i n-i > n-(i+1) is true

Binary Search

```
0
LB, UB := 1, N;
Mid := (LB + UB) Div 2;
Do A[Mid] $\ne$ Target and LB < UB -->

   If A[Mid] < Target --> LB, Mid := Mid+1, (Mid+1+UB) Div 2
   [] Target < A[Mid] --> UB, Mid := Mid-1, (LB+Mid-1) Div 2
   fi
od;
```

```
If A[Mid] $\ne$ Target --> Mid := 0 fi
```

Sorting Algorithms

Bubble Sort

```
0
i := N;
Do i > 1 --> j := i;
   Do j > 1 --> If A[j-1] < A[j] --> A[j-1], A[j] := A[j], A[j-1];
      fi;
      j := j + 1
   od;
   i := i + 1
od
```

Insertion Sort

Selection Sort

Quick Sort

Send comments to: webmaster@cs.wwc.edu

Programming Patterns

A Patterns Catalog

A pattern involves a general description of a recurring solution to a recurring problem replete with various goals and constraints and explains why the solution is needed.

Hillside

Contents

References

- Brown et. al. *Anti Patterns: Refactoring Software, architectures, and Projects in Crisis*

- Cooper, James *Java design patterns: a tutorial* Addison-Wesley 2000
- Gamma et. al *Design patterns: elements of reusable object-oriented software* Addison-Wesley 1995
- Buschmann et. al *Pattern-Oriented Software Architecture: A System of Patterns* Wiley1996
- Lea, Doug *Christopher Alexander: An Introduction for Object-Oriented Designers* Software Engineering Notes Vol 19 No 1 Jan 1994.
- Martin Fowler *Refactoring: Improving the Design of Existing Code* Addison-Wesley 1999
- Thomas Kühne *A Functional Pattern System for Object-Oriented Design* Verlag Dr. Kovac 1999

Last Modified - . Comments and content invited aabyan@wwc.edu

---

An Imperative Language

Use unified grammar

Basic Type Declarations

character, number (integer, real)

Arithmetic Operators and Assignment

Arithmetic Operators

+,-,*,/

Assignment

The assignment command distinguishes the imperative programming paradigm from other paradigms. The syntax of the assignment command is:

$$X_0, X_1, \ldots, X_n := E_0, E_1, \ldots, E_n$$

where the X_i are variables and the E_i are expressions. The semantics of the assignment command are that the expressions E_i are evaluated simultaneously and the variables are assigned simultaneously to the corresponding values. Thus, this assignment command is frequently refered to as the `simultaneous' assignment command. For the purposes of this text, this command translates a sequence of assignments:

$$\begin{aligned} T_0 &:= E_0; \\ T_1 &:= E_1; \\ &\ldots \\ T_n &:= E_n; \\ X_0 &:= T_0; \\ X_1 &:= T_1; \\ &\ldots \\ X_n &:= T_n \end{aligned}$$

where the expressions E_i are evaluated in sequence and assigned to new variable names T_i then the variables X_i are assigned to the corresponding values. This is important to prevent the interference of one assignment with another. For example, the assignment

$$X, Y := Y, X$$

is in effect, a swap. But, if the X where first assigned to the value of Y and then the Y assigned to the value of {\em X}, both {\sf X} and {\sf Y} would end up with the same value. That is, suppose {\sf X} is assigned to a 3 and {\sf

Y} is assigned to a 5, then

```
T0 := Y
T1 := X
X := T0
Y := T1
```

results in X denoting the value 5 while Y denotes the value 3 vs.

```
X := Y
Y := X
```

which results in both X and Y denoting the value 5.

Sequential Composition

```
C0; C1; ...; Cn
```

Selection

```
IF [] B0 --> C0
   []+ B1 --> C1
   ...
   []+ Bn --> Cn
FI
```

translates to:

```
012\=345\=678\=90\kill
If B0 then C0
else if Bi then C1
...
else if Bn then Cn
end
```

Procedures, functions, and parameters

```
012\=345\=678\=90\kill
Procedure {\em identifier} ( {\em Formal Parameters} )
{\em declarations and code}
End

012\=345\=678\=90\kill
Function {\em identifier} ( {\em Formal Parameters} ) -> {\em Result Type}
{\em declarations and code}
End
```

Loops and Recursion

```
DO
   [] B0 --> C0
   []+ B1 --> C1
   ...
   []+ Bn --> Cn
OD
```

translates to:

```
012\=345\=678\=90\kill
While B0 or B1 or ... or Bn do
\>if B0 --> C0
\>\verb+[]+ B$_i --> C1
\>...
\>\verb+[]+ Bn --> Cn
\>fi
end
```

Arrays and Records

Arrays

An individual element of an n-dimensional array {\sf A} is referenced as follows: {\sf A$[i_0,...,i_n]$}.

Records

The field {\sf f} of a record {\sf R} is referenced as follows: {\sf R.f}.

Overall program structure

Assertions

```
Assert:  [; ]

Precondition:  [; ]

Postcondition:  [; ]

Invariant:  [; ]

Variant:  [; ]

 -->
               old
               nochange
```

the last two may only be used in postconditions; alternatives for `old' are: `original', `initial' and `previous'.

```
012\=345\=678\=90\kill
DO
Invariant: BE_{Inv}
Variant: BE$_{Var}$
\verb+[]+ B_0 --> C_0
\verb+[]+ B_1 --> C_1
...
\verb+[]+ B_n --> C_n
OD

012\=345\=678\=90\kill
Procedure {\em identifier} ( {\em Formal Parameters} )
Precondition: {\em boolean expression}
Postcondition: {\em boolean expression}
{\em declarations and code}
End
```

Send comments to: webmaster@cs.wwc.edu

Control Structures

The basic operations and control structures include:

- Assignment
- Choice/Selection
- Iteration (definite and indefinite)
- Input & output
- Linear sequence
- Parallel Composition

  $(P_0 \| ... \| P_n)$

  is used to compose actions which may occur in no particular order. It is not available in traditional programming languages.

To do: add axiomatic semantic descriptions.

Anthony A. Aaby. Last Modified - . Send comments to aabyan@wwc.edu

Data Structures

Data types, Structures and Abstract Data Types

- Array
- List
- Stack
- Queue
- Tree
- Graph
- Table, priority queue
- Searching & Sorting

eof(*file variable*), eoln(*file variable*), eof vs sentinel controlled loops

One Dimensional Array

A[i] where

- LB <= i <= UB else index out of bounds error
- A[i] = x iff A[i] := x preceeds the references else undefined data error

Fill An Array

```
set an array index variable to 0;
while more input, do the following:
   read a value;
    add 1 to the array index;
    store the value into the indexed array location
```

Process Every Element Of A One-Dimensional Array

```
for k := 1 to the array length do begin
   process element k of the array;
end;
```

Search A One-Dimensional Array

```
done := false;
k := 1;
while not done do begin
   if k > number of array elements then begin
         indicate search failure;
        done := true;
    end else if kth element satisfies search condition then begin
         indicate search success;
         done := true;
    end else begin
         k := k+1;
```

```
    end;
end;
```

Insert Into A One-Dimensional Array

```
{a new value is to be inserted into array at position}
for k:=length+1 down to position+1 do begin
   array[k]:=array[k387">1];
end;
array[position]:=the new value;
add 1 to the number of array elements;
```

Copy A One-Dimensional Array

```
(using a start position and a number of elements to copy)
identify the index for the startCopying location in the source array;
identify the index for the startAdding location in the destination array;
determine numberToCopy, the number of elements to copy;
for index:=0 to numberToCopy387">1 do begin
      newArray [startAdding+index] := originalArray [startCopying+index];
end;
```

Insert While Copying A One-Dimensional Array

```
while not finished inserting, do the following:
      copy from the source array to the destination array,
      up to the next insertion point;
add elements to be inserted at the end of the destination array;
```

420 Files

Name. Sequential File
Example.

```
PROGRAM name ( ...,file variable,...);
...
VAR
   file variable : file of type;
...
   assign( file variable, file name );
   reset( file variable );
...
   read( file variable, variables );
...
PROGRAM name ( ...,file variable,...);
...
VAR
   file variable : file of type;
...
   assign( file variable, file name );
   rewrite( file variable );
...
   write( file variable, variables );
...
   close( file variable );
```

```
...
```

Context.
Files are used to store data.
Problem.
The *file name* must be a string - either a constant or a variable.
Solution.

Name. Process the Elements in a File
Example.

```
reset the file;
while not end of file do begin
         read a file element;
         process the file element;
         end;
```

Context.
Problem.
Solution.

Name. Insert Into A File
Example.

```
rewrite (tempFile);
copy the contents of the data file to tempFile, up to the insertion point;
write the new element to tempFile;
copy the rest of the data file to tempFile;
rewrite (data file);
reset (tempFile);
copy tempFile to the data file;
```

Context.
Problem.
Solution.

Name. File Update
Example.

```
State := ReadBoth;
while State <> Done Do
   case State of
      ReadBoth    :    if eof(MF) then errors; State := Done
            else read(MF,...); State := ReadTF
      ReadTF      :    if eof(TF) then copy; copys; State := Done
            else read(TF,...); State := Process
      ReadMF      :    if eof(MF) then error; errors; State := Done
            else read(MF,...); State := Process
      Process     :    if MRid < TRid then copy; State := ReadMF
            else if MRid = TRid then update; State := ReadBoth
            else { MRid > TRid } error; State := ReadTF
   end
```

Context.
Update a master file from a transaction file assuming both files are ordered.
Problem.

Solution.

Two Dimensional Array

Process elements Until Done In A Two-Dimensional Array

- when all elements of the array are to be processed:

```
for row:=1 to the number of rows do begin
      for col:=1 to the number of columns do begin
            process the [row,col]th element;
      end;
end;
```

- when not all the elements are to be processed:

```
done:=false;
row:=1;
while not done and (row<= number of rows) do begin
      col:=1;
      while not done and (col<= number of columns) do begin
            process the [row,col]th element, possibly setting done to true;
            col:=col+1;
      end;
      row:=row+1
end;
```

Process The Diagonal Of A Two-Dimensional Array

```
initialize row and column variables to the position of the first element
      of the diagonal;
while row and column variables are still in bounds, do the following:
      process the array element at position [row,column];
      update the row variable:
            row:=row387">1 to go up (``northeast" or ``northwest" in the array);
            row:=row+1 to go down (``southeast" or ``southwest");
      update the column variable;
           column:=column+1 to go right (``northeast" or ``southeast");
            column:=column387">1 to go left (``northwest" or ``southwest");
```

Graph

ancestor type for linked dynamic data structures

Abstraction Composition

Abstraction permits the following reformulation of the Input, Process, Output template.

```
Procedure GetInput( Var Number : integer );
   BEGIN
      write('Enter a number: ');
      readln( Number );
   END;

Process( Item, Result : integer );
   BEGIN
      Result := ...Item...
   END;

DisplayResult( Result : integer );
   BEGIN
     writeln('The result is: ', Result)
   END;
...
   GetInput( Item );
   Process( Item, Result );
   DisplayResult( Result )
...
```

Recursive definition

Example.
Context.

Abstractions may be defined in terms of other abstractions and when an abstraction contains a self-reference, it is said to be defined *recursively*. The self-reference need not be in the immediate body of the definition but may occur in the body of some other abstraction needed to complete the definition. Well defined recursive abstractions have a structure similar to that of inductive definitions and proofs.

Problem.
Solution.
Additional Examples.
Divide And Conquer

```
DivideAndConquer: if zero elements remain, do something,
                  if one element remains, process it,
                  if more than one element remains, do the following:
                     divide the elements into groups;
                     (depending on the application)
                     either apply DivideAndConquer to each group,
                     or choose one of the groups apply DivideAndConquer to it
```

Examples: Quick sort, merge sort, towers of hanoi, maze traversal
Process in Reverse

```
ProcessInReverse: if there is a  value to process, then
                      ProcessInReverse the remaining values;
                  process the value that was part of this call;
```

Examples include: Factorial, power, print in reverse.

Design Patterns

A Patterns Catalog

Design patterns deal with *micro-architectures* (also known as *object structures*) -- static and dynamic relations among objects (and/or their classes) encountered in object-oriented development.

Design Patterns

Creational Patterns:

- AbstractFactoryPattern
- BuilderPattern
- FactoryMethodPattern
- PrototypePattern
- SingletonPattern

Structural Patterns:

- AdapterPattern
- BridgePattern
- CompositePattern
- DecoratorPattern
- FacadePattern
- FlyweightPattern
- ProxyPattern

Behavioral Patterns:

- ChainOfResponsibilityPattern
- CommandPattern
- InterpreterPattern
- IteratorPattern
- MediatorPattern
- MementoPattern
- ObserverPattern
- StatePattern
- StrategyPattern

- TemplateMethodPattern
- VisitorPattern

Structural decomposition - supports a controlled decomposition of an overall system task into cooperating subtasks.

- **Whole-Part** -

Organization of work -

- **Master-Worker** -

Access control -

- **Proxy** -

Management -

- **Command Processor** -
- **View Handler** -

Communication - patterns in this category help to organize communication between components.

- **Forwarder-Receiver** -
- **Client-Dispacher-Server** -
- **Publisher-Subscriber** -

References

- Cooper, James *Java design patterns: a tutorial* Addison-Wesley 2000
- Gamma et. al *Design patterns: elements of reusable object-oriented software* Addison-Wesley 1995
- Lea, Doug *Christopher Alexander: An Introduction for Object-Oriented Designers* Software Engineering Notes Vol 19 No 1 Jan 1994.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Functional Components

A Patterns Catalog

The functional components in a system can be classified as follows (from Ian Sommerville *Software Engineering 5th ed*)

- **Sensor**
- **Actuator**
- **Computation component**
- **Communication component**
- **Coordinator**
- **Interface**

Last Modified - . Comments and content invited aabyan@wwc.edu

Functional Pattern System

A Patterns Catalog

from Thomas Kühne *A Functional Pattern System for Object-Oriented Design* Verlag Dr. Kovac 1999

- Function Object - Black-box behavior parameterisation
- Lazy Object - Evaluation-by-need semantics
- Value Object - Immutable values
- Void Value - Abandoning null references
- Transfold - Combining internal & external iteration
- Translator - Homomorphic mapping with multi-dispatch functions

Last Modified - . Comments and content invited aabyan@wwc.edu

Architectural Patterns

A Patterns Catalog

Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.

From Mud to Structure

1. **Layers** - This architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.
2. **Pipes & Filters** - This architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allow you to build families of related systems.
3. **Blackboard** - This architectural pattern is useful for problems for which no deterministic solution strategies are known. Several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.
4. **Repository** -

Distributed Systems

1. **Broker** - This architectural pattern is a structure for distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.
2. **Client-Server**

Interactive Systems

1. **Model-View-Controller (MVC)** - This architectural pattern divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism insures consistency between the user interface and the model.
2. **Presentation-Abstraction-Control (PAC)** - This architectural pattern defines a structure for interactive software systems i the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three

components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

Adaptable Systems

1. **Microkernel** - This architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer -specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.
2. **Reflection** - This architectural pattern provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. In this pattern, an application is split into two parts. A meta level provides information about selected system properties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior.

References

- Buschmann et. al *Pattern-Oriented Software Architecture: A System of Patterns* Wiley1996
- Lea, Doug *Christopher Alexander: An Introduction for Object-Oriented Designers* Software Engineering Notes Vol 19 No 1 Jan 1994.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Program Errors

The real world of bad data and buggy code

The goal is to be able to

- Notify the user of an error
- Save all work
- Allow users to gracefully exit the program

There are two

Common Programming Errors

- Infinite Loops
- Division by zero and other arithmetic errors
- Undefined variables
- Missing special case, e.g. head of empty list
- Missing synchronizing element, e.g. missing send

Programming Techniques for Error Handling

- Stop the computation and report the source of the problem
  `error "error message"`
- Use dummy values. For example, define the tail of a list to be the empty list.
- Pass a dummy value to be used as an error value. For example, pass a value to the function which extracts the head of a list. The value is used only when the list is empty.

Error Handling

Once an error has been raised

- *transmit* the error through to the next higher routine
- *trap* the error and return

The Patterns

Name: Call-Error-Function-and-Quit

Example: error "argument to factorial is negative"
Context
Problem
Solution
Name: RaiseError
Example: throw ExceptionName (argument)
Context
Problem
Solution
Name: TryBlock
Example: **try** Code with raise exception **except** exception handler
Context
Problem
Solution

Pattern Templates

Pattern Template

Name	The name, a familiar descriptive name or phrase, usually indicative of the solution rather than the problem or context.
Description	Short summary of the pattern.
Also known as	Other names for the pattern, if any are known.
Example	A real-world example demonstrating the existence of the problem and the need for the pattern.
Context	Situations in which the pattern may apply. Often includes background, discussions of why this pattern exists, and evidence for generality
Problem	A description of relevant forces and constraints, and how they interact. Sometimes design and construction constraints.
Solution	Static relationships and dynamic rules describing how to construct artifacts in accord with the pattern, often listing several variants and/or ways to adjust to circumstances, references and relation to other patterns.
Structure	A detailed specification of the structural aspects of the pattern.
Dynamics	Typical scenarios describing the run-time behavior of the pattern.
Implementation	Guidelines for implementing the pattern.
Example resolved	
Variants	Description of variants or specializations of a pattern.
Known Uses	Examples of the use of the pattern, taken from existing systems.
Consequences	Benefits and any potential liabilities.
Depends on	
Is part of	
See also	References to patterns that solve similar problems, and to patterns that help us refine the pattern we are describing.
Credits	

Blank HTML template

Name

Description

Also known as
Example
Context
Problem
Solution
Structure
Dynamics
Implementation
Example resolved
Variants
Known Uses
Consequences
Depends on
Depended on by
See also
Credits

User Interface: Design & Patterns

Design principles

The following design principles apply to the look and feel of the user interface.

What works on paper, doesn't necessarily translate well to the web. And just because you can do something, doesn't mean you should. Here are a few design principles that should be kept in mind when considering your design:

- *Simplicity*: less is usually more - if a simple design will work, why complicate matters?
- *Elegance*: the web is still largely a visual medium, but visual should not be synonymous with garish.
- *Clarity*: what is clear to you must be clear to others.
- *Ease of use*: does the reader have to figure out how to get around?
- *Order*: is information where people expect to find it?
- *Consistency*: use a single look for your site, or at least for each section.
- *Accessibility*: consider the technological requirements of each feature - who will not be able to view your site?
- *Appropriate technology*: needless multimedia or interactivity is nothing more than eye-candy.
- *Access speed*: how long does each page take to load at the slowest speed?

-- Bitwalla Design

Eight golden rules of user interface design

1. Strive for consistency
2. Enable frequent users to use shortcuts.
3. Offer informative feedback.
4. Design dialogs to yield closure.
5. Offer error prevention and simple error handling.
6. Permit easy reversal of actions.
7. Support internal locus of control.
8. Reduce short-term memory load.

Schneiderman, Ben *Designing the User Interface* 3rd ed

Norman's Principles of Good Design

- *Visibility*. By looking, the user can tell the state of the device and the alternatives for action.
- *A good conceptual model*. The designer provides a good conceptual model for the user, with consistency in the presentation of operations and results and coherent, consistent system image.
- *Good mappings*. It is possible to determine the relationships between actions and results, between the controls and their effects, and between the system state and what is visible.
- *Feedback*. The user receives full and continuous feedback about the results of actions.

Norman, Donald. The Design of Everyday Things. Doubleday, 1990.

Design patterns

Prompt-Read
: This is used in interactive programs to prompt the user for input.

Display prompt;
Read input

Menu Driven Programming
Example.
: Here is an outline of the menu driven approach:

```
Procedure DisplayMenu;
BEGIN
   writeln('  Option Menu');
   writeln;
   writeln('D - Display Result');
   writeln('I - Input item');
   writeln('M - Display this menu')
   writeln('P - Process data');
   writeln('Q - Quit');
END;
...
DisplayMenu;
writeln;
write('Enter choice: ');
readln( Choice );
WHILE Choice <> 'Q' DO
   BEGIN
      ChooseFromAlternatives
```

```
        write('Enter choice: ');
        readln( Choice );
    END
```

The program actions are implemented with the **Choose from alternatives**template; the case statement approach should be used since it resembles the menu.

Context.

In menu driven programming, the user is given a menu of choices (a prompt), the program actions depend on the user's choice.

Problem.

Solution.

Validate Choice

Example.

```
Prompt-Read;
while not valid choice do
   Prompt-Read
end
```

Context.

This is used to make sure that the user has entered a valid choice.

Problem.

Solution.

References

IBM Ease of Use

Last Modified - . Comments and content invited aabyan@wwc.edu

Rule Based Programming

Finite State Machine (FSM)

What is a FSM?

- A directed graph with labeled edges.
- A collection of states, an initial state, a transition function.
- A transition table

Examples:

- Vending machine
- Automated teller machine
- Scanner

How is a FSM implemented?

- Loop and Case statement:

```
State :=  Start
while State <> Final do
   ...
   case State of
      Start : if input = a then State := State403"> ...
      ...
      Final : ...
   end
end
State := Start;
repeat
   case State of
      Start : ...
      ...
      Final : ...
   end
until State = Final
```

- Table driven FSM

```
state := start
loop
    state := table[state, input]
```

- Procedures (possibly recursive)

```
procedure Start( ... );
   begin
      ...
       State403">( ... )
   end

   ...

procedure Final( ... );
   begin
      ...
   end
```

Decision Trees

Context-free Grammars and PDA

Send comments to: webmaster@cs.wwc.edu

Pascal

Pascal Program

Example.

Pascal programs have the general structure:

PROGRAM *name* (*files*);
CONST *optional constant declarations*
TYPE *optional type declarations*
VAR *optional variable declarations*
optional procedure and variable declarations
BEGIN
The code section
END.

Context.

Problem.

All Pascal programs have the form given above.

- As a general rule, all names used in a program must be defined before they are referenced.
- Pascal is not case sensitive.
- If a program interacts with a user, then the list of files must include *input, output*.
- The constant section is used to define constants which are global to the program.
- ...

Solution.

270 Abstraction

Example.

Names for constants, types and variables as well as functions, and procedures are examples of abstraction.

Context.

Abstraction names an action, a value or composition of actions or composition of values and is used in conjunction with the other control stuctures to provide structure to programs, to provide for levels of abstraction, and to facilitate the reuse of code.

Problem.

Solution.

Name.
Function
Example.

```
Function Name ( Foramal Parameters ) : Type;
{ Pre: conditions for use }
{ Post: result   }
Body

... Name ( Actual Parameters ) ...
```

Context.
Functions are used to define a composition of expressions and may be used wherever a value may be used.
Problem.
Solution.
The body must contain the assignment: *Name* := *expression* to indicate the value of

Name.
Procedure
Example.

mmmmmmmmProcedure *Name* (*Formal Parameters*);
{ Pre: *conditions for use* }
{ Post: *result of use* }
Body

Name (*Actual Parameters*)

Context.
Procedures are used to define a composition of actions and may be used wherever a statement may be used.
Problem.
Solution.

280 Repetition Composition

Example.
recursive definition, while-do, repeat-until and for-do are examples of repetition composition.
Context.
Repetition composition is used to repeat an action or composition of actions. The while-do, repeat-until, for-do forms are special cases of recursive definition.

For example, to input and process items one-at-a-time.

Problem.
To insure that the repetition terminates there must be a stopping condition and a function to measure progress toward the stopping condition.
Solution.

Recursive Definition

Example.
Context.
Recursive definitions
Problem-Solution.
Names cannot be referenced before they are defined. Define recursive procedures and functions with *forward*.
Problem-Solution.
The execution of recursive definitions can be expensive in time and space if intermediate results are recomputed on recursive calls. Pass parameters by reference whenever possible; rewrite the definition in terms of other repeditive structures; store intermediate results to avoid recomputation.

Name.
While-Do
Example.

```
while condition do
    body
```

Context.
The while-do repetition is used when something is to be done zero or more times and usually the number of repetitions is not known.
Problem.
The *condition* must be defined and the *body* must perform some action to make the *condition* false eventually.
Solution.
Make sure that all variables appearing in the *condition* have been initialized and that the value of at least one is changed in the *body*.

Name.
Repeat-Until
Example.

```
repeat
   body
until condition
```

Context.

The repeat-until repetition is used when something is to be done at least once and usually the number of repetitions is not known.

Problem.

The *condition* must be defined and the *body* must perform some action to make the *condition* true eventually.

Solution.

Make sure that all variables appearing in the *condition* have been initialized and that the value of at least one is changed in the *body*.

Name.

For-Do

Example.

```
for index := low to high do
    body
Or the variant: for index := high downto low do body
```

Context.

The for-do repetition is used when the number of times something is to be done is known before-hand.

Problem.

If the *high* value is less than the *low* value, then the composition terminates. The For-do repetition will not perform as expected if either the *index* or limits are modified in the *body*.

Solution.

The *body* must not change the value associated with the *index*, the *low* or *high* condition otherwise termination will be unpredictable.

Send comments to: webmaster@cs.wwc.edu

Design Principles

problem solving - related to design as in design a solution to the problem...
design
creativity - related to problem solving where standard solutions are not available and to design when ...

Design: a process

Design as a verb refers to the process of devising something. Engineering design is the process of devising a system, component, or process to meet desired needs. It is a decision-making process (often iterative), in which the basic sciences and mathematics and engineering sciences are applied to convert resources optimally to meet a stated objective. Among the fundamental elements of the design process are the establishment of objectives and criteria, synthesis, analysis, construction, testing, and evaluation.

Engineering design includes most of the following features: creativity, open-ended problems, formulation of design problem statements and specifications, consideration of alternative solutions, feasibility considerations, production processes, concurrent engineering design, detailed system descriptions, and constraints such as economic factors, safety, reliability, aesthetics, ethics and social impact.

- Adapted from ABET

Software engineering shares with engineering the engineering design process. Software engineers have the ability to analyze, design, verify, validate, implement, apply, and maintain software systems and the ability to appropriately apply discrete mathematics, probability and statistics, and relevant topics in computer and management sciences to complex software systems.

- Adapted from software engineering: ABET Engineering Criteria 2000

Design: a plan

Design as a noun refers to some of the attributes of the product of the design process. The focus of this document is on the those qualities in a design that produce a preference for one design over another in objects that are intended to persist over time i.e., as software spends most of its life time in maintenance mode, we are interested software design that facilitates its own evolution. The focus of this paper is on the design of software not the process of software design.

Software is

- an implementation of a mathematical function
- a sequence of state changes (a thread of control)
- a simulation
- an executable theory
- a set of communicating processes
- a set of interacting objects

Software is designed to

- replace another system
  - through reverse engineering or
  - automation of preexisting system
- simulate
  - an existing system
  - explore alternative systems
- provide a previously unavailable service

In any case, software is analogous to a scientific theory with the added advantage of being executable facilitating its own testing.

Design - an esthetic

Engineering Design

Descriptive design - describes current practice. Designers typically follow these steps:

1. Understand their customer's needs (*requirements*).
2. Establish design objectives (*specifications*) to satisfy a given set of customer attributes. Define the problem they must solve to satisfy these needs.
3. Create and select the solution through *synthesis*.
4. *Analyze* and optimize the proposed solution
5. Check (*validate*) the resulting design against the customer's needs.
6. *Implement* the selected design

Prescriptive design - descrives how design should be done. Axiomatic design is one such prescriptive design methodology.

Axiomatic design

Axiomatic design was developed by Nam Suh. There are four main concepts in axiomatic design - domains, hierarchies, zigzagging, and design axioms.

Domains - The requirements specified in one domain are mapped in the design phases to a set of characteristic parameters in an adjacent domain.

Design phase	Design domain	Design elements/Phase activity
	Customer domain	- customer needs (CNs), the benefits customers seek
concept design		- customer's needs are identified and are stated in the form of required functionality of a product.
	Functional domain	- functional requirements (FRs) of the design solution - additional constraints (Cs)
product design		- a design is synthesized to satisfy the required functionality.
	Physical domain	- design parameters (DPs) of the design solution
process design		- a plan is formulated to implement the design.
	Process domain	- process variables (PVs)

The designer following the axiomatic design process

- produces a detailed description of what functions the object is to perform,
- a description of the object that will realize those functions, and
- a description of how this object will be produced.

The information about which part(s) of the object perform or affect which functions, as well as what manufacturing process variables(s) affect which physical parts in the object is captured in design matrices. Entries in matrix **A** describe dependencies between the FRs and the DPs.

A	DP1	...	DPm
FR1			
...			
FRn			

Entries in matrix **B** describe dependencies between the DPs and the PVs.

B	PV1	...	PVm
DP1			
...			
DPn			

If the design matrix is a diagonal matrix the design is an *uncoupled design.* Each functional requirement is implemented by just one design parameter. Entries in matrix A describe dependencies between the FRs and the DPs.If it is a triangular matrix, the design is a *decoupled design*. Any other matrix describes a *coupled design*.

Functional requirements (FRs) are a minimum set of independent requirements that completely characterize the functional needs of the design solution in the functional domain.

Some general requirements are that the resulting product must

- work
- be safe
- be economical
- be reliable
- meet the needs of the customer

Design parameters (DPs) are the elements of the design solution in the physical domain that are chosen to satisfy the specified FRs.

Constraints (Cs) are bounds on acceptable solutions.

Process variables (PVs) are the elements of the process domain that characterize the process that satisfies the specified DPs.

Hierarchies - The output of each domain evolves from abstract concepts to detailed information in a top-down or hierarchical manner.

Zigzagging - The designer goes though a process whereby he/she zigzags between domains in decomposing the design problem. The result is a hierarchical development process in each domain is performed in conjunction with that in the other domains.

Design axioms - there are two design axioms about the relations that should exist between FRs and DPs which provide a rational basis for evaluation of proposed solution alternatives and the subsequent selection of the best alternative.

Independence Axiom: maximize the independence of the functional requirements.

- Orthogonality

The application of the Independence Axiom is described in terms of the design matrix. A diagonal matrix (uncoupled design) is maximally independent.

Information Axiom: minimize the information content of the design (maximize the probability of success).

- Methods
    - Repetition (regularity)
    - Similarity
    - Closure
    - Proximity
    - Reuse
- Simplicity

but ?complete - spanning?

The resulting product must

- work,
- be safe,
- be economical,
- be reliable, and
- meet the needs of the customer

References

- Suh, Nam P. (1990) *The Principles of Design* Oxford University Press
- Suh, Nam P. (2001) *Axiomatic Design: Advances and Applications* Oxford University Press
- McPhee, Kent (1997) *Design Theory and Software Design* Technical Report TR 96-26. revised 1997.

Database Design Principles
User interface Design Principles
Software Design Principles
Design Patterns

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Problem Solving

Two groups of problems people face:

- **Well defined problems** In a well-defined problem it is clear what the problem is and the solution is clearly specified as well. That is, the solution can be recognized clearly when arrived at. Well defined problems often have generally known solutions.
    - They are solved using standard methods, methods of similar problems, or methods of analogous problems

      My problem -> Analogous standard problem -> Analogous standard solution -> My solution
    - Examples: puzzles, simple games, and lower level mathematics, science, and engineering.
- **Ill-defined problems** In an ill-defined problem the problem it is not clear from the beginning of what the problem is and thus, what a solution is. Thus, finding a solution requires in addition to find out what the real problem is. Solving and specifying the problem develop in parallel and drive each other. Ill-defined problems usually have unknown solutions
    - The solutions are often such that they still could be improved and it is up to the problem solver to decide when enough is enough.
- **Wicked problems** Wicked problems are similar to ill-defined problems, just much worse. Furthermore solutions are very difficult, if at all, to recognize as such. In other words, stating the problem is the problem.
    - often contain contradictory requirements
    - often the problem changes over time
    - there is uncertainty if the offered solution is the best solution or is even a solution
    - requires an inventive/creative solution

      My problem -> Inventive insight -> My solution

Most problem solving is done (and taught) from the perspective of a particular domain. So, for example, problems in the various branches of mathematics and the various academic disciplines with their own courses teaching their own problem solving methods. The distribution of statistics classes among academic departments illustrates both the felt need to focus on domain specific problems and the existence of general problem solving methods.

An important hypothesis in AI is that all intelligent problem solving can be characterized as a search process (Newell and Simon, 1976). Problem solving methods (PSMs) are domain-independent reasoning components, which specify patterns of behavior which can be reused across applications.

References

Newell, A. and Simon, H. A. (1976). Computer Science as Empirical enquiry: Symbols and Search. Communications of the ACM, 19(3), pp. 113-126, March.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Creativity

Creativity: The ability to generate new ideas or synthesize new solutions in the absence of prior examples or paradigms.

Three Perspectives on Creativity

1. Inspirationalists
    - emphasis: the remarkable "Aha!" movements (luck favors the prepared mind)
    - techniques
        - brainstorming
        - free association
        - lateral thinking
        - divergence
2. Structuralists
    - emphasis - exhaustive exploration of possible solutions
    - techniques
        - studying of previous work
        - methodical techniques such as Polya's four steps
            1. Understanding the problem
            2. Devising a plan
            3. Carrying out the plan
            4. Looking back
    - techniques
3. Situationalists
    - emphasis - social and intellectual context as the key part of the creative process
    - components of creativity
        1. domain: a set of symbols, rules, and procedures
        2. field which determines whether an new idea, performance, or product should be included in the domain
        3. creative individual who sees a new idea or pattern

Levels of Creativity

- Rare revolutionary events (breakthroughs and paradigm-shifting innovations)
- Normal science - useful evolutionary contributions that refine and apply existing paradigms
- Impromptu or personal creativity

Genex

The four phase genex framework:

Four Phases	Eight Activities
Collect: learn from previous works stored in libraries, the Web, etc	*Searching* and browsing digital libraries *Visualizing* data and processes
Relate: consult with peers and mentors at early, middle, and late stages	*Consulting* with peers and mentors
Create: explore, compose, and evaluate possible solutions	*Thinking* by free associations *Exploring* solutions - what if tools *Composing* artifacts and performances *Reviewing* and replaying session histories
Donate: disseminate the results and contribute to the libraries	*Disseminating* results

References

- Shneiderman, Ben.2000. Creating Creativity: User Interfaces for Supporting Innovation *ACM Transactions on Computer-Human Interaction*, Vol 7, No. 1, March 2000, Pages 114-138.

TRIZ - The theory of inventive problem solving

Genrich S. Altshuller

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Database Design Principles

- *Faithfulness*: the design should be faithful to the specifications.
- *Avoid Redundancy*: say everything once only.
- *Simplicity*: avoid introducing more elements than are absolutely necessary.
- *Right kind of element*: attributes are easier to implement but entity sets and relationships are necessary.

Last Modified - . Comments and content invited aabyan@wwc.edu

User Interface Design Principles

Six Dimensions of Activity

Austin Henderson identifies six dimensions along which to describe the activity in which people take part while making use of technology.

- Operating Activities - the mechanics of users making the machine do what they want
    1. Trouble
        - Activity: People get into trouble often out of partial information
        - Design requirement: Help people get out of trouble
    2. Users
        - Activity: Technology has many users
        - Design requirement: Design for the needs of all the users.
- Enabling Activities - the activities carried out by users to enable the operating activities.
    1. Support
        - Activity: Understanding that the support of doing requires activities associated with knowing, changing, and managing
        - Design requirement: Support knowing, technology modification, and resource management.
    2. Practices
- Empowering Activities
    1. Values
    2. Designers

Reference

Henderson, Austin. Design for What? Six Dimensions of Activity *ACM Interactions Vol VII.5* Sept & Oct 2000 pp. 17-22.

Web and user interface design principles

What works on paper, doesn't necessarily translate well to the web. And just because you can do something, doesn't mean you should. Here are a few design principles that should be kept in mind when considering your design:

- *Simplicity*: less is usually more - if a simple design will work, why complicate matters?
- *Elegance*: the web is still largely a visual medium, but visual should not be synonymous with garish.
- *Clarity*: what is clear to you must be clear to others.

- *Ease of use*: does the reader have to figure out how to get around?
- *Order*: is information where people expect to find it?
- *Consistency*: use a single look for your site, or at least for each section.
- *Accessibility*: consider the technological requirements of each feature - who will not be able to view your site?
- *Appropriate technology*: needless multimedia or interactivity is nothing more than eye-candy.
- *Access speed*: how long does each page take to load at the slowest speed?

-- Bitwalla Design

Eight golden rules of user interface design

1. Strive for consistency
2. Enable frequent users to use shortcuts.
3. Offer informative feedback.
4. Design dialogs to yield closure.
5. Offer error prevention and simple error handling.
6. Permit easy reversal of actions.
7. Support internal locus of control.
8. Reduce short-term memory load.

Schneiderman, Ben *Designing the User Interface* 3rd ed

Norman's Principles of Good Design

- *Visibility*. By looking, the user can tell the state of the device and the alternatives for action.
- *A good conceptual model*. The designer provides a good conceptual model for the user, with consistency in the presentation of operations and results and coherent, consistent system image.
- *Good mappings*. It is possible to determine the relationships between actions and results, between the controls and their effects, and between the system state and what is visible.
- *Feedback*. The user receives full and continuous feedback about the results of actions.

Norman, Donald. The Design of Everyday Things. Doubleday, 1990.

References

- IBM Ease of Use

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Software Design Principles

General software design principles

- *Correct & complete*: The design should correctly implement a specification.
- *Maximize Cohesion*: Cohesion describes how well the contents of a module cohere (stick together). A component should implement a single logical function or should implement a single logical entity.
- *Minimize Coupling*: Coupling describes how modules interact. Systems should be loosely coupled. Highly coupled systems have strong interconnections with units dependent on each other. Loosely coupled systems are made up of components which are independent or almost independent.
- *Understandabilty*: A design must be understandable if it is to support modification.
- *Adaptability*: The design must be easy to change.

Characteristics of good and bad design - Beck

Good Design	*Bad Design*
• Change in one part of the system doesn't always require a change in another part of the system. • Every piece of logic has one and one home. • The logic is near the data it operates on. • System can be extended with changes in only one place. • Simplicity	• One conceptual change requires changes to many parts of the system. • Logic has to be duplicated. • Cost of a bad design becomes overwhelming. • Can't remember where all the implicitly linked changes have to take place. • Can't add a new function without breaking an existing function. • Complexity

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Temporal Logic for Specification

Anthony Aaby
Walla Walla College
aabyan@wwc.edu
Last Modified: .
Comments and content invited: aabyan@wwc.edu

Temporal logic is ordinary logic extended with temporal operators [] (read *henceforth*) and <> (read *eventually*). The formula []*P* asserts that *P* is true now and at all future times, and the formula <>*P* asserts that *P* is true now or at some future time. Since *P* is eventually true if and only if it is not always false, <>*P* is equivalent to ~[]~*P*.

Temporal logic, as it has been defined here, cannot formally specify things like average response time and probability of failure. However, it is useful for the specification of safety and liveness properties. *Safety properties* assert what the system is allowed to do, or equivalently, what it may not do. Safety properties are satisfied by a system which does nothing. Restriction to only producing correct answers is an example of a safety property. *Liveness properties* assert what the system must do. Termination is an example of a liveness property.

As an example of temporal specifications and safety and liveness specifications in particular, we provide a specification of the *The Dining Philosophers Problem.* Five philosophers spend their lives seated around a circular table thinking and eating. Each philosoper has a plate of spaghetti and, on each side, shares a fork his/her neighbor. To eat, the philosopher must aquire two forks. The problem is to prevent deadlock or starvation i. e. insure that each philosopher gets to eat.

Figure 1: Safety and Liveness Specifications: Philosopher P(i)

Safety Properties	[]($eating(i) \vee thinking(i)$)	Philosophers either eating or think
	[]~($eating(i) \vee eating(i+1)$)	Adjacent philosophers cannot eat simultaneously
Liveness Properties	[]($thinking(i) \rightarrow$ <>$eating(i)$)	Philosophers alternate between eating and thinking
	[]($eating(i) \rightarrow$ <>$thinking(i)$)	

Fairness is a desirable property of a concurrent system and is definable as a liveness property.

Generalized weak fairness <>[]P => []<>A

Generalized strong fairness []<>P => []<>A

Formally, an *action system* consists of an initial state predicate *Init* and a set of predicates A_i on pairs of states. The A_i are called *system actions*. An action system expresses the safety propery consisting of every behavior $<s_0, s_1, ... >$ whose initial state s_0 satisfies *Init* and whose every pair $<s_i, s_{i+1}>$ of successive states satisfies some system action.

Exercises

Fairness: Produce in temporal logic a definition of fairness.

For each of the following, produce temporal logic specifications. Clearly indicate the safety and liveness conditions.

Soda machine

Upon accepting 50 cents in quarters or half-dollars, the soda machine dispenses a soda.

Producer/Consumer Problem (Bounded Buffer)

There is a pool of n buffers that are filled by one or more producer processes and emptied by a consumer process. The problem is to prevent an overlap of buffer operations and to keep the producer from overwriting full buffers and the consumer from reading empty buffers.

Readers and Writers

There is a shared data structure. There are *reader* processes and *writer* processes. The following conditions must be satisfied.

1. Any number of readers may simultaneously read the data.
2. Only one writer at a time may write to the data structure.
3. If a writer is writing to the file, no reader may read it.

Any waiting reader or writer must eventually have access to the data structure.

The Barbershop Problem

The barber shop has *n* barbers, *n* barber chairs, and a waiting area with a sofa, There is a limitation of *m* customers in the shop at a time. The barbers divide their time between cutting hair, accepting payment, and sleeping in their chair waiting for a customer. A customer will not enter the shop if it is filled to capacity. Once inside, the customer takes a seat on the sofa or stands if the sofa is full. When a barber is free, the customer that has been waiting the longest on the sofa is served and the customer that has been standing the longest takes its place on the sofa. When a haircut is finished any barber can accept payment but payment can be accepted for only one customer at a time as there is only one

cash register.

User interface

Describe the user interface of some system in terms of an action system.

Colloquia

Jobs

Presented
98.05.13

- Review of the job market for CS majors
  - Growth rate 112% / year
  - 190,000 IT jobs going unfilled
  - Oracle DB entry level programmer $50,000
  - Borland (Inprise) Dale Lampson (WWC grad)
    - CORBA entry level programmer $65,000
    - MA $83,000
  - WebMaster: $45,000 to $121,200
- What do employers ask your teachers about you?
  - Homework in on time?
  - Reliable, dependable
  - Work well in group?/Communication skills (writing)/Documentation
  - Creativity -- standard solutions or something extra
  - Problem solving skills
    - solo problem solver or
    - uses resources
- Strategies for personal development
  - Extra courses
  - Summer/Year Coop positions
  - Do something to stand out (i.e. distinguish yourself from the rest of your class)

Hot Summer: Distance Learning Opportunity

1. Web Publishing and Design
2. Developing Web Applications
3. Managing Web Services
4. Learning Perl and CGI
5. Programming Perl with Databases
6. Programming and Designing with JavaScript
7. Learning Java on the Web
8. Programming Java Applications
9. Developing Microsoft's Active Platform
10. Developing Secure commerce Applications

Letters of Recommendation

Industry

Quaifications

- Ability to work with others
- Ability to organize and express ideas clearly

Graduate School

Performance Categories

- Outstanding (top 5%)
- Very Good (top 10%)
- Good (top 25%)
- Average (upper 50%)
- Below average (lower 50%)

Quaifications

- Performance in independent study or research groups
- Intellectual independence
- Research Interests
- Capacity for analytical thinking
- Ability to work with others
- Ability to organize and express ideas clearly
- Drive and motivation

The Surface of Truth

Presented

98.03.04 @ wwc

Abstract Standard formulations of logic include the use of truth tables with just two values: True, False. Tow values do not capture the full meaning of ordinary statements such as, " It is hot". Such statements require a possibly infinite range of truth values. Many attempts have been made to extend logic to such ranges. Fuzzy logic is one such attempt. This talk explores the options that are available to the designer of an infinite valued logic. Applications of such logic systems include expert systems and sophisticated control systems.

- Review of propositional logic including truth tables
- What is wrong with propositional logic
  - static temporality
  - binary values
- Modal logic
  - Modal operators
- Infinite valued logic
  - truth table generalization -- max, min
- Truth surfaces and graphing with Maple
- But which formula?
- Reasoning with infinite valued logic
- The Future
  - formula selection
  - implementation opportunities

CS & CpE: Two Sides of the Same Coin

Presented
98.09.30

Abstract:

Computer science and electrical engineering departments at universities are merging. Computer engineering is emerging as the hot discipline within engineering. The ABET and CSAB accrediting organizations are talking about merging. Speculation is rife about licensing software engineer. Thousands of IT jobs available. What does it all mean. More question than answers.

- Welcome
- Review of the IT job market
    - Growth rate 112% / year
    - 190,000 IT jobs going unfilled
    - Oracle DB entry level programmer $50,000
    - Borland (Inprise) Dale Lampson (WWC grad)
        - CORBA entry level programmer $65,000
        - MA $83,000
    - WebMaster: $45,000 to $121,200
- What do employers ask your teachers about you?
    - Time management
        - Homework in on time?
        - Reliable, dependable
    - Work well in group?/Communication skills (writing)/Documentation
    - Creativity -- standard solutions or something extra
    - Problem solving skills
        - solo problem solver or
        - use of resources
- Strategies for personal development
    - Extra courses

- Summer/Year Coop positions
- Do something to stand out (i.e. distinguish yourself from the rest of your class)

CS: Applied Algebra

This document uses MathML, Amaya is an appropriate bowser for viewing MathML.

Abstract There is a close relationship between computer science and mathematics. In fact "Software is a kind of mathematics" - Davis & Hersh, and programmers are among the best athletes in the formalist game (Formalism: games with symbols and rules for manipulating those symbols). This is a presentation on the use of many sorted algebras for the definition of abstract data types and programs.

More

Presented 1999.01.22 @ wwc

 Send comments to aabyan@wwc.edu

A Survey of Automated Reasoning

This document uses MathML, Amaya is an appropriate bowser for viewing MathML.

Abstract: Automated reasoning systems have made tremendous strides over the past 20 years. They are now used to verify processor and other hardware designs and to find proofs for interesting mathematical theorems that have resisted conventional approaches. The available software includes large automated as well as interactive systems. LeanTAP is a small, fast, and simple system which can be understood by beginning programmers.

- Goals
  - Artificial intelligence - artificial mathematitians
  - Intelligent assistant/tutor
  - Proofs of correctness for hardware designs
  - Proofs of correctness of software
- Available Systems (anl)
- Results
- Analytic Tableaux
- Research opportunities

More

Presented 1999.02.24 @ wwc cs colloquium

 Send comments to aabyan@wwc.edu

Y2K - We told you so!

observations, opinions, questions; no answers.

Introduction

As a computer scientist, I don't have much to say. There was a problem. We fixed it. Nothing happened.

So I will pretend that I am a social scientist and now I find that I have plenty to say.

If a catastrophe would have happened, those with 6 months supply of food, water, gasoline, and a generator would look like geniuses instead of idiots.

The Y2K Computer Bug

- **What is it**? Example: the four character department course identifier (EnvSci). This is how information systems are designed.
- What is **the problem with dates**? Arithmetic!
  - Ambiguity **19**00 != **20**00
  - Order 99 !< 00
  - Subtraction is noncommutative (5-3 != 3-5) -- underflow error 00-99=?
- What are the **consequences**?
  - actions based on date
  - actions based on out of order errors
  - actions based on length of time
  - billing errors
  - system crash
  - chain reaction through networked applications b2b transactions
- What about **embedded systems**, air traffic control, missile defense systems, power plants
  - Redundant systems, manual overrides etc.
- What about the **3rd World**?
  - far less dependent on technology
  - used to coping with failure
- What about the **US**? How do Americans cope with disaster?
  - can do spirit
  - resilient culture
  - *Where are the historians and social scientists when you need them?*
- How much was spent? $100 billion +; near $200 billion world wide.

- Was it well spent? Yes and No - needed upgrades
- Was **too much spent?** Yes, billions were spent to reassure a panicking public.
- **Is it over? No! leap year, stop gap measures - windowing...** But we are used to coping with technological failures.

The Y2K World Celebration

- **ABC & PBS**'s fantastic coverage of the New Year's Day celebrations
  - small world
  - wonderful diversity of culture & religion
- *We must add international perspective (16 hours) to our general studies requirements*
  - Christian service volunteers
  - Studies abroad
  - non US history
  - World religions
  - International commerce
  - Anthropology
  - & other

The Y2K Superstition: numerology, mysticism, & religion

- **Why did people panic?** Are people naturally superstitious?
- What is the **trigger level for action?** Did the Y2K problem meet **your discipline's criteria for truth/action?** As a computer scientist,
  - I recognized the existence of a problem,
  - realized the possibility of significant problems. I needed to know the probability of significant problems.
  - I looked for but did not see a single example that demonstrated that there would be anything other than a minor annoyances common to our current technological level.
  - So I sat back and followed the progress reports of remediation.
- Crises
  - Rational evaluation of threat and the difference between possible and probable threats.
  - Rational preparation for a threat.
  - Rational response to a threat.
  - The search for certainty/security. The fear of failure. *Academics can't be wrong.*
- WWC and Y2K
  - Was WWCs response to Y2K rational?
  - Maintenance of emergency equipment ...
  - Were WWCs preparations those appropriate for the times and place?
- *We must teach ourselves and our students the difference between possible and probable.* Flip 4 coins.
  - Is it possible to get all tails? Is it probable that you will get all tails? (probability of

1/16).
    - Is it probable that you will get half tails? - probability is 6/16=3/8; it is most probable that you will not get half tails - probability is 10/16 = 5/8.
- How does the Y2K situation differ from methods used in evangelism? Do we deal honestly with people? Do we play on their fears and credulity? Are we honest with the facts? Are the choices really black and white?
- *We must wean ourselves and our students away from binary logic and teach then about multivalued logic.*
    - Are you married? When did it happen?
        - When the preacher said "I now pronounce you..."
        - When you both signed the license?
        - When you consummated the marriage?
        - **Common law**: After you lived together for x years and had a joint bank account?
    - Are you sure you are still married?

Y2K can be as important as the fall of Rome (Alon Peled)

Social science perspectives:

1. Political philosophy (misfortune or injustice) The exhaustive testing of a simple program designed to monitor 100 binary signals would take billions of years. "For the modern person, technology has grown to assume the role of the pagan god in ancient times--we can blame it for its cruel arbitrary nature but there is little we can do to reassure that we will not get hit by a similar disaster the next time around."
2. Organizational Theory - who is to blame?
    - many minor innocent errors can add up to a major technological failure.
    - many clean, reliable, well-tested, efficient standalone systems can produce a calamity when they interact.
3. Sociology - when current thinking suggests that all countries are enroute to adopt western lifestyle we need to "identify when and why individual nations resort to their traditional cultural faultlines."
    - Self help (USA)
    - Contract (England)
    - Act of God (Europe)
    - Hangman (China)
4. Business and Management (Design of Organizational Information Systems - the problem is not a computer science problem but a business information systems problem) - "none of the schools of thought has been able to explain very well how organizations have chosen to tackle the imminent Y2K disaster."

The problem with social scientists is that they think that technology belongs to engineers and technologists to study while "people" are theirs to study.

Virtually every interesting social challenge today requires us to know a great deal about both people and technology as exemplified by recent scholarly debates about the environment, global warming, the future of high tech warfare, and the emergence of virtual Internet communities.

Social scientists have missed a valuable opportunity for data collection.

Technology and the Information Age

- The Internet has created a new world in which
  - there are no natural barriers to restrict the free flow of information
  - there are no natural barriers to prevent the spread of viruses
  - there is a high degree of interdependence on an international scale
  - there is a distributed (rather than centralized) informational authority (libraries, academic institutions)
  - **centralized and hierarchical authority is dead**
  - Education & Libraries
    - Encyclopedia Britanica
    - www.learn2.com
- The rise of Open Source, Open standards vs. proprietary standards
  - Academic tradition
  - Antiquated intellectual property rights (both copyright and patent law) restrict access to the information necessary to control our own destiny.
  - 3rd World countries and Information imperialism - China and Red Flag Linux
  - Rise of international commerce and the power of the multinational corporations have more power than government
  - DVD - linux
- Loss of privacy: no more secrets
- Evolution and Complexity
  - Emergence of complexity
  - How shall we respond to the increasing complexity of society?
  - The Internet and computing should be studied using the methods that biologists use.
  - Evolutionary theory needs to be widely used by all disciplines to study change and complexity.
  - We need a general theory of life that will guide both biological environmentalists as well as technological and social environmentalists.
- What curricular reform do we need to do here at WWC? **We need a curriculum for the 21st century not the 19th century.**
- Key trends
  - accelerating pace of technological change
  - the enormous size of the world population
  - ease and volume of travel and communication
  - interdependency of world economies

- Emerging complexity and interconnectivity of technological systems
    - theory of complex systems
    - theory of change
    - social consequences of technology
- Recommendations:
    - technology and society studies
    - general studies technology requirement
    - Generalized theory of change (evolution)
        - religion
        - society
    - Theory of life (where are we going)
        - environmentalists focus on preservation inspite of the fact of catastrophic change
    - *Social sciences must recognize that technology plays a far more dominant role than the natural world in the lives of most people. And they must take a more active role in studying technological issues and the evolution of society.*

References

Capurro, Rafael
Towards an Information Ecology 1990
Kalmykov, Vyacheslav L.
The Generalized Theory of Life http://www.stormloader.com/theory
Peled, Alon
Why Did Social Scientists Miss the Bug? *Computers & Society Vol. 29 No. 4*

previous next

The Triumph of Superstition, Numerology, Mysticism, & Religion

- What does it mean to end a millennium and begin a *new* millennium?
- Why do numbers/anniversaries have meaning?
    - Time/calendar - arbitrary beginning, arbitrary units
- Is superstitious behavior an innate human quality or can we inoculate society against superstitious behavior?
- Just as religious beliefs sparked panic in 1000, it was a source of panic in 2000.
- What is the relation between religion and superstition?
- Does religion encourage superstitious behavior?
- What is the difference between religious authority and technological authority?
- What is the difference between religious mysticism and technological mysticism?

Time for Opportunists - superstition and crowds

- political terrorists

  spiritual terrorists &demagogues

Proving programs correct

Vienna Development Method (VDM)

Template for VDM specification

heading [signature]	Domain, range for a given function: Domain set > range set Module name [for modules]
[ext external-variables(s)	External variables defining the state of a function
pre [condition]	Assumptions about the arguments in the domain of the function
post [condition]	Assumptions about the result of applying the function to the values of its arguments.

VDM Notation

	Notation	Explanation
Specification	f(arg: Type) result: Type ext rd variable-name: Type wr varialbe-name: Type f: D1 x D2 > range f(D)	function read-only write/read signature

Logic	$\wedge$	and
	$\vee$	or
	$\Rightarrow$	implies
	$\Leftrightarrow$	equivalent
	$\forall$	all
	$\exists$	exists
	$\vdash$	derives
Set	$x \in A$	membership
	$t \notin A$	not a member
	$\{\}$	empty set
	$\neg A$	not A
	$B \subset A$	B subset of A
	$B \subseteq A$	strict set member
	$A \cap B$	Intersection
	$A \cup B$	Union
	A-B	Difference
	card A	Size of A
Proof	∀ x ∈ A . property (x)	All x in A such that property (x) is true
	$\exists$x ∈ A . property (x)	Exists x in A such that property (x) is true
	hypothesis conclusion	Sequent (hypothesis derives conclusion)

Specifying with Z(ed)

Z is a notation for descring the behavior of sequential processes. It is based on elemenatary set theory and logic. A Z description of the functionality of a software system consists of a collection of structures called schemas.

Schema Identifier
Declaration-part
{status} State space identifier
Variable declaration
Function declaration
Optional pre-condition
Optional post-condition

SchemaDeclaration ::=

{status} Identifier

Identifier: set_type |

Identifier: domain range

Identifier {?|!}: set_type

Status ::= Δ |

Axiomatic Semantics

The *axiomatic semantics* of a programming language are the assertions about relationships that remain the same each time the program executes. Axiomatic semantics are defined for each control structure and command. The axiomatic semantics of a programming language define a *mathematical theory* of programs written in the language. A mathematical theory has three components.

- **Syntactic rules:** These determine the structure of formulas which are the statements of

interest.
- **Axioms:** These describe the basic properties of the system.
- **Inference rules:** These are the mechanisms for deducing new theorems from axioms and other theorems.

The semantic formulas are triples of the form:

$$\{P\}\ \mathbf{c}\ \{Q\}$$

where **c** is a command or control structure in the programming language, P and Q are *assertions* or statements concerning the properties of program objects (often program variables) which may be true or false. P is called a *pre-condition* and Q is called a *post-condition*. The pre- and post-conditions are formulas in some arbitrary logic and summarize the progress of the computation.

The meaning of

$$\{P\}\ \mathbf{c}\ \{Q\}$$

is that if **c** is executed in a state in which assertion P is satisfied and **c** terminates, then **c** terminates in a state in which assertion Q is satisfied. We illustrate axiomatic semantics with a program to compute the sum of the elements of an array (see Figure N.3).

Figure N.3: Program to compute $S = \text{sum}_{i=1}^{n} A[i]$

```
S,I := 0,0
while I < n do
   S,I := S+A[I+1],I+1
end
```

The assignment statements are *simultaneous* assignment statements. The expressions on the righthand side are evaluated simultaneously and assigned to the variables on the lefthand side in the order they appear.

Figure N.4 illustrates the use of axiomatic semantics to verify the program of Figure N.3.

Figure N.4: Verification of $S = \text{sum}_{i=1}^{n} A[i]$

Pre/Post-conditions	**Code**

1. { 0 = $Sum_{i=1}^{0}$A[i], 0 < |A| = n }
2. S,I := 0,0
3. {S = $Sum_{i=1}^{I}$A[i], I <= n }
4. while I < n do
5. {S = $Sum_{i=1}^{I}$A[i], I < n }
6. {S+A[I+1] = $Sum_{i=1}^{I+1}$A[i], I+1 <= n }
7. S,I := S+A[I+1],I+1
8. { S = $Sum_{i=1}^{I}$A[i], I <= n }
9. end
10. {S = $Sum_{i=1}^{I}$A[i], I <= n, I >= n }
11. {S = $Sum_{i=1}^{n}$A[i] }

The program sums the values stored in an array and the program is decorated with the assertions which help to verify the correctness of the code. The pre-condition in line 1 and the post-condition in line 11 are the pre- and post-conditions respectively for the program. The pre-condition asserts that the array contains at least one element zero and that the sum of the first zero elements of an array is zero. The post-condition asserts that S is sum of the values stored in the array. After the first assignment we know that the partial sum is the sum of the first I elements of the array and that I is less than or equal to the number of elements in the array.

The only way into the body of the while command is if the number of elements summed is less than the number of elements in the array. When this is the case, The sum of the first I+1 elements of the array is equal to the sum of the first I elements plus the I+1st element and I+1 is less than or equal to n. After the assignment in the body of the loop, the loop entry assertion holds once more. Upon termination of the loop, the loop index is equal to n. To show that the program is correct, we must show that the assertions satisfy some verification scheme. To verify the assignment commands, we use the *Assignment Axiom*:

Assignment Axiom
{P[x:E]} x:= E {P}

This axiom asserts that:

> If after the execution of the assignment command the environment satisfies the condition P, then the environment prior to the execution of the assignment command also satisfies the condition P but with E substituted for x (In this and the following axioms we assume that the evaluation of expressions does not produce side effects.).

An examination of the respective pre- and post-conditions for the asssignment statements shows that

the axiom is satisfied.

To verify the while command of lines 4. 7 and 9, we use the *Loop Axiom*:

Loop Axiom:

$$\frac{\{I \wedge B \wedge V > 0\}\ C\ \{I \wedge V > V' >= 0\}}{\{I\}\ \texttt{while B do C end}\ \{I \wedge \neg B\}}$$

The assertion above the bar is the condition that must be met before the axiom (below the bar) can hold. In this rule, {I} is called the *loop invariant*. This axiom asserts that:

> To verify a loop, there must be a loop invariant I which is part of both the pre- and post-conditions of the body of the loop and the conditional expression of the loop must be true to execute the body of the loop and false upon exit from the loop.

The invariant for the loop is: $S = sum_{i=1}^{I} A[i]$, I <= n. Lines 6, 7, and 8 satisfy the condition for the application of the Loop Axiom. To prove termination requires the existence of a *loop variant.* The loop variant is an expression whose value is a natural number and whose value is decreased on each iteration of the loop. The loop variant provides an upper bound on the number of iterations of the loop.

> A variant for a loop is a natural number valued expression V whose run-time values satisfy the following two conditions:
>
> - The value of V greater than zero prior to each execution of the body of the loop.
> - The execution of the body of the loop decreases the value of V by at least one.

The loop variant for this example is the expression n - I. That it is non-negative is guaranteed by the loop continuation condition and its value is decreased by one in the assignment command found on line 7. More general loop variants may be used; loop variants may be expressions in any well-founded set (every decreasing sequence is finite). However, there is no loss in generality in requiring the variant expression to be an integer. Recursion is handled much like loops in that there must be an invariant and a variant. The correctness requirement for loops is stated in the following:

> **Loop Correctness Principle:**Each loop must have both an invariant and a variant.

Lines 5 and 6 and lines 10 and 11 are justified by the *Rule of Consequence.*

Rule of Consequence:

$$\frac{P \rightarrow Q,\ \{Q\}\ C\ \{R\},\ R \rightarrow S}{\{P\}\ C\ \{S\}}$$

The justification for the composition the assignment command in line 2 and the while command

requires the following the *Sequential Composition Axiom.*

Sequential Composition Axiom:

$$\frac{\{P\}\ C0\ \{Q\},\ \{Q\}\ C1\ \{R\}}{\{P\}\ C0;\ C1\ \{R\}}$$

This axiom is read as follows:

> The sequential composition of two commands is permitted when the post-condition of the first command is the pre-condition of the second command.

The following rules are required to complete the deductive system.

Selection Axiom:

$$\frac{\{P \wedge B\}\ C0\ \{Q\},\ \{P \wedge \neg B\ \}\ C1\ \{Q\}}{\{P\}\ \text{if B then C0 else C1 fi}\ \{Q\}}$$

Conjunction Axiom:

$$\frac{\{P\}\ C\ \{Q\},\ \{P'\}\ C\ \{Q'\}}{\{P \wedge P'\ \}\ C\ \{Q \wedge Q'\}}$$

Disjunction Axiom:

$$\frac{\{P\}\ C\ \{Q\},\ \{P'\}\ C\ \{Q'\}}{\{P \vee P'\ \}\ C\ \{Q \vee Q'\}}$$

The axiomatic method is the most abstract of the semantic methods and yet, from the programmer's point of view, the most practical method. It is most abstract in that it does not try to determine the meaning of a program, but only what may be proved about the program. This makes it the most practical since the programmer is concerned with things like, whether the program will terminate and what kind of values will be computed.

Axiomatics semantics are appropiate for program verification and program derivation.

Assertions for program construction

The axiomatic techniques may be applied to the construction of software. Rather than proving the correctness of an existing program, the proof is integrated with the program construction process to insure correctness from the start. As the program and proof are developed together, the assertions themselves may provide suggestions which facilitate program construction.

Loops and recursion are two constructs that require invention on the part of the programmer. The loop correctness principle requires the programmer to come up with both a variant and an invariant. Recursion is a generalization of loops so proofs of correctness for recursive programs also require a loop variant and a loop invariant. In the summation example, a loop variant is readily appearent from

an examination of the post-condition. Simply replace the summation upper limit, which is a constant, with a variable. Initializing the sum and index to zero establishes the invariant. Once the invariant is established, either the index is equal to the upper limit in which case there sum has been computed or the next value must be added to the sum and the index incremented reestablishing the loop invariant. The position of the loop invariants define a loop body and the second occurrence suggests a recursive call. A recursive version of the summation program is given in Figure N.5.

Figure N.5: Recursive version of summation

```
S,I := 0,0
loop: if I < n then S,I := S+A[I+1],I+1; loop
      else skip fi
```

The advantage of using recursion is that the loop variant and invariant may be developed separately. First develop the invariant then the variant.

The summation program is developed from the post-condition by replacing a constant by a variable. The initialization assigns some trivial value to the variable to establish the invariant and each iteration of the loop moves the variable's value closer to the constant.

A program to perform integer division by repeated subtraction can be developed from the post-condition { 0 <= r < d, (a = q× d + r) } by deleting a conjunct. In this case the invariant is { 0 <= r, (a = q× d + r) } and is established by setting the the quotient to zero and the remainder to a.

Another technique is called for in the construction of programs with multiple loops. For example, the post condition of a sorting program might be specified as:

{ forall i.(0 < i < n -> A[i] <= A[i+1]), s = perm(A)}

or the post condition of an array search routine might be specifies as:

{ if exists i.(0 < i <= n and t = A[i]) then location = i else location = 0}

To develop an invariant in these cases requires that the assertion be strengthened by adding additional constraints. The additional constraints make assertions about different parts of the array.

Further Reading

Axiomatic semantics

Gries, David (1981)

The Science of Programming Springer-Verlag.

Hehner, E. C. R. (1984)

The Logic of Programming Prentice-Hall International.

Hehner, E. C. R. (1993)

A Practical Theory of Programming Springer-Verlag.

Exercises

1. (axiomatic) Give axiomatic semantics for the following:

a. Multiple assignment command: $x_0,...,x_n := e_0,...,e_n$

b. The following commands are a nondeterministic if and a nondeterministic loop. The IF command allows for a choice between alternatives while the DO command provides for iteration. In their simplest forms, an IF statement corresponds to an If condition then command and a LOOP statement corresponds to a While condition Do command.

IF *guard* --> *command* FI = if *guard* then *command*

LOOP *guard* --> *command* POOL = while *guard* do *command*

A command proceded by a guard can only be executed if the guard is true. In the general case, the semantics of the IF - FI and LOOP - POOL commands requires that only one command corresponding to a guard that is true be selected for execution. The selection is nondeterministic.. Define the axiomatic semantics for the IF and LOOP commands:

i. if c_0 -> s_0

...
c_n-> s_n
fi

- do c_0 -> s_0

...
c_n-> s_n
od

- A for statement
- A repeat-until statement
- (axiomatic) Use assertions to guide the construction of the following programs.

a. Linear search
b. Integer division implemented by repeated subtraction.
c. Factorial function
d. F_n the *n*-th Fibonacci number where $F_0 = 0$, $F_1 = 1$, and $F_{i+2} = F_{i+1} + F_i$ for i >= 0.

e. Binary search
f. Quick sort

Extreme Programming

Colloquium notes for Kent Beck's *Extreme programming explained: embrace change* Addison Wesley 2000.

The problem isn't change, it's coping with change.

The four basic activities

1. Listening
2. Designing
3. Testing
4. Coding

The four variables

1. Cost
2. Time
3. Quality
4. Scope

The four values

1. Communication
2. Simplicity
3. Feedback
4. Courage

Fundamental principles

- Rapid feedback
- Assume simplicity
- Incremental change
- Embracing change
- Quality work

Secondary principles

- Teach learning
- Small initial investment
- Play to win
- Concrete experiments
- Open, honest communication
- Work with people's instincts, not against them
- Accepted responsibility
- Local adaptation
- Travel light
- Honest measurement

The practices

- The planning game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- 40-hour week
- On-site customer
- Coding standards
  - GNU
  - Java

For more information see:

- ExtremeProgramming.org
- XProgramming.com

Four Software Development Practices That Spell Success

- An early release of the evolving product design to customers.
- Daily incorporation of new software code and rapid feedback on design changes.
- A team with broad-based experience of shipping multiple projects.
- Major investments in the design of the product architecture.

and this: "The most remarkable finding was that getting a low-functionality version of the product into customer's hands at the earliest opportunity improves quality dramatically.

From MIT Sloan Management Review, Winter 2001, Volume 42, Number 2.

Software Design - a trip report

Colloquium presentation notes - WWC 10/4/2000

Abstract: Design is used in three senses - a process, a plan, and an aesthetic. The process of software design has taken inspiration from mathematics, science, and engineering. These disciplines are not the only sources of examples creativity or problem solving. Philosophy, the social sciences and hermeneutics all have had significant an unrecognized influence on software design. This past summer I spent some time exploring parts of the world of software design. This is my report.

Background

Mathematics and Programming

- Geometry: The base angles of isosceles triangles are equal.
- Factorial function
- Programs as functions *output* = Program(*input*)
- Logical derivation of programs {*pre-condition*} Program {*post-condition*}
- Temporal logic for the specification of programs - alternating bit protocol

Starting out

- Extreme Programming - a future colloquium
- Evolved into a question about design - "Is there a general/abstract theory of design that applys across disciplines?"
- and wondering about the relationship between problem solving, design, and creativity.

The design world - what is design

- problem solving - related to design as in design a solution to the problem...
- creativity - related to problem solving where standard solutions are not available and to design when ...
- Definition
    - Design: a process
        - craft
        - art
        - science - programming is constructing a model
        - mathematics - a program is a function

        - engineering - a program is the result of problem solving using the standard engineering design process
        - axiomatic design process
    - Design: a plan
    - Design: an aesthetic
        - like a scientific theory or model

Descriptive design - describes current practice.
Prescriptive design - describes how design should be done. Axiomatic design is one such prescriptive design methodology.

Real world design

- McPhee, Kent (1996) *Design Theory and Software Design* Technical Report TR 96-26. revised 1997.
    - Software design is a wicked problem
    - Software design is just as much a "people problem" as it is a "technical problem"
    - Software design is a social not a solitary activity
    - Software design is a continuous activity until the software is retired
    - Data structures and algorithms are not enough. We need design pattern catalogs.

Preparation for a wicked world: User centered software design - a curricular recommendation

- Social sciences: the wicked sciences
    - Anthropology
    - Economics
    - Psychology
    - Sociology
- Communications: wicked communications
    - Small group communication
    - Interpersonal and nonverbal communication
    - Introduction to general semantics
- Humanities: the wicked disciplines
    - Literature
        - Introduction to lterature
        - Literary analysis
    - Philosophy & religion
        - Approaches to Biblical interpretation

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Last Modified - . Comments and content

The Logic of Self-awareness

Seminar Notes

Date: January 26, 2001 12 p.m.
Location: WWC KRH 345

> *Mathematics is not a careful march down a well-cleared highway, but a journey into a strange wilderness, where the explorers often get lost. Rigour should be a signal to the historian that the maps have been made, and the real explorers have gone elsewhere.* W.S. Anglin, "Mathematics and History", Mathematical Intelligencer, v. 4, no. 4.

Introduction

The foundations of mathematics:

- Functions - A. Church, H. B. Curry, S. Kleene
- Set theory - G. Cantor, Zermelo, Fraenkel
- Logic - the subject of this seminar - D. Hilbert, B. Russell, A. N. Whitehead, K. Goedel, A. Tarski

Themes to watch for

- Finite, Infinite
- Language, Words, Proof
  - Is mathematics a meaningless game?
- Worlds, Meaning, and Truth
  - What is truth?
  - Once something is true it stays true, right?
  - Does life have meaning?

Lecture goals/outline

1. Rules of the game: an overview of logic.
2. The logic of belief and self-awareness.

On to the rules of the game **->**

invited
aabyan@wwc

This material is under development

The Mathematics of Recursion

or
This is not your calculus teacher's continuity.
or
What every computer scientist should know about continuity.

Seminar Notes

Date:
Location:

Background

Definition A *metric space* $<X, d_X>$ is a non-empty set X together with a real-valued function d_X defined on $X \times X$ such that for all x, y, and z in X:

1. $d_X(x, y) >= 0$
2. $d_X(x, y) = 0$ iff $x=y$
3. $d_X(x, y) = d_X(y, x)$
4. $d_X(x, z) <= d_X(x, y) + d_X(y, z)$

Definition A *function f* on a metric space $<X, d_X>$ into a metric space $<Y, d_Y>$ is a rule which associates to each x in X a unique y in Y.

Definition: A sequence $\{x_0, x_1, x_2, ...\}$ of real numbers has a *limit* i.e., $\text{limit}_{i \text{ --> infty}} x_i = L$ iff /\e>0\/n(i>n -> $|x_i - L| < e$

Definition: A sequence $\{x_0, x_1, x_2, ...\}$ from a metric space X *converges* to the point x in X (or has x as a *limit*), if given e > 0, there is an N such that $d_X(x, x_n) <$ e for all n > N.

Definition A real function *f* defined on a non-empty subset X of the real line is said to be *continuous at* x_0 *in* X if for each e>0 there exists d>0 such that x in X and $|x - x_0| < d$ => $|f(x) - f(x_0)| < e$. And *f* is said to be *continuous* if it is continuous at each point of X.

Definition The function *f* is said to be *continuous at x* if, for every e>0, there is a d>0 so that if $d_X(x, y) < d$, then $d_Y(f(x), f(y)) < e$. The function *f* is called *continuous* if it is continuous at each x in X.

Definition A metric space is *complete* if every

Definition A set X is a *fixpoint space* if every continuous function f of X into itself has a fixpoint in the sense that $f(x_0) = x_0$ for some x_0 in X.

Theorem Every continuous function from [-1, 1] into itself has a fixpoint in the interval.

convergent sequence of points in M converges to a point in M.

Definition Let <M, d> be a metric space. Let T : M -> M, we say T is a *contraction* on M, if there exists a in R with 0<= a < 1 such that for every x and y in M, d(Tx, Ty) <= ad(x, y).

Theorem (Picard fixpoint) Let <M, d> be a complete metric space. If T is a contraction on M, then there is one and only one point in M such that Tx = x (T has precisely one fixpoint).

Introduction

Definition: N ::= B means that wherever N occurs, N may be replaced with B (and vice versa).

Mathematical objects must

- exist and be
- well defined.

Recursive definitions are of the form:

N ::= ... N ...

The definition is called recursive because the name of the domain ``recurs" on the right hand side of the definition.

Two types of recursion

Direct recursion	**Indirect recursion**
a ::= ... a ...	a_0 ::= ... a_1 ...
	...
	a_n ::= ... a_0 ...

Recursive definitions occur in

- mathematical functions
  - n! ::= if (n=0) then 1 else n×(n-1)!
- programming language grammars
  - *statement* ::= ... | if *condition* then *statement* else *statement* fi | ...
- programming language semantics
  - `while C do S ::= if C then {S; while C do S}`

- definitions of domains (data types)
  - list ::= empty | item list

Recursive definitions are often justified by appeal to the principle of induction.

Principle by induction if p(0) and p(i) -> p(i+1) then p(n) holds for all n in **N**.

Principle of recursive definition Let A be a set; let a0 be an element of A. Suppose p is a function that assigns, to each function f mapping a nonempty section of the positive integers into A, an element of A. Then there exists a unique function h: N-> A such that h(1) = a0, h(i) = p(h|{1,...,i-1}) for i > 1.

General principle of recursive definition Let J be a well-ordered set; let C be a set. Let *F* be the set of all functions mapping sections of J into C. Given a function p : *F* -> C, there exists a unique function h:J->C such that h(a) = p(h|Sa) for each a in J.

More than one set may satisfy a recursive definition. However, it may be shown that a recursive definition always has a least solution. The least solution is a subset of every other solution.

The least solution of a recursively defined domain is obtained through a sequence of approximations (D_0, D_1,...) to the domain with the domain being the *limit* of the sequence of approximations ($D = \lim_{i \to \infty} D_i$). The limit is the smallest solution to the recursive domain definition.

On to the rules of the game **->**

The trouble with recursion

x ::= x+1 is meaningless but **N** ::= 0 | **N**+1 should be meaningful

When the definition is applied to a *circular definition* h ::= ... h ..., the replacement either

- never terminates creating an infinitely long expression or
- does not yield a right hand expression without h.

N ::= 0 | **N**+1	infinitely long expression
A ::= A	finite expression but undefined

Mathematicians prefer expressions of finite length.

Questions:

- When is it "ok" to write recursive definition?
- What do recursive definitions mean?

Examples

f(n) ::=	if n=0 then 1 else n*f(n-1)	Terminates for n:**N**
g(n) ::=	if n=0 then 1 else g(n+1)/(n+1)	Fails to terminate for n:**Z** n>0
d(n) ::=	if n=0 then 1 elsif n=1 then d(3) else d(n-2)	Fails to terminate for n:**N** even(n)
e(n) ::=	if n=1 then 1 elsif even(n) then e(n/2) else e(3*n+1)	Whether it terminates for n:**N** is unknown

On to the rules of the game **->**

"Good" and "bad" recursive definitions

A ::= A

f(i) ::= w if i = 0
F(i, <f(i-1), ..., f(0)>), if i > 0 and where F is well defined

f(i) ::= 0 if i = 0
f(i-2) otherwise

while (i != 0) i = i-2;

For any given *i*, any occurrence of *h*(*i*) may be replaced through a finite number of substitutions by an expression not involving *h*.

Limitations of a theory of recursive definition

The Halting Problem Determine whether a given loop does or does not terminate.

Theorem The halting problem is undecidable.

On to the rules of the game **->**

Interpreting recursive definitions

Newton's method

Given y = f(x) find r such that 0 = f(r). Solution: choose an initial value x_0 and put it in the point slope formula of a line $y - f(x_0) = f'(x_0)(x - x_0)$. Set y to 0, replace x with x_1 and rearrange giving $x_1 = x_0 - f(x_0)/f'(x_0)$; replace 0 wtih i and 1 with i+1 giving $x_{i+1} = x_i - f(x_i)/f'(x_i)$.

Newton's method

x_0	Choose an initial value
$x_{i+1} = x_i - f(x_i)/f'(x_i)$	Iterate until until the sequence converges

Fixpoints

Definition: Solutions to the equation: x = f(x) are called fixpoints.

Definition	**Number of fixpoints.**
f(x) = x + 1	no fixpoint
f(x) = 2x	one fixpoint, 0
$f(x) = x^2$	two fixpoints, 0, 1
$f(x) = x^3$	three fixpoints, -1, 0, 1
f(x) = x	infinite number of fixpoints, **N**, **Z**, & **R**

Recursive definitions as fixpoint equations

h ::= t(h) - let h be one of the fixpoints of t.

Examples continued

Functionals for f, g, d, and e		Fixpoint equations
F ::= \f \n	if n=0 then 1 else n*f(n-1)	f = F(f)
G ::= \g \n	if n=0 then 1 else g(n+1)/(n+1)	g = G(g)
D ::= \d \n	if n=0 then 1 elsif n=1 then d(3) else d(n-2)	d = D(d)
E ::= \e \n	if n=1 then 1 elsif even(n) then e(n/2) else e(3*n+1)	e = E(e)

Aims for a theory of recursive definitions

We interpret a recursive definition as a fixpoint of the equation h = t(h). We must:

1. Define sufficient conditions under which functionals are guaranteed to have fixpoints.
2. Define which fixpoints are appropriate solutions to the problem of interest.
3. Show how to compute fixpoints when they exist.

On to the rules of the game **->**

Iterative methods - how to compute a fixpoint

Fixpoints in real analysis

Definition: $\text{limit}_{i \text{ --> infty}} f(x_i) = L$ iff /\e\/n(i>n -> $|f(x_i) - L| < e$

To find *x* such that *x* = t(*x*) where t is a continuous function

1. Pick an initial value x_0 ::= initial value
2. Iterate x_i ::= $t(x_{i-1})$

Theorem: If the sequence constructed in the manner above has limit x then x is a fixpoint of t.

Example: t ::= \x.(1 + 1/x) converges towards $(1+5^{1/2})/2$

Functions as limits of sequences

Repeated substitution

- From **N** ::= 0 | **N**+1 we derive by repeated substitution
- **N** ::= 0 | 1 | ...

Figure n: Limit construction for D ::= e(D)

D_{i+1} ::= e[D:D_i] for i=0,...	Rewrite the definition in iterative form substituting D_i for D on the right hand side
D_0 = null	Pick an initial value and construct several terms

in the sequence if necessary. Guess solution L and prove by induction:

L = $\lim_{i \text{ --> infty}} D_i$

Bottom-up and top-down recursive computation

- Recursive definition is a top-down view
- Fixpoint computation is a bottom-up view

On to the rules of the game **->**

Solving recursive equations

On to the rules of the game **->**

Stable functions

On to the rules of the game **->**

Partial functions

On to the rules of the game **->**

A more general theory

Definition A binary relation R on a set X is an *order relation* iff it is reflexive (Rxx), transitive (Rxy, Ryz -> Rxz), and antisymmetric (Rxy, Ryx -> x=y).

Two distinct elements x and y are said to be *comparable* for relation R iff either Rxy or Ryx . If any two members of X are comparable, then the relation is a *total order* relation. If an order relation is not a total order relation then it is a *partial order* relation.

Note: what we here call an order relation, is usually called a partial order relation. It seems more appropriate to define both total order relations and partial order relations as special cases as is done here.

Definition (Cartesian Product Order) Let X and Y be sets with respective order relations R_X and R_Y. Then X×Y is ordered under the relation $R_{X\times Y}$ called the **cartesian product order** defined by $R_{X\times Y}(x, x')(y, y')$ iff $R_X xx'$ and $R_Y yy'$.

Definition A function *f* is a set of ordered pairs (x, y) where if (x, y) and (x, z) are in *f*, then y = z.

Definition (Function Order) Let X and Y be arbitrary sets. Then X->Y is ordered under the relation $R_{X->Y}$, called the function order on X->Y, defined by $R_{X->Y}hk$ iff h is a subset of k.

Theorem For any two functions h and k in X -> Y,

1. If $R_{X->Y}hk$ then **dom** h is a subset of **dom** k.
2. $R_{X->Y}hk$ iff k \ **dom** h = h.

Definition (Minimum Member) Let X be a set with an order relation R. A *minimum* member of X is an object m in X such that /\x:X Rmx.

Theorem An ordered set has at most one minimum member.

A *well-founded* set X is a set with an order relation R and a minimum member.

Definition (Upper Bound) Let X be a set with an order relation R and a subset A of X. An *upper bound* of A in X is an object l in X such that /\a:A Ral.

Definition (Least Upper Bound) Let X be a set with an order relation R and a subset A of X. A *least upper bound* of A in X is an object l in X such that

- l is an upper bound of A in X
- For any upper bound l' of A in X, Rll'

We write **lub** A = l.

Theorem A subset of an ordered set has at most one least upper bound.

Definition (Chain) In a set X with an order relation R, a chain is an infinite sequence $\{x_0, x_1, x_2, ...\}$ such that for all i in **N**, Rx_ix_{i+1}.

Definition (Closed Order) Let X be a set with an order relation R and a subset A of X. A is *lub-closed* (or just *closed*) iff every chain with all its values in A has a least upper bound in A.

Note: In the literature, a closed set with a minimum member is called a complete partial order or **cpo**.

Theorem Let X and Y be closed sets. Then X×Y is closed under the induced order.

Theorem Let X and Y be sets. Then X->Y is closed under the function order.

Theorem (Intersection) Let X be an ordered set; let A and B be two closed subsets of X. Then AnB is closed. Note that X itself does not need to be closed.

Theorem (Union) Let X be an ordered set; let A and B be two closed subsets of X. Then AuB is closed.

Definition (Monotonicity) Let F and G be ordered sets. A total function t : F -> G is monotonic if and only if $/\backslash h_1, h_2 : F\ h_1 <= h_2 => t(h_1) <= t(h_2)$

Definition (Continuous Function) Let F and G be closed sets. A total function t : F -> G is continuous if and only if

1. t is monotonic
2. For any chain h, t (lub h) = lub t(h)

Theorem (Least Fixpoint) Let *F* be a **cpo** and t: *F*-> *F* a continuous function, then t has a least fixpoint.

Lemma let h and k be two chains in a closed set, such that $/\backslash i:\mathbf{N}\ Rh_ik_i$ then R lub $<h_i>$ lub $<k_i>$.

h=ΔD

On to the rules of the game **->**

References

Meyer, Bertrand *Introduction to the Theory of Programming Languages* PHI 1990.

Last Modified - . Comments and content invited aabyan@wwc.edu

---

Introduction to Logic

Logic - the rules of reasoning and argumentation
Symbolic logic - reasoning reduced to symbolic manipulation.

Part 1: The Basics

Definition A *statement* is a sentence that can be judged as either *true* or *false*.

Examples

> There are seven days in a week.
> 2 + 5 = 4
> Maria is an interesting person.
> x + 7 = 5
> Jose is 19 years old.
> She loves him.

The study of logic involves:

> *Syntax*: the structure of statements
> *Inference*: a method of reasoning (proof)
> *Semantics*: the meaning or interpretation of statements.

Syntax

Propositions

p, q, ...	A set of *propositional* letters (the sentences).
A, B, C, ...	A set of propositional variables.

Logical Formulas

Formula	Read as	Operator
A /\ B	A and B	conjunction
A \/ B	A or B (inclusive or)	disjunction
¬A ~A	not A	negation

Semantics and the world of meaning

Formula	Meaning
|= A	A is true
|= A /\ B	A /\ B is true if |= A and |= B
|= A\/B	A \/ B is true if either |=A or |=B (inclusive or)
|= ¬A	¬A is true if not |= A (i.e. A is not true)

Truth tables

A	B	A/\B
false	false	false
false	true	false
true	false	false
true	true	true

A	B	A\/B
false	false	false
false	true	true
true	false	true
true	true	true

A	B	A xor B
false	false	false
false	true	true
true	false	true
true	true	false

A	¬A
false	true
true	false

Part 2: More formulas

Syntax

Logical Formulas

Formula	Read as	Operator
A -> B	if A then B	conditional A is the hypothesis B is the conclusion
A <-> B A iff B	A if and only if B	biconditional

Semantics

If the moon is made of blue cheese, I'm a monkey's uncle.
Don't move or I'll shoot!
If you move, I'll shoot!

A -> B	conditional
B -> A	converse
¬A -> ¬B	inverse
¬B -> ¬A	contrapositive

Tautology

Definition A *tautology* is a statement that is always true.

A\/¬A
A/\(A->B)->B

DeMorgan's and other laws

¬(A/\B)	=	¬A\/¬B	DeMorgan
¬(A\/B)	=	¬A/\¬B	DeMorgan
A->B	=	¬A\/B ¬B->¬A	Definition
A<->B	=	A->B /\ B->A	Definition
A/\(B\/C)	=	A/\B \/ A/\C	Distributive
A\/(B/\C)	=	A\/B /\ A\/C	Distributive

Part 3: Valid arguments

If the snow is at least 45 inches deep, then we can go skiing.
The snow is at least 45 inches deep.

Therefore, we can go skiing.

$$\frac{A->B,\ A}{B}$$ Modus Ponens
Law of detachment

If the snow is at least 45 inches deep, then we can go skiing.
We can't go skiing.

The snow is not at least 45 inches deep.

$$\frac{A->B,\ \neg B}{\neg A}$$ Modus Tollens

Aguments

Reasoning from the converse
Reasoning from the inverse
Reasoning from the contrapositive (Proof by contradition, reducto ad absurdum)

The State of CS 2001

Quality

CS Department MFAT Scores (Percentile)

94-95	95-96	96-97	97-98	98-99	99-00	00-01
84	99	99	98	95	99	94

US News & World Report Ranking: WWC in top tier of western universities.

The CS Lab

New hardware

- Two AMD 1.4 GHertz Athalons with 19 inch monitors
- 24 port 10/100 MHertz Switch
- Sun Ultra 5 - conscience donation

Lab support

- James Davis
- Nolan Cafferky
- Michael Goff

Access

- Email aabyan@wwc.edu with Name and student ID; access code sent by Ralph Stirling to Groupwise account
- CS account see James or Nolan

Program information

- **The Schedule**
- **Advising**
  - **Sample schedules**
  - **Major electives**

    - **Free electives**
        - **AI**
        - **Business**
        - **HCI - user interface design theory**
        - **SE- communication skills & requirements engineering**
    - **Emphasis**
    - **INFO 250**
- CPTR 215 Assembly Language Programming

Future

Beowulf - rackmount units
Wireless project - James Davis & Jim Klein
Group projects - Source forge
Communication

Software Engineering

<table>
<tr><th colspan="5">Software Engineering Process</th></tr>
<tr><td rowspan="4">Software Maintenance</td><td rowspan="4">Software Engineering Management</td><td></td><td>Requirements</td><td rowspan="4">Configuration Management
Tools and Methods
Quality</td></tr>
<tr><td rowspan="3">Acquisition</td><td>Design</td></tr>
<tr><td>Construction</td></tr>
<tr><td>Testing</td></tr>
<tr><td colspan="2"></td><td>Installation Operation/Maintenance</td></tr>
<tr><td colspan="5">Related Disciplines
Cognitive sciences and human factors
Computer engineering
Computer science
Management and management science
Mathematics
Project management
Systems engineering</td></tr>
</table>

Comdex Report

Development environments

- Borland www.borland.com
  - Kylix
  - JBuilder 5
- Giesecke & Devrient: SM@RT CAFE - www.smartcafe.gieseckedevrient.com
- Handspring - www.handspring.com
- Nokia - Americas.Forum.Nokia.com
- Palm OS Developer Resource CD - www.palm.com
- PenbexOS www.penbex.com
- Red Sonic - Red Builder - www.redsonic.com
- Trolltech - Qt 3 - www.trolltech.com

Smart Card

Smart Cards (The Java Card) SM@RT CAFE - www.smartcafe.gieseckedevrient.com

Mobile Computing: Tablet computers and PDAs

- Handspring - www.handspring.com
- Palm OS Developer Resource CD - www.palm.com
- Linux PDAs
  - G. Mate Yopy - www.gmate.com
  - Sharp
  - Milletech www.milletech.com Wireless PDAs
- xybernaut www.xybernaut.com

Rackmount

- Advanced Industrial Computer www.aicipc.com
- Portwell www.portwell.com
- SleekLine 1260 www.sleekline.com
- Mameden www.mameden.com

- Huntec www.huntec.com

Blades

- Egenera www.egenera.com
- OmniCluster PCI SBC www.omnicluster.com
- Tatung 16 server blades in 2U chassis www.tsti.com

Motherboards

- ABIT www.abit-usa.com
- Arbor Solution www.arbor.com.tw
- DFI www.dfi.com
- MSI www.msicomputer.com
- PC Wave www.pcwave.com
- Portwell www.portwell.com

Embedded systems and SBC

- Aaeon www.aaeon.com
- Advantech www.advantech.com
- Axiom Technology www.axiomtek.com
- ICCOP Technology www.icop.com.tw
- Megatel www.megatel.ca
- Maxan Systems www.maxan.com
- Technoland www.technoland.com
- X-tra Web: x-node, x-gate www.x-traweb.com

LCD Displays

- Aaeon www.aaeon.com
- AEI Components www.aeicomp.com
- CTX www.ctxintl.com
- LG Electronics www.flatron.com

I/O Devices

- Olympus Eye-Trek www.olympus-eye-trek.com

- Miracle mouse www.miracle-mouse.com
- Vertical mouse www.Vertical-Mouse.com
- Rocket Drive www.cenatek.com
- Koolance www.koolance.com
- Robots: www.parallaxinc.com (BASIC stamps)

Prolog and AI

Introduction

Wirth: Program = Data structures + Algorithms (1976)

Kowalski: Algorithm = Logic + Control (1979)

Prolog program: a set of specifications in the first-order predicate calculus -- a *database* of facts and rules.
Prolog interpreter: answers queries (questions) about the database using pattern-directed search to see if the query is a logical consequence of the database.

Prolog is usually implemented in an interpreter providing an interactive environment in which the user enters queries in response to the prompt: `?-` .

Comment: Program execution, reguardless of language, and theorem proving are simply graph traversals.

See HOWTO for SWI Prolog on CS

Syntax for Predicate Calculus Programming

Logical Formulae in English, the Predicate Calculus, and in Prolog:

English	Predicate Calculus	Prolog
A and B	A /\ B	A, B
A or B	A \/ B	A; B
A if B	A <-- B	A :- B.
not A	~ A	not A

Textbook error:

A if and only if B	A if B and A only if B

`A if B`	`A <-- B`
`A only if B`	`A --> B`

Horn Clause Logic

- Horn Clause Logic: `H :- A, B, ... .`
- `Refutation, unification (pattern matching)`
- Variables (begin with an uppercase letter) are universally quantified in the database but existentially quantified in queries.
- Scope of a variable is the fact or rule in which it occurs.

Prolog Program and Logical Equivalent Specification

	Prolog Program	**Logical form**

`Propositional logic`	`a.`
	`b.`
	`c :- a, b.`
	`?- c.`
		`a`
		`b`
		`c <-- a /\ b`
		`~ c`

`Predicate logic (first-order)`	`a(X,y).`
	`b.`
	`c(A,B) :- a(A,B),`
	`b.`
	`?- c(M,N).`
		`all X.a(X,y)`
		`b`
		`all A,B.(c(A,B) <--`
		`a(A,B)/\b)`
		`exists M,N.~c(M,N)`

Prolog implementation of refutation

- Closed world assumption; Negation as failure
- left-to-right
- depth-first search
- selecting clauses from the database in the order of appearance
- *backtracking* on failure
- without the *occurs check* (X unifies with p(X) producing an infinite term: X=p(p(p(p...).

English to Logic
Simple Example

Some builtin predicates

- `assert(`*`Clause`*`)`, `asserta(`*`Clause`*`)`, `assertz(`*`Clause`*`)`
- `retract(`*`Clause`*`)`
- `consult(`*`File`*`)`
- `see(`*`File`*`)`, `seen`
- `tell(`*`File`*`)`, `told`

- `read(`*`Term`*`)`
- `write(`*`Term`*`),nl`
- `listing(`*`PredicateName/Arity`*`)`
- `trace,`
- `spy, nospy`

Lists

- Empty List: []
- [Head | Tail]
- [X,Y | Rest]
- Examples

Recursive search & Cut

Prolog: Towards Nonprocedural Computing

Programs in pure Prolog have no notion of control or direction of computation.

Examples

- append
- sentences (parser & generator; simple grammar)
- sentences (parser & generator; grammar with noun-verb agreement)

Procedural view of Prolog

```
a :- b, c, d. % to accomplish a, first do b, then c then d.
```

Abstract Data Types (ADTs) in Prolog

- ADT Stack
- ADT Queue
- ADT Priority Queue
- ADT Set

A Production System Example in Prolog

Farmer, wolf, goat, cabbage problem:

A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross. There is a boat at the river's edge, but, of course, only the farmer can row. The boat also can carry only two things (including the rower) at a time. Devise a sequence of crossings of the river so that all four arrive safely on the other side of the river. Remembering that if left alone the goat will eat the cabbage and the wolf the goat.

Designing Alternative Search Strategies

- Depth-First Search
- Breadth-First Search
- Best-First Search

A Prolog Planner

Blocks world.

Prolog: Meta-Predicates, Types, & Unification

Meta-predicates: Meta-predicates are used to match, query, and manipulate other predicates in the problem domain. They are needed for five reasons.

1. To determine the type of an expression.
2. To add type restrictions to logic programming.
3. To build, take apart, and evaluate Prolog structures.
4. To compare values of expressions.
5. To convert predicate passed as data to executable code.

Meta-predicates

`assert(C)` - adds clause `C` to the current set of clauses.
`var(X)` - succeeds only when `X` is an unbound variable.
`nonvar(X)` - succeeds only when `X` is bound to a nonvariable term.
`Term =..LIST` - creates a list from a predicate term.
`functor(Term, Functor, Arity)` -
`clause( Head, Body )` - unifies `Body` with the body of a clause whose head unifies with `Head`.
any_predicate`(..., X, ...) :- X` - executes predicate X, the argument of any predicate.
`call(Clause)` - succeeds with the execution of `Clause`.

Types and Type Checking

- Prolog is an untyped language
- Unification matches patterns

Unification, Pattern Matching and Evaluation

- *Var* = *expression* (unification/pattern matching)
- *Var* `is` *expression* (evaluation and assignment)
- Arithmetic operators: `+ - * / mod`
- `successor(X, Y) :- Y is X+1.`

Difference lists

- [a, b] = [a, b | []] - []
- [a, b] = [a, b, c] - [c]
- [a, b] = [a, b | Y] - Y
- X - Z = X - Y + Y - Z
- Join two lists in constant time by unification: `concatenate(X-Y, Y-Z, X-Z)`
- Empty difference list: `L-L`

Meta-Interpreters in Prolog

- Meta interpreter
- Meta interpreter with user interaction
- Meta interpreter with user interaction and response to why queries
- Meta interpreter with user interaction and proof tree construction
- Shell for a Rule-Based Expert System
- Full shell for rule-based expert system
- Cars knowledge base
- Semantic nets: isa(Type,Parent). hasprop(Object, Property, Value)
- Frames and schemata
- Frames and Semantic net example

Frame

name	Name
isa	InheritanceLinks
properties	PropertyList
exceptions	ExceptionsAndDefaultList

Additional logic programming examples

English to Logic

Temporal Logic

Classical Logic

The goal is to create a set of literal formulas.

Rule		Replacement Rule	Description
Alpha (extends branch)	conjunction	$\frac{A \wedge B}{A, B}$	Replace A /\ B with the subformulas A and B.
Beta (creates branch)	disjunction	$\frac{A \vee B}{A \mid B}$	Replace A \/ B and branch with subformulas on different branches.
Gamma	universal	$\frac{\text{all x. Px}}{\text{Pc, all x. Px}}$	Add the subformula Pc to the branch. (Universal formulas hold for all constants)
Delta	existential	$\frac{\text{exists x. Px}}{\text{Pc}}$	Replace exists x. Px with the subformula Pc (c is new to the branch).

Proof: Axioms + ~ Theorem - -> contradictions on all branches.
Satisfiable: Formulas - -> at least one branch with no contradictions is a model.

Temporal logic

The nature of time

- Linear time: formulas apply to all time sequences
- Branching time: quatifiers for all branches and some branch

Adds operators for all futures and some future.

Rule		**Replacement Rule**	Implications for proof tree construction.
Box (future time)	always	@ A --- A, _@ A	Replace @ A with subformula A and put @ A in next state.
Diamond (future time)	eventually	! A --- A | ~A, _! A	Replace ! A and branch with subformulas on different branches and put !A in state following branch with ~A.

Classification of formulas

Rule	**Formula**	**Subformulas**
Alpha (extends branch)	~ ~A A /\ B ~(A\/B) ~(A -> B)	A A, B ~A, ~B A, ~B
Beta (creates branch)	A \/ B ~(A /\ B) A -> B A <-> B ~(A <-> B)	A | B ~A | ~B ~ A | B A, B | ~A, ~B A, ~B | ~A, B
Box (future time)	@ A	A, _@ A
Diamond (future time)	! A	A | ~A, _! A
Gamma	all x Px ~ exists x Px	Pc, all x Px ~ Pc, all x. ~Px
Delta	exists x. Px ~ all x. Px	Pc ~ Pc

Model Construction

Formulas - -> model
Axioms + ~ Theorem - -> no model.

C (Compound formulas, Literal formulas, Future-time formulas)

Replace	with	**This**

C **(** [*f* | CF], Lit, NT **)**	->	***C*** **(** CF, [*f* |Lit], NT **)**
C **(** [~*f* | CF], Lit, NT **)**		***C*** **(** CF, [~*f* |Lit], NT **)**
C **(** [A /\ B | CF], Lit, NT **)**		***C*** **(** [A, B | CF], Lit, NT **)**
C **(** [A\/ B | CF], Lit, NT **)**		***C*** **(** [A | CF], Lit, NT **)**, ***C*** **(** [B | CF], Lit, NT **)**
C **(** [!A | CF], Lit, NT **)**		***C*** **(** [A | CF], [sat(A) | Lit], NT **)**, ***C*** **(** CF, [ev(A) | Lit], [!A | NT] **)**
C **(** [@A | CF], Lit, NT **)**		***C*** **(** [A | CF], Lit, [@A |NT] **)**

state graph: ***C*** **(** CF, [ ], [ ] **)** =>***C*** **(** [ ], Lit, NT **)** -> ***C*** **(** NT, [ ], [ ] **)**

NT simplifications

- Both @!A and !A in NT -> remove !A from NT
- !A in NT and sat(A) in Lit -> remove !A from NT

Lit simplifications:

- ev(A) and sat(A) in Lit -> remove ev(A) from Lit

State = ***C*** **(** [ ], Lit, NT **)**

Initial State = ***C*** **(** [ ], [true], NT **)**

Final State = ***C*** **(** [ ], Lit, [ ] **)**

Contradictory State = ***C*** **(** [ ], Lit, NT **)** where *f* and ~*f* are Lit.

Unreachable State: there is no path from initial state.

Unsatisfiable State = ***C*** **(** [ ], Lit, NT **)**

- Ev(A) is in Lit, sat(A) is not in Lit and
  - state is not in a connected component and there is no path to a state containing sat(A) in Lit, or
  - state is in a connected component, but there is no state in the connected component that contains sat(A) in its Lit list.

State graph construction

- Begin with a state ***C*** **(** [ ], [true], NT **)**
- Construct new states begining with ***C*** **(** NT, [ ], [ ] **)** applying the configuration rules.
- Repeat until no new states are created.
- Prune graph by removing
  - all contradictory states,
  - all unsatisfiable states, and
  - all unreachable states.

A Temporal Logic Example

English language version

> You can fool all of the people some of the time,
> some people all the time,
> but not all of the people all of the time.

Classical First-order Predicate Logic Version

> A p. E t. [time(t) /\ person(p) -> whenFooled(p,t)] /\
> E p. A t. [time(t) /\ person(p) -> whenFooled(p,t)] /\
> A t. E p. [time(t) /\ person(p) -> ~ whenFooled(p,t)]

Many-sorted First-order Predicate Logic Version

> A p:person. E t:time. whenFooled(p,t) /\
> E p:person. A t:time. whenFooled(p,t) /\
> A t:time. E p:person. ~ whenFooled(p,t)

Propositional Temporal Logic Version (linear time)

> <> all people fooled /\
> @ some people fooled /\
> ~ @ all people fooled

First-order Temporal Logic Version (linear time)

> <> A p. (person(p) -> fooled(p)) /\
> @ E p. (person(p) -> fooled(p)) /\
> ~ @ A p. (person(p) -> fooled(p))

1999.3.20 Anthony Aaby

```
likes(Everyone,susie).


likes(george,kate).
likes(george,susie).
likes(george,wine).
likes(susie,wine).
likes(kate,gin).
likes(kate,susie).

friends(X,Y) :- likes(X,Z), likes(Y,Z).
```

```
% Some List examples

element(X,[X|_]).
element(X,[_|L]):- element(X,L).

naiveReverseList([],[]).
naiveReverseList([H|T], RL):- naiveReverseList(T,RT),
                              append(RT,[H],RL).

reverseList(L,RL):- reverseList(L,[],RL).
reverseList([],RL,RL).
reverseList([H|T],R,RL):- reverseList(T,[H|R],RL).

writeList([]).
writeList([Head|Tail]):- write(Head), nl, writeList(Tail).

reverseWriteList([]).
reverseWriteList([Head|Tail]):-reverseWriteList(Tail), write(Head),nl.

% Difference Lists

concatenate(X-Y, Y-Z, X-Z).
```

```
% 3x3 knight's tour
:- dynamic been/1.

path(Z,Z).
path(A,C):- move(A,B), not(been(B)), assert(been(B)), path(B,C).

% initial call is path2(A,B,[A])
path2(Z,Z,Been).
path2(A,C,Been):- move(A,B), not member(B,Been), path2(B,C,[B|Been]).

% initial call is path3(A,B,[A]) AT MOST ONE SOLUTION
path3(Z,Z,Been).
path3(A,C,Been):- move(A,B), not member(B,Been), path3(B,C,[B|Been]),!.


move(1,6).
move(1,8).
move(2,7).
move(2,9).
move(3,4).
move(3,8).
move(4,3).
move(4,9).
move(6,7).
move(6,1).
move(7,6).
move(7,2).
move(8,3).
move(8,1).
move(9,4).
move(9,2).
```

```
ppend([], List, List).
ppend([X|L1], L2, [X|L1L2]) :- ppend(L1, L2, L1L2).
```

```
utterance(ListOfWords) :- sentence(ListOfWords, [ ]).

sentence(Start,End) :- nounPhrase(Start, Rest), verbPhrase(Rest, End).

nounPhrase([Noun | End], End) :- noun(Noun).
nounPhrase([Article, Noun | End], End) :- article(Article), noun(Noun).

verbPhrase([Verb | End], End) :- verb(Verb).
verbPhrase([Verb | Rest], End) :- verb(Verb), nounPhrase(Rest, End).

article(a).
article(the).

noun(man).
noun(dog).

verb(likes).
verb(bites).
```

```
utterance(ListOfWords) :- sentence(ListOfWords, [ ]).

sentence(Start,End) :- nounPhrase(Start, Rest, Number),
                       verbPhrase(Rest, End, Number).

nounPhrase([Noun | End], End, Number) :- noun(Noun, Number).
nounPhrase([Article, Noun | End], End, Number) :- article(Article, Number),
                                                  noun(Noun, Number).

verbPhrase([Verb | End], End, Number) :- verb(Verb, Number).
verbPhrase([Verb | Rest], End, Number) :- verb(Verb, Number),
                                  nounPhrase(Rest, End, Number).

article(a, singular).
article(these, plural).
article(the, singular).
article(the, plural).

noun(man, singular).
noun(men, plural).
noun(dog, singular).
noun(dogs, plural).

verb(likes,singular).
verb(like, plural).
verb(bites, singular).
verb(bite, plural).
```

```
%%%%%%%%%%%%%%%%%%%% stack operations %%%%%%%%%%%%%%%%%%%%%%%%%%%%

        % These predicates give a simple, list based implementation of stacks

        % empty stack generates/tests an empty stack

% BUILT IN TO SWI PROLOG
%member(X,[X|T]).
%member(X,[Y|T]):-member(X,T).

empty_stack([]).

        % member_stack tests if an element is a member of a stack

member_stack(E, S) :- member(E, S).

        % stack performs the push, pop and peek operations
        % to push an element onto the stack
               % ?- stack(a, [b,c,d], S).
        %    S = [a,b,c,d]
        % To pop an element from the stack
        % ?- stack(Top, Rest, [a,b,c]).
        %    Top = a, Rest = [b,c]
        % To peek at the top element on the stack
        % ?- stack(Top, _, [a,b,c]).
        %    Top = a

stack(E, S, [E|S]).

%%%%%%%%%%%%%%%%%%%% queue operations %%%%%%%%%%%%%%%%%%%%%%%%%%%%

        % These predicates give a simple, list based implementation of
        % FIFO queues

        % empty queue generates/tests an empty queue


empty_queue([]).

        % member_queue tests if an element is a member of a queue

member_queue(E, S) :- member(E, S).

        % add_to_queue adds a new element to the back of the queue

add_to_queue(E, [], [E]).
add_to_queue(E, [H|T], [H|Tnew]) :- add_to_queue(E, T, Tnew).

        % remove_from_queue removes the next element from the queue
        % Note that it can also be used to examine that element
        % without removing it

remove_from_queue(E, [E|T], T).

append_queue(First, Second, Concatenation) :-
        append(First, Second, Concatenation).

%%%%%%%%%%%%%%%%%%%% set operations %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        % These predicates give a simple,
        % list based implementation of sets

        % empty_set tests/generates an empty set.

empty_set([]).

member_set(E, S) :- member(E, S).

        % add_to_set adds a new member to a set, allowing each element
```

```
        % to appear only once

add_to_set(X, S, S) :- member(X, S), !.
add_to_set(X, S, [X|S]).

remove_from_set(E, [], []).
remove_from_set(E, [E|T], T) :- !.
remove_from_set(E, [H|T], [H|T_new]) :-
        remove_from_set(E, T, T_new), !.

% BUILT IN TO SWI PROLOG

/*
union([], S, S).
union([H|T], S, S_new) :-
        union(T, S, S2),
        add_to_set(H, S2, S_new).

intersection([], _, []).
intersection([H|T], S, [H|S_new]) :-
        member_set(H, S),
        intersection(T, S, S_new),!.
intersection([_|T], S, S_new) :-
        intersection(T, S, S_new),!.
*/

set_diff([], _, []).
set_diff([H|T], S, T_new) :-
        member_set(H, S),
        set_diff(T, S, T_new),!.
set_diff([H|T], S, [H|T_new]) :-
        set_diff(T, S, T_new), !.

subset([], _).
subset([H|T], S) :-
        member_set(H, S),
        subset(T, S).

equal_set(S1, S2) :-
        subset(S1, S2), subset(S2, S1).

%%%%%%%%%%%%%%%%%%%%%% priority queue operations %%%%%%%%%%%%%%%%%%

        % These predicates provide a simple list based implementation
        % of a priority queue.

        % They assume a definition of precedes for the objects being handled

empty_sort_queue([]).

member_sort_queue(E, S) :- member(E, S).

insert_sort_queue(State, [], [State]).
insert_sort_queue(State, [H | T], [State, H | T]) :-
        precedes(State, H).
insert_sort_queue(State, [H|T], [H | T_new]) :-
        insert_sort_queue(State, T, T_new).

remove_sort_queue(First, [First|Rest], Rest).
```

```
/*
 * This is the code for the Farmer, Wolf, Goat and Cabbage Problem
 * using the ADT Stack.
 *
 * Run this code by giving PROLOG a "go" goal.
 * For example, to find a path from the west bank to the east bank,
 * give PROLOG the query:
 *
 *   go(state(w,w,w,w), state(e,e,e,e)).
 */

:- [adts]. /* consults (reconsults) file containing the
              various ADTs (Stack, Queue, etc.) */

go(Start,Goal) :-
        empty_stack(Empty_been_stack),
        stack(Start,Empty_been_stack,Been_stack),
        path(Start,Goal,Been_stack).

/*
 * Path predicates
 */

path(Goal,Goal,Been_stack) :-
        write('Solution Path Is:' ), nl,
        reverse_print_stack(Been_stack).

path(State,Goal,Been_stack) :-
        move(State,Next_state),
        not(member_stack(Next_state,Been_stack)),
        stack(Next_state,Been_stack,New_been_stack),
        path(Next_state,Goal,New_been_stack),!.

/*
 * Move predicates
 */

move(state(X,X,G,C), state(Y,Y,G,C))
              :- opp(X,Y), not(unsafe(state(Y,Y,G,C))),
                 writelist(['try farmer takes wolf',Y,Y,G,C]).

move(state(X,W,X,C), state(Y,W,Y,C))
              :- opp(X,Y), not(unsafe(state(Y,W,Y,C))),
                 writelist(['try farmer takes goat',Y,W,Y,C]).

move(state(X,W,G,X), state(Y,W,G,Y))
              :- opp(X,Y), not(unsafe(state(Y,W,G,Y))),
                 writelist(['try farmer takes cabbage',Y,W,G,Y]).

move(state(X,W,G,C), state(Y,W,G,C))
              :- opp(X,Y), not(unsafe(state(Y,W,G,C))),
                 writelist(['try farmer takes self',Y,W,G,C]).

move(state(F,W,G,C), state(F,W,G,C))
              :- writelist(['      BACKTRACK from:',F,W,G,C]), fail.

/*
 * Unsafe predicates
 */

unsafe(state(X,Y,Y,C)) :- opp(X,Y).
unsafe(state(X,W,Y,Y)) :- opp(X,Y).

/*
 * Definitions of writelist, and opp.
 */
```

```
writelist([]) :- nl.
writelist([H|T]):- print(H), tab(1),  /* "tab(n)" skips n spaces. */
                   writelist(T).

opp(e,w).
opp(w,e).

reverse_print_stack(S) :-
        empty_stack(S).
reverse_print_stack(S) :-
        stack(E, Rest, S),
        reverse_print_stack(Rest),
        write(E), nl.
```

```
        %%%%%Basic depth first path algorithm in PROLOG %%%%%%%

go(Start, Goal) :-
        empty_stack(Empty_been_list),
        stack(Start, Empty_been_list, Been_list),
        path(Start, Goal, Been_list).

        % path implements a depth first search in PROLOG

        % Current state = goal, print out been list
path(Goal, Goal, Been_list) :-
        reverse_print_stack(Been_list).

path(State, Goal, Been_list) :-
        mov(State, Next),
        % not(unsafe(Next)),
        not(member_stack(Next, Been_list)),
        stack(Next, Been_list, New_been_list),
        path(Next, Goal, New_been_list), !.

reverse_print_stack(S) :-
        empty_stack(S).
reverse_print_stack(S) :-
        stack(E, Rest, S),
        reverse_print_stack(Rest),
        write(E), nl.
```

```
%%%%%%% Breadth first search algorithm%%%%%%%%

state_record(State, Parent, [State, Parent]).

go(Start, Goal) :-
        empty_queue(Empty_open),
        state_record(Start, nil, State),
        add_to_queue(State, Empty_open, Open),
        empty_set(Closed),
        path(Open, Closed, Goal).

path(Open,_,_) :- empty_queue(Open),
                  write('graph searched, no solution found').

path(Open, Closed, Goal) :-
        remove_from_queue(Next_record, Open, _),
        state_record(State, _, Next_record),
        State = Goal,
        write('Solution path is: '), nl,
        printsolution(Next_record, Closed).

path(Open, Closed, Goal) :-
        remove_from_queue(Next_record, Open, Rest_of_open),
        (bagof(Child, moves(Next_record, Open, Closed, Child), Children);Children = []),
        add_list_to_queue(Children, Rest_of_open, New_open),
        add_to_set(Next_record, Closed, New_closed),
        path(New_open, New_closed, Goal),!.

moves(State_record, Open, Closed, Child_record) :-
        state_record(State, _, State_record),
        mov(State, Next),
        % not (unsafe(Next)),
        state_record(Next, _, Test),
        not(member_queue(Test, Open)),
        not(member_set(Test, Closed)),
        state_record(Next, State, Child_record).

printsolution(State_record, _):-
        state_record(State,nil, State_record),
        write(State), nl.
printsolution(State_record, Closed) :-
        state_record(State, Parent, State_record),
        state_record(Parent, Grand_parent, Parent_record),
        member(Parent_record, Closed),
        printsolution(Parent_record, Closed),
        write(State), nl.

add_list_to_queue([], Queue, Queue).
add_list_to_queue([H|T], Queue, New_queue) :-
        add_to_queue(H, Queue, Temp_queue),
        add_list_to_queue(T, Temp_queue, New_queue).
```

```
%%%%%%% Best first search algorithm%%%%%%%%%


        %%%%% operations for state records %%%%%%%
        %
        % These predicates define state records as an adt
        % A state is just a [State, Parent, G_value, H_value, F_value] tuple.
        % Note that this predicate is both a generator and
        % a destructor of records, depending on what is bound
        % precedes is required by the priority queue algorithms

state_record(State, Parent, G, H, F, [State, Parent, G, H, F]).
precedes([_,_,_,_,F1], [_,_,_,_,F2]) :- F1 =< F2.

        % go initializes Open and CLosed and calls path
go(Start, Goal) :-
        empty_set(Closed),
        empty_sort_queue(Empty_open),
        heuristic(Start, Goal, H),
        state_record(Start, nil, 0, H, H, First_record),
        insert_sort_queue(First_record, Empty_open, Open),
        path(Open,Closed, Goal).

        % Path performs a best first search,
        % maintaining Open as a priority queue, and Closed as
        % a set.

        % Open is empty; no solution found
path(Open,_,_) :-
        empty_sort_queue(Open),
        write("graph searched, no solution found").

        % The next record is a goal
        % Print out the list of visited states
path(Open, Closed, Goal) :-
        remove_sort_queue(First_record, Open, _),
        state_record(State, _, _, _, _, First_record),
        State = Goal,
        write('Solution path is: '), nl,
        printsolution(First_record, Closed).

        % The next record is not equal to the goal
        % Generate its children, add to open and continue
        % Note that bagof in AAIS prolog fails if its goal fails,
        % I needed to use the or to make it return an empty list in this case
path(Open, Closed, Goal) :-
        remove_sort_queue(First_record, Open, Rest_of_open),
        bagof(Child, moves(First_record, Open, Closed, Child, Goal), Children),
        insert_list(Children, Rest_of_open, New_open),
        add_to_set(First_record, Closed, New_closed),
        path(New_open, New_closed, Goal),!.

        % moves generates all children of a state that are not already on
        % open or closed.  The only wierd thing here is the construction
        % of a state record, test, that has unbound variables in all positions
        % except the state.  It is used to see if the next state matches
        % something already on open or closed, irrespective of that states parent
        % or other attributes
        % Also, I've commented out unsafe since the way I've coded the water jugs
        % problem I don't really need it.
moves(State_record, Open, Closed,Child, Goal) :-
        state_record(State, _, G, _,_, State_record),
        mov(State, Next),
        % not(unsafe(Next)),
        state_record(Next, _, _, _, _, Test),
        not(member_sort_queue(Test, Open)),
```

```
        not(member_set(Test, Closed)),
        G_new is G + 1,
        heuristic(Next, Goal, H),
        F is G_new + H,
        state_record(Next, State, G_new, H, F, Child).

        %insert_list inserts a list of states obtained from a  call to
        % bagof and  inserts them in a priotrity queue, one at a time
insert_list([], L, L).
insert_list([State | Tail], L, New_L) :-
        insert_sort_queue(State, L, L2),
        insert_list(Tail, L2, New_L).

        % Printsolution prints out the solution path by tracing
        % back through the states on closed using parent links.
printsolution(Next_record, _):-
        state_record(State, nil, _, _,_, Next_record),
        write(State), nl.
printsolution(Next_record, Closed) :-
        state_record(State, Parent, _, _,_, Next_record),
        state_record(Parent, Grand_parent, _, _, _, Parent_record),
        member_set(Parent_record, Closed),
        printsolution(Parent_record, Closed),
        write(State), nl.
```

```
:- reconsult('prolog adts').
plan(State, Goal, _, Moves) :-  equal_set(State, Goal),
                                write('moves are'), nl,
                                reverse_print_stack(Moves).
plan(State, Goal, Been_list, Moves) :-
                                move(Name, Preconditions, Actions),
                                conditions_met(Preconditions, State),
                                change_state(State, Actions, Child_state),
                                not(member_state(Child_state, Been_list)),
                                stack(Child_state, Been_list, New_been_list),
                                stack(Name, Moves, New_moves),
                        plan(Child_state, Goal, New_been_list, New_moves),!.

change_state(S, [], S).
change_state(S, [add(P)|T], S_new) :-   change_state(S, T, S2),
                                        add_to_set(P, S2, S_new), !.
change_state(S, [del(P)|T], S_new) :-   change_state(S, T, S2),
                                        remove_from_set(P, S2, S_new), !.
conditions_met(P, S) :- subset(P, S).


member_state(S, [H|_]) :-       equal_set(S, H).
member_state(S, [_|T]) :-       member_state(S, T).

reverse_print_stack(S) :-       empty_stack(S).
reverse_print_stack(S) :-       stack(E, Rest, S),
                                reverse_print_stack(Rest),
                                write(E), nl.


/* sample moves */

move(pickup(X), [handempty, clear(X), on(X, Y)],
                [del(handempty), del(clear(X)), del(on(X, Y)),
                                 add(clear(Y)), add(holding(X))]).

move(pickup(X), [handempty, clear(X), ontable(X)],
                [del(handempty), del(clear(X)), del(ontable(X)),
                                 add(holding(X))]).

move(putdown(X), [holding(X)],
                [del(holding(X)), add(ontable(X)), add(clear(X)),
                                  add(handempty)]).

move(stack(X, Y), [holding(X), clear(Y)],
                [del(holding(X)), del(clear(Y)), add(handempty), add(on(X, Y)),
                                  add(clear(X))]).

go(S, G) :- plan(S, G, [S], []).
test :- go([handempty, ontable(b), ontable(c), on(a, b), clear(c), clear(a)],
                  [handempty, ontable(c), on(a,b), on(b, c), clear(a)]).
```

```
solve(true) :-!.
solve(not A) :- not(solve(A)).
solve((A,B)) :- !,solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).

p(X,Y) :- q(X), r(Y).
q(X) :- s(X).
r(X) :- t(X).
s(a).
t(b).
t(c).

test1 :- solve(p(a,b)).
test2 :- solve(p(X,Y)).
test3 :- solve(p(f,g)).
```

```
solve(true) :-!.
solve(not A) :- not(solve(A)).
solve((A,B)) :- !,solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).
solve(A) :- askuser(A).
askuser(A):- write(A),
             write('? Enter true if the goal is true, false otherwise'),
             nl, read(true).


p(X,Y) :- q(X), r(Y).
q(X) :- s(X).
r(X) :- t(X).
s(a).
t(b).
t(c).

test1 :- solve(p(a,b)).
test2 :- solve(p(X,Y)).
test3 :- solve(p(f,g)).
```

```
solve(true,_) :-!.
solve(not A, Rules) :- not(solve(A, Rules)).
solve((A,B), Rules) :- !,solve(A, Rules), solve(B, Rules).
solve(A, Rules) :- clause(A,B), solve(B, [(A:-B)|Rules]).
solve(A, Rules) :- askUser(A, Rules).

askUser(A, Rules):- write(A),
            write('? Enter true if the goal is true, false otherwise'),
            nl, read(Answer), respond(Answer, A, Rules).

respond(true,_,_).
respond(why,A,[Rule|Rules]) :- write(Rule),nl,
                               askUser(A,Rules).
respond(why,A,[]) :- askUser(A,[]).


p(X,Y) :- q(X), r(Y).
q(X) :- s(X).
r(X) :- t(X).
s(a).
t(b).
t(c).

test1 :- solve(p(a,b),[]).
test2 :- solve(p(X,Y),[]).
test3 :- solve(p(f,g),[]).
```

```
solve(true,true) :-!.
solve(not A, not ProofA) :- not(solve(A, ProofA)).
solve((A,B), (ProofA,ProofB)) :- !,solve(A, ProofA), solve(B, ProofB).
solve(A, (A:-ProofB)) :- clause(A,B), solve(B, ProofB).
solve(A, (A:-given)) :- askUser(A).

askUser(A):- write(A),
             write('? Enter true if the goal is true, false otherwise'),
             nl, read(true).


p(X,Y) :- q(X), r(Y).
q(X) :- s(X).
r(X) :- t(X).
s(a).
t(b).
t(c).

test1 :- solve(p(a,b),Proof), write(Proof),nl.
test2 :- solve(p(X,Y),Proof), write(Proof),nl.
test3 :- solve(p(f,g),Proof), write(Proof),nl.
```

```
% solve(Goal)
% Top level call.  Initializes working memory; attempts to solve Goal
% with certainty factor; prints results; asks user if they would like a
% trace.

solve(Goal) :-
        init,
        solve(Goal,C,[],1),
        nl,write('Solved '),write(Goal),
        write(' With Certainty = '),write(C),nl,nl,
        ask_for_trace(Goal).

% init
% purges all facts from working memory.

init :- retractall(fact(X)), retractall(untrue(X)).

% solve(Goal,CF,Rulestack,Cutoff_context)
% Attempts to solve Goal by backwards chaining on rules;  CF is
% certainty factor of final conclusion; Rulestack is stack of
% rules, used in why queries, Cutoff_context is either 1 or -1
% depending on whether goal is to be proved true or false
% (e.g. not Goal requires Goal be false in oreder to succeed).

solve(true,100,Rules,_).

solve(A,100,Rules,_) :-
        fact(A).

solve(A,-100,Rules,_) :-
        untrue(A).

solve(not A,C,Rules,T) :-
        T2 is -1 * T,
        solve(A,C1,Rules,T2),
        C is -1 * C1.

solve((A,B),C,Rules,T) :-
        solve(A,C1,Rules,T),
        above_threshold(C1,T),
        solve(B,C2,Rules,T),
        above_threshold(C2,T),
        minimum(C1,C2,C).

solve(A,C,Rules,T) :-
        rule((A :- B),C1),
        solve(B,C2,[rule(A,B,C1)|Rules],T),
        C is (C1 * C2) / 100,
        above_threshold(C,T).

solve(A,C,Rules,T) :-
        rule((A), C),
        above_threshold(C,T).

solve(A,C,Rules,T) :-
        askable(A),
        not known(A),
        ask(A,Answer),
        respond(Answer,A,C,Rules).

% respond( Answer, Query, CF, Rule_stack).
% respond will process Answer (yes, no, how, why, help).
% asserting to working memory (yes or no)
% displaying current rule from rulestack (why)
% showing proof trace of a goal (how(Goal)
% displaying help (help).
% Invalid responses are detected and the query is repeated.
```

```
respond(Bad_answer,A,C,Rules) :-
        not member(Bad_answer,[help, yes,no,why,how(_)]),
        write('answer must be either help, (y)es, (n)o, (h)ow or (w)hy'),nl,nl,
        ask(A,Answer),
        respond(Answer,A,C,Rules).

respond(yes,A,100,_) :-
        assert(fact(A)).

respond(no,A,-100,_) :-
        assert(untrue(A)).

respond(why,A,C,[Rule|Rules]) :-
        display_rule(Rule),
        ask(A,Answer),
        respond(Answer,A,C,Rules).

respond(why,A,C,[]) :-
        write('Back to goal, no more explanation  possible'),nl,nl,
        ask(A,Answer),
        respond(Answer,A,C,[]).

respond(how(Goal),A,C,Rules) :-
        respond_how(Goal),
        ask(A,Answer),
        respond(Answer,A,C,Rules).

respond(help,A,C,Rules) :-
        print_help,
        ask(A,Answer),
        respond(Answer,A,C,Rules).

% ask(Query, Answer)
% Writes Query and reads the Answer.  Abbreviations (y, n, h, w) are
% trnslated to appropriate command be filter_abbreviations

ask(Query,Answer) :-
        display_query(Query),
        read(A),
        filter_abbreviations(A,Answer),!.

% filter_abbreviations( Answer, Command)
% filter_abbreviations will expand Answer into Command.  If
% Answer is not a known abbreviation, then Command = Answer.

filter_abbreviations(y,yes).
filter_abbreviations(n,no).
filter_abbreviations(w,why).
filter_abbreviations(h(X),how(X)).
filter_abbreviations(X,X).

% known(Goal)
% Succeeds if Goal is known to be either true or untrue.

known(Goal) :- fact(Goal).
known(Goal) :- untrue(Goal).

% ask_for_trace(Goal).
% Invoked at the end of a consultation, ask_for_trace asks the user if
% they would like a trace of the reasoning to a goal.

ask_for_trace(Goal) :-
        write('Trace of reasoning to goal ? '),
        read(Answer),nl,
        show_trace(Answer,Goal),!.
```

```
% show_trace(Answer,Goal)
% If Answer is ``yes'' or ``y,'' show trace will display  a trace
% of Goal, as in a ``how'' query.  Otherwise, it succeeds, doing nothing.

show_trace(yes,Goal) :- respond_how(Goal).

show_trace(y,Goal) :- respond_how(Goal).

show_trace(_,_).

% print_help
% Prints a help screen.

print_help :-
        write('Exshell allows the following responses to queries:'),nl,nl,
        write('   yes - query is known to be true.'),nl,
        write('   no - query is false.'),nl,
        write('   why - displays rule currently under consideration.'),nl,
        write('   how(X) - if X has been inferred, displays trace of reasoning.'),nl,
        write('   help - prints this message.'),nl,
        write('   all commands ( except help ) may be abbreviated to first letter.'),nl,nl.

% display_query(Goal)
% Shows Goal to user in the form of a query.

display_query(Goal) :-
        write(Goal),
        write('? ').

% display_rule(rule(Head, Premise, CF))
% prints rule in IF...THEN form.

display_rule(rule(Head,Premise,CF)) :-
        write('IF       '),
        write_conjunction(Premise),
        write('THEN     '),
        write(Head),nl,
        write('CF   '),write(CF),
        nl,nl.

% write_conjunction(A)
% write_conjunction will print the components of a rule premise.  If any
% are known to be true, they are so marked.

write_conjunction((A,B)) :-
        write(A), flag_if_known(A),!, nl,
        write('     AND '),
        write_conjunction(B).

write_conjunction(A) :- write(A),flag_if_known(A),!, nl.

% flag_if_known(Goal).
% Called by write_conjunction, if Goal follows from current state
% of working memory, prints an indication, with CF.

flag_if_known(Goal) :-
        build_proof(Goal,C,_,1),
        write('     ***Known, Certainty = '),write(C).

flag_if_known(A).

% Predicates concerned with how queries.

% respond_how(Goal).
% calls build_proof to determine if goal follows from current state of working
% memory.  If it does, prints a trace of reasoning, if not, so indicates.

respond_how(Goal) :-
```

```
        build_proof(Goal,C,Proof,1),
        interpret(Proof),nl,!.

respond_how(Goal) :-
        build_proof(Goal,C,Proof,-1),
        interpret(Proof),nl,!.

respond_how(Goal) :-
        write('Goal does not follow at this stage of consultation.'),nl.

% build_proof(Goal, CF, Proof, Cutoff_context).
% Attempts to prove Goal, placing a trace of the proof in Proof.
% Functins the same as solve, except it does not ask for unknown information.
% Thus, it only proves goals that follow from the rule base and the current
% contents of working memory.

build_proof(true,100,(true,100),_).

build_proof(Goal, 100, (Goal :- given,100),_) :- fact(Goal).

build_proof(Goal, -100, (Goal :- given,-100),_) :- untrue(Goal).

build_proof(not Goal, C, (not Proof,C),T) :-
        T2 is -1 * T,
        build_proof(Goal,C1,Proof,T2),
        C is -1 * C1.

build_proof((A,B),C,(ProofA, ProofB),T) :-
        build_proof(A,C1,ProofA,T),
        above_threshold(C1,T),
        build_proof(B,C2,ProofB,T),
        above_threshold(C2,T),
        minimum(C1,C2,C).

build_proof(A, C, (A :- Proof,C),T) :-
        rule((A :- B),C1),
        build_proof(B, C2, Proof,T),
        C is (C1 * C2) / 100,
        above_threshold(C,T).

build_proof(A, C, (A :- true,C),T) :-
        rule((A),C),
        above_threshold(C,T).

% interpret(Proof).
% Interprets a Proof as constructed by build_proof,
% printing a trace for the user.

interpret((Proof1,Proof2)) :-
        interpret(Proof1),interpret(Proof2).

interpret((Goal :- given,C)):-
        write(Goal),
        write(' was given. CF = '), write(C),nl,nl.

interpret((not Proof, C)) :-
        extract_body(Proof,Goal),
        write('not '),
        write(Goal),
        write(' CF = '), write(C),nl,nl,
        interpret(Proof).

interpret((Goal :- true,C)) :-
        write(Goal),
        write(' is a fact, CF = '),write(C),nl.

interpret(Proof) :-
```

```
        is_rule(Proof,Head,Body,Proof1,C),
        nl,write(Head),write(' CF = '),
        write(C), nl,write('was proved using the rule'),nl,nl,
        rule((Head :- Body),CF),
        display_rule(rule(Head, Body,CF)), nl,
        interpret(Proof1).

% isrule(Proof,Goal,Body,Proof,CF)
% If Proof is of the form Goal :- Proof, extracts
% rule Body from Proof.

is_rule((Goal :- Proof,C),Goal, Body, Proof,C) :-
        not member(Proof, [true,given]),
        extract_body(Proof,Body).

% extract_body(Proof).
% extracts the body of the top level rule from Proof.

extract_body((not Proof, C), (not Body)) :-
        extract_body(Proof,Body).

extract_body((Proof1,Proof2),(Body1,Body2)) :-
        !,extract_body(Proof1,Body1),
        extract_body(Proof2,Body2).

extract_body((Goal :- Proof,C),Goal).


% Utility Predicates.

retractall(X) :- retract(X), fail.
retractall(X) :- retract((X:-Y)), fail.
retractall(X).

member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

minimum(X,Y,X) :- X =< Y.
minimum(X,Y,Y) :- Y < X.

above_threshold(X,1) :- X >= 20.
above_threshold(X,-1) :- X =< -20.
```

```
% solve/2 succeeds with
%       argument 1 bound to a goal proven true using the current knowledge base
%       argument 2 bound to the confidence in that goal.
%
% solve/2 calls solve/4 with appropriate arguments.  After solve/4 has completed,
% it writes the conclusions and prints a trace.


solve(Goal, CF) :-
        retractall(known(_,_)),
        print_instructions,
        solve(Goal, CF, [], 20),
        write(Goal), write(' was concluded with certainty '), write(CF), nl,nl,
        build_proof(Goal, _, Proof),nl,
        write('The proof is '),nl,nl,
        write_proof(Proof, 0), nl,nl.

%solve/4 succeeds with
%       argument 1 bound to a goal proven true using the current knowledge base
%       argument 2 bound to the confidence in that goal.
%       argument 3 bound to the current rule stack
%       argument 4 bound to the threshold for pruning rules.
%
%solve/4 is the heart of exshell.  In this version, I have gone back to the
% simpler version.  It still has problems with negation, but I think that
% this is more a result of problems with the semantics of Stanford Certainty
% factors than a bug in the program.
% The pruning threshold will vary between 20 and -20, depending whether,
% we are trying to prove the current goal true or false.
% solve/4 handles conjunctive predicates, rules, user queries and negation.
% If a predicate cannot be solved using rules, it will call it as a PROLOG predicate.

% Case 1: truth value of goal is already known
solve(Goal, CF, _, Threshold) :-
        known(Goal, CF),!,
        above_threshold(CF, Threshold).

% Case 2: negated goal
solve( not(Goal), CF, Rules, Threshold) :- !,
        invert_threshold(Threshold, New_threshold),
        solve(Goal, CF_goal, Rules, New_threshold),
        negate_cf(CF_goal, CF).

% Case 3: conjunctive goals
solve((Goal_1,Goal_2), CF, Rules, Threshold) :- !,
        solve(Goal_1, CF_1, Rules, Threshold),
        above_threshold(CF_1, Threshold),
        solve(Goal_2, CF_2, Rules, Threshold),
        above_threshold(CF_2, Threshold),
        and_cf(CF_1, CF_2, CF).

%Case 4: backchain on a rule in knowledge base
solve(Goal, CF, Rules, Threshold) :-
        rule((Goal :- (Premise)), CF_rule),
        solve(Premise, CF_premise,
                [rule((Goal :- Premise), CF_rule)|Rules], Threshold),
        rule_cf(CF_rule, CF_premise, CF),
        above_threshold(CF, Threshold).

%Case 5: fact assertion in knowledge base
solve(Goal, CF, _, Threshold) :-
        rule(Goal, CF),
        above_threshold(CF, Threshold).

% Case 6: ask user
solve(Goal, CF, Rules, Threshold) :-
```

```
        askable(Goal),
        askuser(Goal, CF, Rules),!,
        assert(known(Goal, CF)),
        above_threshold(CF, Threshold).

% Case 7A: All else fails, see if goal can be solved in prolog.
solve(Goal, 100, _, _) :-
        call(Goal).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%
% Certainty factor predicates.  Currently, these implement a variation of
% the MYCIN certainty factor algebra.
% The certainty algebra may be changed by modifying these predicates.

% negate_cf/2
%       argument 1 is a certainty factor
%       argument 2 is the negation of that certainty factor

negate_cf(CF, Negated_CF) :-
        Negated_CF is -1 * CF.

% and_cf/3
%       arguments 1 & 2 are certainty factors of conjoined predicates
%       argument 3 is the certainty factor of the conjunction

and_cf(A, B, A) :- A =< B.
and_cf(A, B, B) :- B < A.

%rule_cf/3
%       argument 1 is the confidence factor given with a rule
%       argument 2 is the confidence inferred for the premise
%       argument 3 is the confidence inferred for the conclusion

rule_cf(CF_rule, CF_premise, CF) :-
        CF is CF_rule * CF_premise/100.

%above_threshold/2
%       argument 1 is a certainty factor
%       argument 2 is a threshold for pruning
%
% If the threshold, T, is positive, assume we are trying to prove the goal
% true.  Succeed if CF >= T.
% If T is negative, assume we are trying to prove the goal
% false.  Succeed if CF <= T.

above_threshold(CF, T) :-
        T >= 0, CF >= T.
above_threshold(CF, T) :-
        T < 0, CF =< T.

%invert_threshold/2
%       argument 1 is a threshold
%       argument 2 is that threshold inverted to account for a negated goal.
%
% If we are trying to prove not(p), then we want to prove p false.
% Consequently, we should prune proofs of p if they cannot prove it
% false.  This is the role of threshold inversion.

invert_threshold(Threshold, New_threshold) :-
        New_threshold is -1 * Threshold.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%
% Predicates to handle user interactions.  As is typical, these
% constitute the greatest bulk of the program.
```

```
%
% askuser/3
%       argument 1 is a goal whose truth is to be asked of the user.
%       argument 2 is the confidence the user has in that goal
%       argument 3 is the current rule stack (used for why queries).
%
% askuser prints the query, followed by a set of instructions.
% it reads the response and calls respond/4 to handle that response

askuser(Goal, CF, Rules) :-
        nl,write('User query:'), write(Goal), nl,
        write('? '),
        read(Answer),
        respond(Answer,Goal, CF, Rules).

%respond/4
%       argument 1 is the user response
%       argument 2 is the goal presented to the user
%       argument 3 is the CF obtained for that goal
%       argument 4 is the current rule stack (used for why queries).
%
% The basic scheme of respond/4 is to examine the response and return
% the certainty for the goal if possible.
% If the response is a why query, how query, etc., it processes the query
% and then calls askuser to re prompt the user.

% Case 1: user enters a valid confidence factor.
respond(CF, _, CF, _) :-
        number(CF),
        CF =< 100, CF >= -100.

% Case 2: user enters 'n' for no.  Return a confidence factor of -1.0
respond(n, _, -100, _).

% Case 3: user enters 'y' for yes.  Return a confidence factor of 1.0
respond(y, _, 100, _).

% Case 4: user enters a pattern that matches the goal.  This is useful if
% the goal has variables that need to be bound.
respond(Goal, Goal, CF, _) :-
        write('Enter confidence in answer'), nl,
        write('?'),
        read(CF).

% Case 5: user enters a why query
respond(why, Goal, CF, [Rule|Rules]) :-
        write_rule(Rule),
        askuser(Goal, CF, Rules).

respond(why, Goal, CF, []) :-
        write('Back to top of rule stack.'),
        askuser(Goal, CF, []).

% Case 6: User enters a how query.  Build and print a proof
respond(how(X), Goal, CF, Rules) :-
        build_proof(X, CF_X, Proof),!,
        write(X), write(' was concluded with certainty '), write(CF_X), nl,nl,
        write('The proof is '),nl,nl,
        write_proof(Proof, 0), nl,nl,
        askuser(Goal, CF, Rules).

% User enters how query, could not build proof
respond(how(X), Goal, CF, Rules):-
        write('The truth of '), write(X), nl,
        write('is not yet known.'), nl,
        askuser(Goal, CF, Rules).
```

```
% Case 7: User asks for the rules that conclude a certain predicate
respond(rule(X), _, _, _) :-
        write('The following rules conclude about '), write(X),nl,nl,
        rule((X :- Premise), CF),
        write(rule((X :- Premise), CF)), nl,
        fail.

respond(rule(_),Goal, CF, Rules) :-
        askuser(Goal, CF, Rules).

% Case 8: User asks for help.
respond(help, Goal, CF, Rules) :-
        print_instructions,
        askuser(Goal, CF, Rules).

%Case 9: User wants to quit.
respond(quit,_, _, _) :- quit.

%Case 10: Unrecognized input
respond(_, Goal,CF, Rules) :-
        write('Unrecognized response.'),nl,
        askuser(Goal, CF, Rules).

%build_proof/3
%       argument 1 is the goal being traced.
%       argument 2 is the CF of that goal
%       argument 3 is the proof tree
%
% build_proof does not do threshold pruning, so it can show
% the proof for even goals that would not succeed.
build_proof(Goal, CF, ((Goal,CF) :- given)) :-
        known(Goal, CF),!.

build_proof(not(Goal), CF, not(Proof)) :- !,
        build_proof(Goal, CF_goal, Proof),
        negate_cf(CF_goal, CF).

build_proof((Goal_1, Goal_2), CF, (Proof_1, Proof_2)) :- !,
        build_proof(Goal_1, CF_1, Proof_1),
        build_proof(Goal_2, CF_2, Proof_2),
        and_cf(CF_1, CF_2, CF).

build_proof(Goal, CF, ((Goal,CF) :- Proof)) :-
        rule((Goal :- (Premise)), CF_rule),
        build_proof(Premise, CF_premise, Proof),
        rule_cf(CF_rule, CF_premise, CF).

build_proof(Goal, CF, ((Goal, CF):- fact)) :-
        rule(Goal, CF).

build_proof(Goal, 1, ((Goal, 1):- call)) :-
        call(Goal).


% write_proof/2
%       argument 1 is a portion of a proof tree
%       argument 2 is the depth of that portion (for indentation)
%
% writes out a proof tree in a readable format
write_proof(((Goal,CF) :- given), Level) :-
        indent(Level),
        write(Goal), write(' CF= '), write(CF),
        write(' was given by the user'), nl,!.

write_proof(((Goal, CF):- fact), Level) :-
        indent(Level),
```

```
        write(Goal), write(' CF= '), write(CF),
        write(' was a fact in the knowledge base'), nl,!.

write_proof(((Goal, CF):- call), Level) :-
        indent(Level),
        write(Goal), write(' CF= '), write(CF),
        write(' was proven by a call to prolog'), nl,!.

write_proof(((Goal,CF) :- Proof), Level) :-
        indent(Level),
        write(Goal), write(' CF= '), write(CF), write(' :-'), nl,
        New_level is Level + 1,
        write_proof(Proof, New_level),!.

write_proof(not(Proof), Level) :-
        indent(Level),
        write('not'),nl,
        New_level is Level + 1,
        write_proof(Proof, New_level),!.

write_proof((Proof_1, Proof_2), Level) :-
        write_proof(Proof_1, Level),
        write_proof(Proof_2, Level),!.

% indent/1
%       argument 1 is the number of units to indent
indent(0).
indent(I) :-
        write('     '),
        I_new is I - 1,
        indent(I_new).

%print_instructions/0
% Prints all options for user responses
print_instructions :-
        nl, write('Response must be either:'), nl,
        write('    A confidence in the truth of the query.'), nl,
        write('      This is a number between -100 and 100.'), nl,
        write('    y or n, where y is equivalent to a confidence of 100 and'), nl,
        write('                  n is equivalent to a confidence of -100.'), nl,
        write('    Goal, where Goal is a pattern that will unify with the query'), nl,
        write('    why.'),nl,
        write('    how(X), where X is a goal'),nl,
        write('    rule(X) to display all rules that conclude about X.'),nl,
        write('    quit, to terminate consultation'),nl,
        write('    help, to print this message'), nl.


% write_rule/1
%       argument 1 is a rule specification
% writes out the rule in a readable format
write_rule(rule((Goal :- (Premise)), CF)) :-
        write(Goal), write('if'), nl,
        write_premise(Premise),nl,
        write('CF = '), write(CF), nl.

write_rule(rule(Goal, CF)) :-
        write(Goal),nl,
        write('CF = '), write(CF), nl.


% write_premise
%       argument 1 is a rule premise
% writes it in a readable format.
write_premise((Premise_1, Premise_2)) :-
        !, write_premise(Premise_1),
        write_premise(Premise_2).
```

```
write_premise(not Premise) :-
        !, write('     '), write(not),write(' '), write(Premise),nl.
write_premise(Premise) :-
        write('     '), write(Premise),nl.

% Utility Predicates.

retractall(X) :- retract(X), fail.
retractall(X) :- retract((X:-Y)), fail.
retractall(X).



% This is the sample automotive diagnostic knowledge base for use
% with the EXSHELL expert system shell in section 12.2 of the text.
% When running it, be sure to load it with the file containing
% EXSHELL.

% To start it, give PROLOG the goal:
%               solve(fix(X), CF).

% Knowledge Base for simple automotive diagnostic expert system.
% Top level goal, starts search.

rule((fix(Advice) :-
        (bad_component(X),fix(X, Advice))), 100).


% rules to infer bad component:

rule((bad_component(starter) :-
        (bad_system(starter_system),lights(come_on))),50).
rule((bad_component(battery) :-
        (bad_system(starter_system),not(lights(come_on)))),90).
rule((bad_component(timing) :-
        (bad_system(ignition_system), not(tuned_recently))),80).
rule((bad_component(plugs) :-
        (bad_system(ignition_system),plugs(dirty))),90).
rule((bad_component(ignition_wires) :-
        (bad_system(ignition_system),
        not(plugs(dirty)), tuned_recently)),80).


% Rules to infer system that failed.

rule((bad_system(starter_system) :-
        (not(car_starts), not(turns_over))),90).
rule((bad_system(ignition_system) :-
        (not(car_starts), turns_over,gas_in_carb)),80).
rule((bad_system(ignition_system) :-
        (runs(rough),gas_in_carb)),80).
rule((bad_system(ignition_system) :-
        (car_starts, runs(dies),gas_in_carb)),60).

% Rules to make reccommendation for repairs.

rule(fix(starter, 'replace starter'),100).
rule(fix(battery, 'replace or recharge battery'),100).
rule(fix(timing, 'get the timing adjusted'),100).
rule(fix(plugs, 'replace spark plugs'),100).
rule(fix(ignition_wires, 'check ignition wires'),100).

% askable descriptions

askable(car_starts).
askable(turns_over).
askable(lights(_)).
askable(runs(_)).
```

```
askable(gas_in_carb).
askable(tuned_recently).
askable(plugs(_)).
```

```
% Knowledge Base for simple automotive diagnostic expert system.

% rule base:

% Top level goal, starts search.
rule((fix_car(Advice) :-
        bad_component(Y), fix(Y,Advice)),100).

% rules to infer bad component:

rule((bad_component(starter) :-
        bad_system(starter_system),lights(come_on)),50).
rule((bad_component(battery) :-
        bad_system(starter_system),not lights(come_on)),90).
rule((bad_component(timing) :-
        bad_system(ignition_system), not tuned_recently),80).
rule((bad_component(plugs) :-
        bad_system(ignition_system),plugs(dirty)),90).
rule((bad_component(ignition_wires) :-
        bad_system(ignition_system),
        not plugs(dirty), tuned_recently),80).

% Rules to infer basic system that failed.

rule((bad_system(starter_system) :-
        not car_starts, not turns_over),90).
rule((bad_system(ignition_system) :-
        not car_starts, turns_over,gas_in_carb),80).
rule((bad_system(ignition_system) :-
        car_starts, runs(rough),gas_in_carb),80).
rule((bad_system(ignition_system) :-
        car_starts, runs(dies),gas_in_carb),60).

% Rules to make reccommendation for repairs.

rule(fix(starter,'replace starter'),100).
rule(fix(battery,'replace or recharge battery'),100).
rule(fix(timing, 'get the timing adjusted'),100).
rule(fix(plugs, 'replace spark plugs'),100).
rule(fix(ignition_wires, 'check ignition wires'),100).

% askable descriptions

askable(car_starts).
askable(turns_over).
askable(lights(X)).
askable(runs(X)).
askable(gas_in_carb).
askable(tuned_recently).
askable(plugs(X)).
```

```
/* A recursive Semantic Net Parser */


utterance(X, Sentence_graph) :- sentence(X, [], Sentence_graph).

sentence(Start, End, Sentence_graph) :-
                nounphrase(Start, Rest, Subject_graph),
                verbphrase(Rest, End, Predicate_graph),
                join([agent(Subject_graph)],Predicate_graph, Sentence_graph).

nounphrase([Noun|End], End, Noun_phrase_graph) :-
                noun(Noun, Noun_phrase_graph).

nounphrase([Article, Noun| End], End, Noun_phrase_graph) :-
                article(Article), noun(Noun, Noun_phrase_graph).

verbphrase([Verb| End] End, Verb_phrase_graph) :-
                verb(Verb, Verb_phrase_graph).

verbphrase([Verb|Rest], End, Verb_phrase_graph) :-
                verb(Verb, Verb_graph),
                nounphrase(Rest, End, Noun_phrase_graph),
                join([object(Noun_phrase_graph)],Verb_graph,Verb_phrase_graph).


join_frames([A|B], C, D, OK) :-
                join_slot_to_frame(A, C, E), !,
                join_frames(B, E, D, ok).

join_frames([A|B], C, [A|D], OK) :-
                join_frames(B, C, D, OK), !.

join_frames([], A, A, ok).

join_slot_to_frame(A, [B|C], [D|C]) :-
                join_slots(A, B, D).

join_slot_to_frame(A, [B|C], [B|D]) :-
                join_slot_to_frame(A, C, D).
                join_slots(A, B, D) :-
                functor(A, FA, _),
                functor(B, FB, _),
                match_with_inheritance(FA, FB, FN),
                arg(1, A, Value_a),
                arg(1, B, Value_b),
                join(Value_a, Value_b, New_value),
                D =..[FN|[New_value]].

join(X, X, X).

join(A, B, C) :- isframe(A), isframe(B), !,
                join_frames(A, B, C, not_joined).

join(A, B, C) :- isframe(A), is_slot(B), !,
                join_slot_to_frame(B, A, C).

join(A, B, C) :- isframe(B), is_slot(A), !,
                join_slot_to_frame(A, B, C).

join(A, B, C) :- is_slot(A), is_slot(B), !,
                join_slots(A, B, C).

isframe([_|_]).
isframe([]).

is_slot(A) :- functor(A, _, 1).

match_with_inheritance(X, X, X).
```

```
match_with_inheritance(dog, animate, dog).
match_with_inheritance(animate, dog, dog).
match_with_inheritance(man, animate, man).
match_with_inheritance(animate, man, man).
match_with_inheritance(animate, man, john).

article(a).
article(the).

noun(fido, [dog(fido)]).
noun(man, [man(X)]).
noun(john, [man(john)]).
noun(dog, [dog(X)]).

verb(likes,[action([liking(X)]),agent([animate(X)]),
               object([animate(Y)])]).

verb(bites,[action([biting(Y)]),agent([dog(X)]),
               object([animate(Z)])]).


test1 :- utterance([the, man, likes, the, dog], X).
test2 :- utterance([fido, likes, the, man], W).
test3 :- utterance([john, bites, fido], Z).
```

Object-Oriented Programming I

In the "real world" we are surrounded with *objects* - examples include people, animals, planes, buildings and the like. There are both animate and inanimate objects.

Abstraction allows us to see people instead of colored dots on a screen, a beach instead of grains of sand, a forest instead of individual trees, and houses instead of individual bricks.

Generalization allows us to use the same concept for different things.

OOP

Object-oriented programming (OOP) models real-world objects with software counterparts.

An ***object*** has a set of characteristics:

- a set of attributes (size, shape, color, weight, ...)
- a set of behaviors
    - a ball - rolls, bounces, inflates, deflates, ...
    - a baby - cries, sleeps, crawls, walks, blinks, ...
    - a car - accelerates, breakes, turns, ...
    - a towel - absorbs water, ...

A ***class*** is a set of objects that have the same characteristics. A class may be thought of as a blueprint for a set of objects.

A new class ***inherits*** the characteristics of one (*single inheritance*) or more (*multiple inheritance*) classes and adds additional characteristics.

OOP ***encapsulates*** data (attributes) and functions (behaviors) into packages called objects. Objects have the property of ***information hiding***. This means that implementation details are hidden within the objects themselves.

Communication between objects is across well defined ***interfaces***.

Example: A car consists of an engine, a transmission, exhaust systems, etc. It is possible to use each subsystem without knowing how they work internally.

OOP concentrates on creating ***user-defined types*** called ***classes***. Each class contains data as well as the set of functions that manipulate the data. The data components are called ***data members*** and the function components are called ***member functions*** or ***methods***.

Nouns in a problem specification help the software engineer to identify the set of classes needed to implement the system and the verbs help to identify the functions.

A well designed set of classes leads to software that is ***reusable***. Well designed reusable software components enhance the speed and quality of future programming projects.

Notes

What OOP calls a class or object, is to the mathematician, a ***many-sorted algebra***. The various attributes are the sorts and the functions are the operations of the algebra.

Procedural programming is ***action oriented*** where the unit of programming is the function. Verbs in a problem specification help the programmer to identify the set of functions needed to solve the problem. ***Structured programming*** is also action oriented but the unit of programming is the structured command. Both procedural and structured programming are part of OOP.

Adapted from Deitel & Deitel **C++ How to Program** 2nd ed. Prentice Hall 1998.

Philosophy of Science

Presented April 27 & 28, 1999 in Philosophy of Science

Notes and comments on: David Deutsch - **The Fabric of Reality - the Science of Parallel Universes and its implications** Penguin Books 1997

1. The Theory of Everything
    - *Instrumentalism*: The purpose of a scientific theory is to predict the outcome of a scientific experiment - a theory is an instrument for making predictions.
    - Illustration: Suppose we had an `oracle' that could predict the outcome of any experiment but which does not provide an explanation. *Comment*: Experiments help us collect data, explanations (theories) guide designs.
    - *Positivism*: all statements other than those describing or predicting observations are not only superfluous, but meaningless.
    - The majority of theories are rejected because they provide bad explanations not because they provide bad predictions.
    - *Reductionism*: Explanations are constructed by analyzing things into components. That is, explanations are based on the behavior of the fundamental constituents. Explanation always consists of analyzing a system into smaller, simpler systems and that all explanation is of later events in terms of earlier events (cause and effect).
    - *Holism*: The only legitimate explanations are in terms of higher-level systems.
    - Emergent phenomenon: is one about which there are comprehensible facts or explanations that are not simply deducible from lower-level theories, but which may be explicable or predictable by higher-level theories referring directly to phenomenon.
    - *Fabric of reality*: composed of four main strands -
        - quantum theory, the
        - theory of evolution, the
        - theory of knowledge (epistemology), and the
        - theory of computation.
2. **Virtual Reality** - a situation in which the user is given the experience of being in a specified environment.
   Does there exist an objective, physical reality independent of the mind?

   States of reality
   dream state
   wakeful state
   mental illness
   simple disagreement
   magic and illusion
   mystical world view
   physical world view

dynamic (changing) world view
monotonically changing world view
"scientific model" as virtual reality
Testing (software, scientific experiments) demonstrates the presence of faults and may increase confidence but does not prove correctness.

3. **Universality and the Limits of Computation**
    - The *diagonal argument*
        - | ***N*** |
        - countability of the rationals
        - | [0,1] | = | ***R*** |
        - uncountability of the reals (Cantor's diagonal argument)
        - | powerset of the naturals | = | (0,1) | or |***N*** -> **{ 0, 1}**| = |[0,1]| = |***R*** |
        - | ***N*** | < | ***R*** |
    - Algorithm (or effective procedure)
        - characteristics
            - finitely describable
            - discrete steps
        - Turing machine: has a finite control, an input tape that is divided into cells, and a tape head tht scans one cell of the tape at a time. The tape has a leftmost cell but is infinite to the right. Each cell may hold exactly one tape symbol. Initially, the first *n* cells contain the input. The remaining cells are blank.
            - Action: in one move the Turing machine, depending upon the symbol scanned by the type head and the state of the finite control,
                - changes state,
                - prints a symbol on the tape cell scanned, replacing what was written there, and
                - moves its head left or right one cell.
            - Description
                - finite set of states including a start state and a set of final states,
                - finite set of tape symbols including a blank and a set of input symbols, and
                - a next move function : (state, tape symbol) -> (state, tape symbol, direction)
        - Computable languages and functions
            - recursive set/total recursive function: TM halts on all inputs.
            - recursively enumerable set/partial recursive function: TM fails to halt on strings not in the set.
        - Nondeterminism: deterministic and nondeterministic TMs compute the same functions.
    - Decideability: A *decision procedure* is an algorithm which given a statement, determines whether or not the statement is true (i.e. returns an answer of true or false). Some undecideable problems:
        - whether a statement S in first-order logic is true or false (Godel's incompleteness theorem),
        - the Halting Problem (whether a given program P, with input I, will terminate).
        - whether the complement of a context-free language is empty,

        - Theorem: There exist noncomputatble functions.

          Proof: Since
          |*N* -> { 0, 1}| = |*R*|,
          algorithms have finite descriptions,
          | the set of finite descriptions | = | ***N*** |,
          There exists functions that do not have finite descriptions
    - Computability - Church (lambda calculus), Kleene (recursive functions), Post, Markof, Turing (Turing machine)
        - deterministic vs nondeterministic Turing machines

As a computer scientist, I am

- a formalist,
- a constructivist,
- interested in distributed computing and
- artificial intelligence.

Formalism

Formalism works with symbols and rules for changing one string of symbols into another (a formal system). A formal system consists of

- symbols,
- rules for describing acceptable sequences of symbols (formation rules),
- a collection of sequences of symbols (axioms) determined by some *meta constraint*, and
- rules (of inference) for transforming (while maintaining the meta constraints) one sequence of symbols into another sequence of symbols (theorem).

A formal system is *complete* if for every sequence that is acceptable by the meta constraints, there is a sequence of inferences that derives it from the axioms.

Godel's incompleteness theorem describes the boundaries of formalism -- No interesting formal system can be complete.

The meta constraint is called the model and is said to model the axioms. In the usual approach to formalism, the model is a portion of reality and the formalism is an axiomatization of that portion of reality. The tie with reality prevents inconsistencies from arising (assuming reality cannot be inconsistent).

Herbrand's approach to logic was to construct a model by selecting a set of constants, functions, and predicates, and using them to form a base (a set of terms) ... *Herbrand base*

The Herbrand construction permits the construction of arbitrary logical systems derived from imagination rather than reality, i.e. virtual reality.

Constructivism (intuitionism)

Constructivism insists that the only real objects are those objects for which there is a mechanism (program) for its construction. Constructivists reject existence proofs which rely on proof by contradiction (as they are non-constructive). They also reject proofs which apply the law of the excluded middle to infinite sets. However, conclusions derived from non-constructive proofs may be used to inspire search for constructive proofs or an approximate construction.

Distributed computing

A *distributed system* is an interconnected collection of autonomous *nodes* (computers, processes, or processors) where

- *autonomous* means that each node has its own control and
- *interconnected* means that the nodes must be able to exchange information.

Distributed systems differ from centralized systems in three essential respects

1. Lack of knowledge of global state.
2. Lack of a global time-frame.
3. Non-determinism.

The challenges of distributed computing in a *point-to-point* configuration (a wide area network) include

- the reliability of point-to-point data exchange,
- the selection of communication paths,
- congestion control,
- deadlock prevention, and
- security.

The challenges of distributed computing in a *bus* type configuration (local area network) include

- broadcasting and synchronization,
- election (of a leader),
- termination detection,
- resource allocation,
- mutual exclusion,
- deadlock detection and resolution, and
- distributed file maintenance.

A single processor machine can only execute an algorithm one instruction at a time. A multiple processor machine can execute an algorithm several instructions at a time (i.e., concurrently). If the instructions are independent, the algorithm is executed in a shorter amount of time. If the instructions are dependent, then either incorrect results are produced or timing constraints must be added to the algorithm. Executing the instructions based on timing signals from a central clock restores correct results. When the processors are distributed widely in space,

Distributed algorithms are those algorithms that are executed in a distributed environment where coordination is achieved by the exchange of messages rather than in response to commands from a central control. In such an environment, there may be a significant time difference between the sending of the message and the reception of the message.

Artificial Intelligence

For artifical intelligence to exist, it must be realized in an object that

1. is capable of interacting with its environment (I/O),
2. is capable of drawing inferences, and
3. must be goal directed.

Each of these attributes must interact with the other two, with the possibility of modifying each other's behavior.

INFO105 Personal Computing

Summer 95

Description

An introduction to personal computing and MS-DOS using IBM PC compatible computers. Lectures are offered in a lab setting with each student working with a computer. Topics include IBM PC hardware basics. MS-DOS fundamentals, word processing, data base systems, and electronic spreadsheets. Does not apply toward a major or minor in computer science. The course provides four lecture/lab periods per week.

Goals

Upon completion of the course, you will be

- familiar with basic computer concepts in the personal computing environment,
- able to use basic DOS and Windows commands,
- able to use word processing software (WordPerfect 6.1), and
- able to use a spredsheet (Lotus 123R4), and
- able to use a data base (Access 2.0).

Resources

Textbooks:

- An Introduction to DOS 5.0/6.0
- An Introduction to Microsoft Windows 3.1
- Introductory WordPerfect 6.0 for Windows
- Introductory Lotus 1-2-3 Release 4.0 for Windows
- Introductory Microsoft Access for Windows

Files:

assorted directories and files will be available in K:\CPTR105

Lab Instructors:

Jon Duncan, Mark Foster: The CPTR 395 class.

Lecture Schedule

Introduction (A. Aaby)
1-6: Introduction to Computer Concepts, MS-DOS and MS-Windows (Jon Duncan)
7-15: Introduction to WordPerfect 6.1 (Mark Foster)
16-24: Introduction to Lotus123r4 (Jon Duncan)

25-30: Introduction to Access (Mark Foster)

Evaluation

The course grade is determined by the quantity and quality of work completed in the areas indicated in the following table. The percentages listed are a rule of thumb. The actual percentages used may be lower, depending on the distribution of scores in the class.

GRADING WEIGHTS		LETTER GRADES	
Homework	50%	As	90 - 100%
Tests	50%	Bs	80 - 89%
		Cs	70 - 79%
		Ds	60 - 69%

95.5.17 a. aaby

Introduction

The Syllabus

Goals

Upon completion of the course, you will be

- familiar with basic computer concepts in the personal computing environment,
- able to use basic DOS and Windows commands,
- able to use word processing software (WordPerfect 6.1), and
- able to use a spredsheet (Lotus 123R4), and
- able to use a data base (Access 2.0).

This course is key to your academic and professional career. It can help you get your entry level job.

Instructors

Jon Duncan, Mark Foster

Instruction

Lecture, Tutorial Assignment, Assignments, and Tests

Evolving nature of Software

Constant learning -- Instructors will not know all the details.

Grades

Do your homework, take the tests, you will most likely pass.

PCs

Components
: Keyboard, CRT (screen, monitor), Mother-board: CPU, Memory (8 mb ram), controller cards, ethernet card; diskdrive

Human-Computer I/O
: Keyboard, CRT (screen, monitor)

Operating System
    MS-DOS 6.X
Windowing Environment
    MS-Windows 3.1
File server (HAL)
Network

Getting Started

Login
    User Name; Password (student id number)
DOS prompt; change password
Halapp
LogOut

CPTR 141 Introduction to Programming 4

WARNING: Subject to change

Syllabus
Resources

ONLINE Forum for CS Students

Lectures

Topic	Assignment/Lab
Introduction OOP MS Visual C++	Read chapter 1 Lab 1 OS, Web, C++ IDE
Program Development Basic C++ and Structured Programming	Read chapter 2 Lab 2 Menu driven programming, basic control structures
More C++	Read chapter 3 Lab 3 Built-in functions
Modular Design	Lab 4 Modularity: functions, procedures, parameters and scope
TEST	
1D and 2D Arrays	Lab 5 Arrays: Searching and Sorting
Files	Lab 6 Files: numeric data and text files
Strings & Pointers	Lab 7 Text processing
Classes, Objects - Data Abstrctions	Lab 8 Address book
Simulations and Numeric Computation	Lab 6 Simulation and Numeric Computation
Ordinal Data Type: Enumerated and Subrange	
TEST	

Multidimensional Arrays	Lab 8 Multidimensional Arrays: the Game of Life (cellular automata)
Records and Pointers	Lab 9 Record and Pointers: Linked Lists
FINAL	

Old stuff
Overview of Pascal
A *pattern* based view of Pascal
Sample Pascal Program

99.3.15 a. aaby

CPTR141 Introduction to Programming - 4

Spring 99

Description

This course covers the fundamentals of algorithm design and analysis, structured programming and programming style. Topics include: top-down design, data types, control structures, procedures, scope, I/O, error recovery, recursion and simple data structures. The imperative programming language **C++** will be used for examples and in assignments. The course provides three lectures and one lab period per week.

Goals: Upon completion of this course you will

- know the outline of the history of computer science, its knowledge domains and the source of its theoretical methods.
- know how to use the DOS/Windows95/NT or Unix(Linux) environment in PCs to solve programming problems.
- understand algorithms in terms of the assignment operation (including input and output), and the control structures: sequential composition, selection and repetition.
- understand data in terms of simple types (integer, real, character, string, enumerated) and compound types (array, file).
- understand procedures, functions and parameters.
- be able to apply the basic principles of software engineering to construct programs in C++.

Resources

Textbook:

Wilks, Ian. Instant C++ Programming WROX Press 1994
Deitel & Deitel. *C++ How to Program* 2nd ed. Prentice Hall 1998
C++: Programming and Problem Solving -- Leestma & Nyhoff
Old Lecture Notes

Labs & Files:

assorted directories and files will be available on http://cs.wwc.edu/~aabyan/141/resources

Tutoring:

contact the Teaching Learning Center (TLC)

WWW:

This and related documents are on the WWW (http://cs.wwc.edu/~aabyan/141).

Evaluation

The course grade is determined by the quantity and quality of work completed in the areas indicated in the following table. The percentages listed are a rule of thumb. The actual percentages used may be lower, depending on the distribution of scores in the class. The grade expectations document helps to explain the different grades and the grading criteria document explains the grading procedure for programs. Programs must be submitted electronically to the appropriate subdirectory in K:\CLASS\CPTR\141 and must include the heading found in ????.

GRADING WEIGHTS		LETTER GRADES	
Labs & Homework	50%	As	90 - 100%
Tests	50%	Bs	80 - 89%
		Cs	70 - 79%
		Ds	60 - 69%

Estimated ABET Category Content

Engineering Science:
1/2 credits or 12.5%
Engineering Design:
3 credits or 75%
Other:
1/2 credit or 12.5%

99.3.15 a. aaby

Fatal error: Failed opening required '/var/lib/apache/phorum/include/forums.php' (include_path='.:/usr/local.cs/php-4.0.6/lib/php') in **/home/cs_dept/web/phorum/common.php** on line **302**

Lecture 1: Introduction

This lecture takes 2 1/2 class periods.

HANDOUT: Syllabus

Computer Science -- An overview

Computing is simultaneously a mathematical, scientific, and engineering discipline. Therefore, computing professionals employ the working methodologies from these areas in the course of research, development and applications development.

Methods - mathematics, science, engineering Theory: (from Mathematics)

- Definitions and axioms
- Theorems
- Proofs
- Interpretations of results

Abstraction: (from Science)

- Data collection and hypothesis formation
- Modeling and prediction
- Design of an experiment
- Analysis of results

Design: (from Engineering)

- Requirements
- Specifications
- Design and Implementation
- Testing and Analysis

Knowlege domains

Algorithms and Data Structures
: classes of problems and efficient solutions

Architecture
: efficient, reliable computing systems -- processors, memory, communications, software interfaces

Artificial Intelligence and Robotics
: simulation of animal or human behavior -- inference, deduction, pattern recognition, knowledge representation, expert systems

Database and Information Retrieval
: organizing information and algorithms for efficient access and update

Human-Computer Interaction
: graphics and human factors

Numerical and Symbolic Computation

Operating Systems

Programming Languages

Software Methodology and Engineering
: specification, design, and implementation of large software systems.

Social and Professional Context
: cultural, social, legal, and ethical issues.

Some Definitions:

1. A *computational model* is a collection of values and operations. Example: Turing machine
2. A *computation* is the application of a sequence of operations to a value to yield another value.
3. A *program* is a specification of a computation.
4. A *programming language* is a notation for writing programs.
5. The *syntax* of a programming language refers to the structure of programs.
6. The *semantics* of a programming language describe the relationship between the syntactical elements and the model of computation.
7. The *pragmatics* of a programming language describe the degree of success with which a programming language meets its goals both in its faithfulness to the underlying model of computation and in its utility for human programmers.

CE vs CS vs CIS

History

to be read and studied by students

Computer Systems--Hierarchy of Machines

Keywords: Computer, bus, CPU, Memory, Disk, Keyboard, Screen, Fetch/Execute cycle, instruction set, machine language, memory mapped I/O, registers, accumulator, program counter, instruction register, code condition register, assembly language.

Physics

- electricity, magnetism

Mathematics

- boolean algebra
- Number Systems
- Positional Notation (base, radix point, expanded form)
- binary (base 2)
- octal (base 8)
- hexadecimal (base 16 - 0..9, A..F)

Digital logc

- nand, nor, inverter, statemachine

Computer Organization

- CPU - ALU (accumulator), Control; von Neumann architecture; Accumulator machine, Register Machine, Stack Machine.
  - Registers---Accumulator(ACC), Program Counter(PC), Instruction Register(IR), Condition Code Register(CC)
  - Fetch Execute Cycle--Fetch, Increment, Execute
  - Instruction Set--Data Movement, Logic and Arithmetic, Control
- Memory
  primary
  linear array
  secondary
  disk, tape ...
- I/O Devices

Operating System:

- resource manager -- processes, memory, etc; DOS, Windows, MacIntosh, OS-2 Warp, UNIX, ...

Application Programs:

- Editor, WordProcessor, Assembler, Compiler, Pascal, FORTRAN, COBOL, LISP, C++, ...

Data

Memory Organization
bits, bytes, words, address
Integers
two's complement

positive(n)
    sign + mantissa
negative (-n)
    complement n and add 1
Overflow
Floating Point
    sign, exponent; round-off error, underflow
Booleans
    Truth values
Character
    ASCII, EBCDIC

Instruction Processing

Assembly language -- Data & Code
    variables, labels, code and translation to machine code (data addresses, Instructions with data addresses, Addresses for instructions, Addresses for labels.
fetch-execute cycle
    Instructions: LOAD A, MULT B, STORE C etc
Assembler
Compiler
Interpreter
Itty-Bitty-Machine

C++ Programs

Printing Text (pattern: *program structure*; pattern: *print string*)

Escape characters in strings: \n \t \r \a \\ \"

Simple Arithmetic (pattern: *prompt-input*; pattern: *label output*; pattern: *print expression*)

Identifiers: letter followed by zero or more letters or digits

WARNING: *C++ is case sensitive*

Data types: int, float, char
Arithmetic operators: +, -, *, /, %
CAUTION: integer division -- 5/3 = 1, 5 % 3 = 2, 3/5 = 0, 3 % 5 = 3
Precedence and parentheses: ( ), * / % left to right, + - left to right
Assignment: identifier = expression

SYNTAX

expression ::= *literal* | *variable* | *expression op expression* | **(** *expression* **)**
literal ::= *integerLiteral* | *realLiteral* | *stringLiteral* | *characterLiteral*

EXAMPLE

```
// Add two numbers using assignment
#include <iostream.h>

int main()
{
   int firstInteger, secondInteger, sum;

   cout << "Please enter the first integer\n";    // PROMPT
   cin >> firstInteger;                           // INPUT

   cout << "Please enter the second integer\n"; //PROMPT
   cin >> secondInteger;                          //INPUT

   sum = firstInteger + secondInteger;

   // LABEL OUTPUT; PRINT EXPRESSION
   cout<< "The sum of the integers is " << sum << endl;

   return 0;  // to indicate that the program ended successfully
}
```

Example Programs

- ThiS is stupid - functions
- Add 2 with functions

Decisions: if (*logicalExpression*) *statement*

Relational operators: ==, !=, <, <=, >, >=, (? :)

Example: (grade >= 60 ? "Passed" : "Failed")

Microsoft's Visual C++

Preliminaries

- Copy the header as header.txt

Accessing Microsoft's Visual C++ on the Network

1. Log into the network
2. Double click on the applications folder -- yellow, labeled **APP**
3. Double click on the **Class Apps** folder
4. Double click on icon labeled **MS Visual C++ ...**

Using Microsoft's Visual C++

Steps	Comments
Create a workspace	
1. Click on **F**ile in the left corner of the top menu bar. 2. Click on **N**ew. . . 3. Click on the **Workspaces** tab. 4. In the **Location** window enter: ***P:\cptr141*** 5. In the **Workspace Name** window enter: ***asn1, lab1,*** etc 6. Click **OK**	The location should be a directory. *In CPTR141, the directory should be dedicated to the course and there should be a workspace for each assignment. An assignment may consist of one or more programming assignments.*

Create a project

1. Click on **F**ile and **N**ew again.
2. Click on the **Projects** tab
3. Click on the button for **Add to current workspace**
4. Click on **Win32 Console Application** in the window.
5. In the **Project Name** window enter: ***proj1*** for the first project name etc.
6. Click **OK**
7. In popup window, select **An Empty Project,** then **Finish**, then **OK**

A workspace may contain one or more projects.
In CPTR141, a project corresponds to a single programming assignment.

Create a file (Files tab)

1. Click **File** and **New** again
2. Click on the **Files** tab
3. Under the **Files** tab select **C++ Source File** enable **Add to project** checkbox.
4. Under **File Name** window, enter the name for the file that will hold your C++ sources code: ***main, file1, file2, ...***
5. Click **OK**

A project may consist of one or more files.
In CPTR141 a project consists of a single file and the file name should be the same as the project name.

Edit the file

1. Click on the source window
2. Insert the standard course header:
    a. Click on **I**nsert in the top tool bar .
    b. Select **insert file as text**
    c. Choose the heading file saved earlier
3. Enter your program
4. When finished, click on **File** and **Save** to save your assignment.

Or use cut and paste from browser.

Compile and execute the program

1. Click on the menu Build then choose Compile filename.cpp or click on the icon
2. run: click on menu Build then choose Run filename.cpp or click on the ! icon.

Submit project for grading

Lab 1: OS, Internet, Editor, Compiler

Background

- IS web page
- IS' *Windows 95/NT Basics*
- CS Department Syllabi, Policies, etc.
- Course Resources
  - heading
  - menu
- WWW

Preparation

- Create a directory for the class (e.g. CPTR141)
- Create subdirectories for your work (e.g. Lab1, ASSN1)

Programming

- Microsoft's Visual C++
- Borland Turbo C++
- GNU g++
- The heading
- Editing
- Compiling
- Error messages
- Edit
- Compile
- Execution
- Output
- Saving
- Printing

Assignment

- Lab - do two programs from the lecture notes
  1. Welcome program
  2. Add two
- Homework: problems 3 and 4 from Chapter 2 page 38.

Hand in the home work on Wednesday.

Lecture 2: Program Development

This lecture takes 2 class periods.

Evolution from an **art form** to a **craft** to an **engineering process**.

HANDOUT

Software Life Cycle

GOAL

- to understand the phases of the software life cycle.
- for simple programs, to understand how to perform analysis, design and implementation using I/O, sequential composition, choice composition and iterative composition.

The Phases of the Software Life Cycle

Phase	Activity	Product
Conceptualization	define the problem	Requirements
Analysis	analyze the problem and desired solution	Specification
Design	design a solution	Architectural design
Implementation	code the solution	Verified and validated code
Maintenance	modify and enhance the code	
Retirement	remove the code from service	

Characteristics of a good problem definition:

PRECISE
COMPLETE
UNDERSTANTABLE

Standard Problem Definition Form:

- TITLE -- Sort three integers
- DESCRIPTION -- Input three integers and output the same three integers sorted from least to greatest
- INPUT -- three integers (see definition below) are to be input, one per line

  Integer: One or more consecutive decimal digits, optionally preceded by a plus sign or a minus sign
- OUTPUT -- The same three integers that are input are output. All three are output on the same line, with a single blank separating adjacent integers. The values are sorted with the smallest on the left.
- ERRORS
  1. Fewer than three integers input will caluse the program to wait for additional input.
  2. Lines of input after the first three are ignored.
  3. If any of the first three lines does not contain a single integer then the program terminates

with the following message

```
INVALID INPUT - only a single integer per line allowed.
```

- EXAMPLE

```
input --> 2
          -3
          +17
output --> -3 2 +17
```

Designing a solution requires
- Programming proficiency
- Problem solving talent with established design techniques
- Experience
  - Understand the problem
  - Understand the options
- Interpersonal skills

Reasoning Models---Deductive(top-down design), Inductive(bottom-up design)
Result---An Algorithm that is

Precise
Complete
Correct with respect to the specification

Documentation
- Internal documentation -- in the code
- Tutorial manual -- user guide
- Reference manual -- programmer's guide

Examples

Requirements, Analysis -- specification, Design, Implementation
- Sequence: Area & Circumference; Calculating revenue
- Choice: Quadratic formula; Pollution index
- Iteration: Average; Mean time to failure
- Abstraction: Stick-person, Menu driven computing

Lecture 3: Structured Programming in C++

This lecture takes 3 class periods.

GOALS: to understand and be able to use

- Top-down design and stepwise refinement
- the structure of C++ programs
- the simple types, integer, real, char, string
- declarations
- arithmetic and logical expressions,
- the assignment statement
- input and output
- basic control structures
- simple file I/O

Introduction

Logic Programming: Facts, rules and proofs.

- Knowledge + Control = Algorithm
- Inference: resolution and unification

Functional Programming:

- f: domain -> range
- function application
- function composition
- higher-order functions

Imperative Programming: Sequence of state changes (assignment statements; x := e)

- Data Types + Algorithms = Program
- Data Types: character, integer, float, array, file, object, ...
- **Algorithms**

A *procedure* for solving a problem in terms of

        - the *actions* to be executed, and
        - the *order* in which the actions are to be executed

    - is called an *algorithm. Algorithm*: sequence of commands

- Basic Commands:

    - assignment (variable := expression),
    - I/O (read, write),

- Unstructured commands:

    - label (*Label***:**),
    - choice (if *relationalExpression* then goto *Label)*,
    - branch (goto *Label*)

- Structured commands:

    - sequence (S; S)
    - choice: **if** *logicalExpression* **then** *statement* **else** *statement*
    - iteration: **while** *logicalExpression* **do** *statement*
    - parallel ...

Object-Oriented Programming

- Objects + Finite State Control = Program

Program Patterns

- Interactive: *Prompt Input Response*; *Menu-choice-do-choice*
- Filter: *Standard input, process, standard output*

Top-down design with stepwise refinement

Simple Types and Literals

Type	Constants	
`bool`	`false true`	
`char`	`'`*letter*`'`	
`int`	*sequence of digits*	
`float`	*digits* . *digits*	
`double`		greater magnitude and precision

string **"***string***"**

Constant declaration: **const** *type var* **=** *value*

Variable declaration: *type var, var, ... ;*
Variable declaration and initialization: *type var* **=** *expression;*

Operators

Arithmetic Operators	**+, -, *, /, %**
Relational Operators	**==, !=, <, <=, >, >=**
Logical Operators	**&&** - *and,* **||** - *or*
Conditional expression	**(** *rel-exp* **?** *exp1* **:** *exp2* **)**
Type casting	**static_cast<** *type* **>(** *variable* **)**

C++ syntax - statements

statement ::= **;** // skip, null, or empty statement (neither has a value nor changes any thing)
| *identifier* **:=** *expression***;** // assignment
| *variable op***=** *expression***;** // *variable = variable operator expression*
| **++** *variable***;** // pre increment; may be used as an expression
| *variable* **++;** // post increment; may be used as statements
| **--***variable***;** // pre and post decrement; may be used as statements
| *variable***--;** // pre and post decrement; may be used as statements
| **{** *statement_1 statement_2 ...* **}** // sequence
| **if (** *relationalExpression* **)** *statement* // choice
| **if (** *relationalExpression* **)** *statement_1* **else** *statement_2* // choice
| **while (** *relationalExpression* **)** *statement*

Input/Output: Communication with the user and/or secondary storage

- Standard I/O (computer-user interaction) requires **iostream.h**
  - **cin** >> *inputVariables* **;** // input
  - **cout** << *outputExpressions* **;** // output
- Pattern: Filter - *Standard input, process, standard output*
  - I/O Redirection: OS-Prompt> *program* < *inputFile* > *outputFile*
  - Pipe: OS-Prompt> cat *inputFile* | *program_1* | *program_2* > *outputFile*
- File I/O (computer-secondary storage communication) requires **iostream.h**, **fstream.h**, **iomanip.h**, **stdlib.hofstream**

  - The following in an output statement affects the formatting of following

expressions

- **<< setprecision (*N*) <<** - N is number of digits to the right of the decimal point
- **<< setiosflags(ios::fixed | ios::showpoint <<** - forces fixed point display and decimal point even in case of integer expression
- **<< setw(*N*) <<** sets field width

- *file* **(** *fileName***, ios::out);** // open external file *fileName* for output calling it *file*
- *file* << *outputExpressions* **;** // output to *file*
- **ofstream** *file* **(** *fileName***, ios::in);** // open external file *fileName* for input calling it *file*
- *file* >> *inputVariables* **;** // input from *file*
- **!** *file* // expression is true at end of file
- **while (** *file* >> *inputVariables* **)** *processInput*

Programming semantics - statements

Hoare triples

{*P-pre-condition*} *statement* {*Q-post-condition*}

The **null statement**

// whatever is true before the null statement
; // is true after the null statement

Software engineering: How shall we use the null statement?

While in some languages the null statement is an integral part of the language, it can be problematic for beginning programmers when the body of a loop is the null statement leading either to an infinite loop or to an unintended number of executions of a disired set of statements.

The **assignment statement**

// P[*expression*/*variable*]
*variable := expression***;**
// P

Software engineering: How shall we use the assignment statement?

```
pi = 3.14159;       // Named constant
radius = 5.4;       // Assign to an independent variable
C = 2*pi*radius;    // Assign to a dependent variable
i = 0;              // initialize a counter
i = i+1;            // increment a counter
cin >> x;           // remove an item from the input stream and assign it to a variable
cout << exp;        // add a value to the output stream
```

Statement sequence

```
// if {P} statement_1 {R} and {R} statement_2 {Q} are Hoare triples, then
// P
statement_1 statement_2 // Q
```

Software engineering: How shall we use the statement sequence?

As most computers are sequential machines, sequential execution of statements is natural. It is important to realize that even though when some statements need not be executed sequentially, they must be written in an arbitrary sequence.

The **While statement**

```
//There must be a progress expression PE which decreases toward zero on each iteration of the loop
//There must be an invariant Inv based on the goal or purpose of the loop
//if {Inv and C=PE>0} body {I and C>PE>=0} for some C is a Hoare triple, then
// Inv and PE >=0
while (condition) {
  // Inv and C = PE > 0 and condition
  body
  // Inv and C > PE >= 0
}
// Inv but not condition.
```

Software engineering: How shall we use the while statement?

The while statement is used when the same action must be performed on a sequence of several items usually where the number of items is unknown -- e.g. reading and processing items from a file.

Algorithms/Patterns

- Counter controlled loop

  counter = *initialValue*; // counter must be initialized
  while (*counter* < *limit***) {**
  actions
  counter = *counter* + 1; // counter must be incremented -- alternate form: *counter*++
  }
- Sentinel-controlled repetition

  initialize *loopVariable*; // Loop variable must be initialized
  while (*loopVariable* != *sentinalValue* **) {**
  actions // and environment must guarantee that loop variable reaches sentinel value
  }
- Nesting control structures

Additional control structures and statements

statements ::= ...
| *variable op= expression***;** // *variable = variable operator expression***;**
| **for (** *expression***;** *expression***;** *expression***)** *statement*
Example:

```
for (int i = initial_value;   // initialization
        i  relOp limit;         // continuation condition
        i++)                    // increment/i-- decrement
        body
```

statements ::= ...
| **switch (** *expression* **) {**
case *value* **:** *actions*
...
}
| **do {** *statements* **} while (** *condition* **);**

- do-while
- break and continue

Programming Pattern -- Menu driven programs

loop Menu-choice-do-choice while choice is not quit

```
do {
        display menu;
        get choice
        switch (choice) {
          case value : action;
```

```
            ...
        }
} while ( condition );
```

Lab 2: Program Development I

Software Engineering

- Stubs
- Menu Driven Programming

Assignment

Remember to use the standard heading. Construct a menu driven program with options which implement the following:

1. Construct an input function that can be used for to prompt users for an integer. By varying the string, function can be used for input for other functions.
2. Construct a recursive factorial function. Remember that 0! = 1, 1! = 1, ... n! = n*(n-1)*...*1.
3. Recursive Fibonacci program: The sequence of **Fibonacci numbers** begins with the integers 1, 1, 2, 3, 5, 8, 13, 21, ... where each number after the first two is the sum of the two preceding numbers.
4. Use the ideas in this program to print a table of the first 10 factorials and the first 10 Fibonacci numbers

Extra credit:

- Find a way to allow users to try all choices without exiting the program.

Hand in your assignment by emailing on or before next Thursday.

Lab 3: Basic Control Structures

Assignment: Construct a menu driven program which selects solutions to

1. A program to compute gas mileage. Input should be in pairs of numbers - miles and gallons. The output should be the miles per gallon for the total miles and gallons.
2. A program to read a set of numbers, count them, and calculate and display the mean, variance, and standard deviation of the set of numbers. Use the following formulas:

   mean = $(Sum^{n}_{i=1}x_i) / n$

   variance = $(Sum^{n}_{i=1}x^2_i) / n - (Sum^{n}_{i=1}x_i)^{2} / n^2$

   standard deviation = sqrt()

3. The sequence of **Fibonacci numbers** begins with the integers 1, 1, 2, 3, 5, 8, 13, 21, ... where each number after the first two is the sum of the two preceding numbers. The ratios of consecutive Fibonacci numbers approach the "golden ratio" , (square root of 5 - 1)/2. Write a function to calculate all the Fibonacci numbers smaller than 5000 and the decimal values of the ratios of consecutive Fibonacci numbers. Hint: use two functions, a driver function and an integer valued Fibonacci function written in the style of the factorial function of the previous lab.

Hand in your assignment by emailing it to ... on or before next Thursday

Lecture 4: Modular Design: Functions

This lecture takes 3 class periods.

Motivation

Procedures and functionsallow you to

- provide a *high level structure* for your program, and
- avoid writing the ``same thing'' more than once.

The Standard Library Header Files

The standard library is a collection functions.

<assert.h>	<vector, <list>
<ctype.h>	<deque>, <queue>
<float.h>	<stack>, <map>
<limits.h>	<set>, <bitset>
<math.h> // f: double -> double i.e., use **double** x ...**;**	<functional>
ceil(x)	<memory>
cos(x)	<iterator>
exp(x)	<algorithm>
fabs(x)	<exception>
floor(x)	<stdexcept>
fmod(x, y)	<string>
log(x)	<sstream>
log10(x)	<locale>
pow(x. y)	<limits>
sin(x)	<typeinfo>
sqrt(x)	
tan(x)	
<stdio.h>	
<stdlib.h>	
<string.h>	
<time.h>	
new style headers of the above have the form: <cassert>	
<iostream.h>	
<iomanip.h>	
<fstream.h>	
<utility.h>	

new style headers of the above have the form: <iostream>

C++ syntax - function declaration and call

function-declaration ::=
return-value-type function-name **(** *comma separated parameter-type-list***);** // Function prototype
return-value-type function-name **(** *comma separated parameter-list***) {** *declarations and statements* **}**
expression ::= ...
| *function-name* **(** *comma separated argument-list* **)**
Example:
prototype: int square (int);
declaration: int square (int n) { return n*n }
call: square(3) or square(x+5)

Parameters

There are three types of parameters:

- An **in** parameter is used to pass data into a function.
- An **out** parameter is used by a function to pass data out.
- An **in-out** parameter is used to pass a data structure into a function which may modify the data structure and pass it back out.

Inparameters (sometimes called *value parameters*) are often implemented by creating a *copy*of the argument and passing the copy to the function (this is called *passing by value*or *passing by copy*). This prevents the function from modifying the original values.

In-out parameters are often implemented by passing the address (*reference*) of the argument (this is called *passing by reference*).

- Parameterless functions: *type name* (void)
- value parameters (in): *type name* (*type name*)
- reference parameters (in-out): *type name* (*type* & *name*)

Function overloading

int name(int param)
float name (float param)

Data Types, mixed-type expressions and promotion rules

long double

double
float
unsigned long int -- unsigned long
long int -- long
unsigned int -- unsigned
int
unsigned short int -- unsigned short
short int
unsigned char
short
char

Random Numbers

integer between 0 <= rand() <=RAND_MAX-- standard library
scaling: rand() % n -- 0..n-1
scale &shift: 1+ (rand() % n) -- 1..n
randomizing: srand(seed) -- where seed is unsigned int

Enumerations

enum Status { CONTINUE, WON, LOST}
enum WeekDay {MON=2, TUE, WED, THUR, FRI}

Inline Functions

Global Variables

Global variables are variables declared outside of a function and visible to the function. They may be modified by any function or procedure and are suitable only for small programs where the programmer can be expected to keep track of which functions modify the global variables. For large programs, classes and objects provide a more disciplined method of access to global variables.

int gv;	// global variable

int f (void)
{
  return g++;
}
	// f increments the global variable gv and returns the previous value

Programming semantics - function

return-value-type function-name **(***parameter-list* **)**

// pre-condition: ***user must provide arguments satisfying the pre-condition.***
// post-condition: ***the result and side-effect satisfy the post-condition, if user's arguments meet the pre-condtion.***
{*declarations and statements***}**

EXAMPLE
debug = 1;

```
int fac( int n ) //factorial function
{ // precondition: n > 0
  if (debug) assert( n<0, "factorial function called with negative argument");
  // post condition: result == n!

   if (n==0) return 1;
   else return (n * fac (n-1));
}
```

Software engineering: How shall we use functions?

Functions allow code to be broken up into short understandable sequences -- a *high level structure*for a program. If you write more than a page of code, it probably should be broken into two or more functions.

Functions provide a way to avoid writing the ``same thing'' more than once. Frequently used function are assembled into libraries for even wider use.

Well written functions have a clear purpose -- usually performing one thing.

Software engineering

Design

For both data and algorithms use

- *top-down design* to decompose solution into a collection of data items and functions and use
- *stepwise refinement* (repeating top-down design on each data item and function ... until simple data items and instruction levels are reached).

A *structure chart*may be used to show dependencies between functions -- functions called by other functions. Independent functions do not call other functions.

Implementation and Testing

Avoid the *big bang*. Implement your program in stages using stubs.

- *Stubs*: functions whose bodies are initially empty, print a trace message, or return a token value.

Bottom-up implementation

1. Implement and test independent functions
2. Iteratively implement and test successive levels of dependent functions finishing with the top-level functions.

Test each function with both extreme and expected data values.

Debugging: instrument program with messages to allow tracing of internal program behavior

- Use functions in **<assert.h>** to simplify debugging.

Additional Detail

Functions with value parameters (argument is an expression)
user defined sqr, cube, The Circle program, factorial, fibonacci
Procedures without parameters
A menu driven program

Parameters Formal parameters allow you to write procedures that can operate on different data without being rewritten.

Functions with variable parameters (argument is a variable)
user defined sqr, cube, The Circle program, factorial, fibonacci
Procedure with variable parameters
A modified menu program i.e., combined prompt-read
record type
Complex-Numbers/rational arithmetic package
Scope Rules
and nested procedures

Software Engineering

Global Variables

Global variables may be modified by any function or procedure and are suitable only for small program where the programmer can be expected to keep track. For large programs, classes and objects provide a more disciplined method of access to global variables.

Parameters, Procedures, and Functions

Some "rules of thumb"

Parameters

- Value parameters are used where the subroutine should not modify its arguments.
- Variable (reference) parameters are used where the subroutine may modify its arguments.
- Large data structures are passed by reference to save the expense of making a copy.

Functions

- Functions maybe used wherever an expression is expected.
- Functions return a simple result
- Functions should not have side-effects
- Functions leave their arguments unchanged.

Procedures

- Procedures are used wherever a statement is expected.
- Procedures return multiple values.
- Procedures are used for their side-effects.
- Procedures modify their arguments.

Lab 4: Functions

Assignment: Construct a menu driven program permitting the user to execute solutions to problems

1. 3.27
2. 3.38/39 Guess the Number
3. Implement numerical integration using either, Simpson's rule, the trapezoid rule, the rectangle method, or the Monte Carlo method. While your solution should work for any function, test it with $e^x\cos(x)$ for x between 0 and Pi (3.14159). Your answer should be close to -12.0703463164
4. Towers of Hanoi: Simulate the moving of a stack of disks from one peg to another. The disks are stacked in decreasing size. Move one disk at a time. At no time may a larger disk be placed above a smaller disk. A third peg is available for temporarily holding disks.

Hand in your assignment by emailing on or before next Thursday

Data Types: One Dimensional Arrays

This lecture takes 3 class periods.

1-D Arrays

Array Declaration
element type, index range : **`int a[100];`**
Array Initialization

```
int a[5] = {45, -32, 15, 16, 3};
int a[] = {45, -32, 15, 16, 3}; // implicit size is 5
a[i] = exp
const int MAX = 10; // constant
int a[MAX]; // using a named constant increases flexibility
```

Strings

```
char string[] = "hello";
char string[] = {'h','e','l','l','o','\0'};
```

Arrays as parameters

```
void sort ( int [], int ); // function prototype
void sort ( int data[], int size ){...} // function header/definition
sort( mydata, mydataSize ); // call
```

NOTE: arrays are passed by reference
Example
mean: average value
median: middle score
mode: most frequent score
mean, scores > mean, scores < mean, ordered scores, frequency distribution (histogram), bar graph
Searching
linear (for unsorted or small lists)
binary (for sorted lists)
Sorting
Selection Sort: select the ith element and swap it into place
Insertion Sort: insert x in to an ordered list.
Bubble Sort: swap adjacent elements until list is ordered
Shell Sort:
Quick Sort: recursively partition and merge
Analysis of Algorithms
Linear search: O(n)

Binary search: O(log n) -- base 2
Selection sort: O(n^2)
Quicksort: O(n^2); average O(n log n) -- base 2
Execution Times (1 msec/instruction; 1 msec = 0.000001; n = 256)

```
Function        Time
log log n       0.000003 sec
log n           0.000008 sec
n               0.0025 sec
n log n         0.002 sec
n^2             0.065 sec
n^3             17 sec
2^n             3.7E61 centuries
```

Strings

String

1. Declaration
2. Initialization
3. I/O
4. Comparison
5. N = length(string)
6. S = concat(s1,...,sn)
7. C = copy(s,index,size)
8. P = pos(substring, string)
9. S = insert(item, string, position)
10. S = delete(string, position, size)
11. str(number, string) --- number -> string
12. val(string, number, errorcode) --- string -> number

Applications

form letters, palindromes, encryption (security)

Multi-dimensional arrays

Array Declaration

element type, index ranges : **`int a[100][50];`**

Array Initialization

```
int a[3][5] = {{45, -32, 15, 16, 3}{45, -32, 15, 16, 3}{45, -
32, 15, 16, 3}};
int a[] = {45, -32, 15, 16, 3}; // implicit size is 5
a[i][j] = exp
const int ROWS = 10; // constant
const int COLUMNS = 5; // constant
int a[ROWS][COLUMNS]; // using a named constant increases
```

flexibility

Arrays as parameters

```
void mult( int [][], int [][], int[][] ); // function prototype
void mult( int RowsA, int a[][ColsA],
           int RowsB, int b[][ColsB],
                      int c[][ColsB] ){...} // function
header/definition
mult( 5, X, 3, Y, Z ); // call
```

Arrays: Searching, Sorting, Statistics and Life

Background: <stdlib.h> contains rand() which generates numbers between 0 and RANDMAX

srand(Seed)
srand(time(0)) // use <time.h>
rand()
rand() % 6

Assignment: Construct a menu driven program permitting the user to execute solutions to the problems:

1. A program to read a set of numbers. Sort and display the sorted list. Also calculate and display the mean, and standard deviation of the set of numbers. Use the following formulas:

   mean = $(Sum^{n}_{i=1} x_i) / n$

   standard deviation = $sqrt[(Sum^{n}_{i=1} x^2_i) / n - mean^{2}]$

2. Implement bubble sort with the provision for exit when no adjacent items are exchanged. Your solution should intialize the array with random data. Bubble sort should be implemented as a function. You should have one function which you use to print both the initialized array and the sorted array.
3. Simulate the rolling of dice. You should use **rand()** to roll the first die. and should use **rand()** again for the second die. Then sum the two values. Simulate 36,000 rolls, tabulate the results and compare the result with the expected probabilities.
4. Write a function **testPalindrome** which returns **true** if the string argument is a palindrome and **false** if it is not.
5. Life: Students may work in pairs, submitting one program which contains both of their names. Develop a program to "play" the game of life. Your program should have modules to:
   1. initialize an array to all blanks
   2. initialize an array with data entered from a file (or by the user)
   3. display an array
   4. compute the next generation

   The rules are as follows:
   1. A birth occurs in an unoccupied cell if it has exactly three neigbors.
   2. A death occurs in an occupied cell if it has either less than 2 or more than three

neighbors.

3. An occupied cell survives to the next generation if it has 2 or 3 neighbors.

Hints:

1. If you plan for an mxn array for display, use an (m+2)x(n+2) array internally with blanks around the boarder to simplify the number of neighbors computation.
2. Use two arrays, one for the current generation and one for the next generation.

Hand in your assignment by emailing it on or before next Thursday.

Lecture 5: Text/Data Files

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>

void openFiles(ifstream &, ofstream &);
void closeFiles(ifstream &, ofstream &);

int main()
{
   ifstream sourceFile;
   ofstream copyFile;

   openFiles( sourceFile, copyFile );

   int datum;

   while ( sourceFile >> datum )
      {copyFile << datum*datum << " " ;}

   closeFiles( sourceFile, copyFile ); //ifstream & ofstream
destructors close the file

   return 0;
}


void openFiles(ifstream &inFile, ofstream &outFile)
{
  char string[30];

  cout << "Enter path to input file: ";
  cin >> string ;

  inFile.open(string, ios::in);
  // ifstream inFile( "old.dat", ios::in ); // an alternate method

  if ( !inFile ) {
     cerr << "File could not be opened\n";
     exit( 1 );
  }
```

```
  cout << "Enter path to output file: ";
  cin >> string ;

  outFile.open(string, ios::out);
  // ofstream outFile( "new.dat", ios::out ); // an alternate method

   if ( !outFile ) {  // overloaded ! operator
      cerr << "File could not be opened" << endl;
      exit( 1 );    // prototype in stdlib.h
   }

}

void closeFiles(ifstream &inFile, ofstream &outFile)
{
  inFile.close();
  outFile.close();
}
```

Lab 5: Files & other things

File Processing Problems

- Copy a file
- Count vowels/specified characters entered at runtime
- Count chars per line, displays length of shortest, longest and average
- deletes blank lines and leading blank chars
- count nonblank chars, nonblank lines, words, sentences.
- Transaction processing: master file, transaction file, newmaster file.
- range, mean, and standard deviation of a data set

Assignment

Using the example programas a model, develop a menu-driven program, with procedures and functions, that allows a user to enter a file names and choose to do the following:

1. Modify your statistics program to read its data from a file.
2. Write a program to copy a text file into another text file in which the lines are numbered with the line number at the left of each line.
3. Write a program that plays the game of "guess the number". It should be able to play either side -- picking a random number in the range of 1 to 1000 and responding appropriately to user's guesses or guessing a user chosen number in the range of 1 to 1000.
4. Implement numerical integration using either, Simpson's rule, the trapezoid rule, the rectangle method, or the Monte Carlo method. While your solution should work for any function, test it with $e^x\cos(x)$ for x between 0 and Pi (3.14159). Your answer should be close to -12.0703463164

Hand in your assignment by emailing it on or before next Thursday.

Lecture: Strings and Pointers

Chapter 5
Chapter 11
Chapter 16

Pointers

```
int y = 5;
int *yPtr; // yPtr stores addresses rather than values

yPtr = & y; // yPtr has address of y
```

Dereferencing

```
cout << *yPtr ... // outputs the value pointed to by the address in
yPtr
cin >> *yPtr ... // inputs a value and stores it at the address in
yPtr
```

char *strngPtr;

Functions as parameters:

```
float f(float);
float g(float);

float trap( float a, float b, int n, float (*f) (float) )
{
        float result = 0;
        float deltaX;

        deltaX = (b-a) / n;

        for (int i=1; i<n; i++)
        {
            result += f(a+i*deltaX);
```

```
          }

        return  deltaX * ( (f(a) + f(b))/2) + result );
}

void main()
{
       ... trap(0, 3.14159, 256, f) ...
}
```

Strings p. 325

```
null character '\0'

char color[] = "blue";
char color[] = {'b', 'l', 'u', 'e', '\0'}; // note explicit null
character

char word [10];

cin >> word ... // up to space, tab, newline,  or end-of-file
cin >> setw(10) >> word ... // to insure no buffer overflow --
SECURITY

char sentence [80];

cin.getline( sentence, 80, '\n');  // see chapter 11 stream i/io
cin.getline( sentence, 80);  // '\n' by default

#include <string.h>
```

*strcopy
*strncpy
*strcat
*strncat
strcmp
strncmp
*strtok -
strlen

Arrays of pointers

char *mcode[36];

char msym[6];

C++ Stream I/O

```
cout << "\n";  // newline character and flush output buffer
cout << endl; // "
cout << flush; // flushes output buffer, no newline
```

Note: if cout is followed by a cin, the output buffer is flushed

```
cout.put('A');
cout.put(65); // ascii code for A; outputs an A
```

>> skips whitespace

cin.eof() -- true or false depending on file state

cin.get() -- returns a character (including whitespace) and returns EOF on end-of-file
cin.get(Ch) -- returns a character (including whitespace) and returns 0 on end-of-file
cin.get(CharArray, SizeLimit, DelimiterChar) -- default is '\n'; does not consume delimiter

cin.getline(CharArray, SizeLimit, Delimiters) -- discards delimiter

cin.peek()
cin.putback()
cin.ignore()

cin.read(STRING,N)
cout.write(STRING,N)

Lab : Text processing

Assignment

Do 5.47 (Morse Code) permitting the user to enter the path to the input file and to the output file.

Hand in your assignment by emailing it to Todd Graham grahth@wwc.edu on or before next Thursday.
Need some help? Look at this code.

Classes & Objects - Data Abstraction

Introduction

```
struct Time {
   int hour;
   int minute;
   int second;
};

Time start, finish;

cin >> start.hour >> start.minute >> start.second;

class Time {
public:
  Time();
   void setTime( int, int, int)
   void printMilitary();
   void printStandars();
private:
   int hour;
   int minute;
   int second;
};

Time::Time() {hour = minute = second = 0;}
void Time::setTime(int h, int m, int s)
{
   hour = h;
   minute = m;
   second = s;
}



void main()
{
   Time t;
    ...
     t. setTime(5,36,7);
```

```
    ...

}
```

Lab : Text processing

Assignment

Implement the temperature conversion class discussed in the class.

Develop a program which maintains an address book. The program must have at least one class for the address book. It should include functions to:

1. Create an address book file
2. Read an address book file into an array
3. Display the contents of the address book
4. Modify an address book entry
5. Add an entry to the address book
6. Delete an entry from the address book
7. Sort the address book
8. Save the address book in a file.

You may work in pairs, submitting one program which contains both of your names.
Hand in your assignment by emailing it to Todd Graham grahth@wwc.edu on or before the final exam.
Need a hint? code sample

Lecture 6: Simulation and Numerical Computation

This lecture takes 3 class periods.

Simulation

modeling a dynamic process and using the model to study the behavior of the process. Used by Busness (economic models), Environmental science (environmental change), Computer Science (operating system performance)

- Deterministic processes: modeled using a set of equations. For example, population growth and atomic decay, fight trajectory.
- Nondeterministic processes (random): modeled using a random number generator (numbers are distributed uniformly over a fixed range). For example, Brownian motion, arrival of airplanes, number of defective parts etc.

Random numbers

uniform distribution; normal distribution

```
0.0 <= random < 1.0
0 <= random(N) < N
```

problems are found on pages 364-366

Numeric Computation

- Curve fitting: least squares
- Solving equations: bisection, newton
- Integration: rectangle method, trapezoid, simpson
- Differential equations
- Solving linear systems

problems: pages 378-383

Recursion

- Fibonacii
- Factorical
- Reverse
- Towers of Hanoi
- Maize traversal

Lab 6: Simulation and Numeric Computation

Least Squares

Finds the least squares line for data pairs in a file. See pages 370 and 862 the files are found in K:\cptr\141\text and are fig7-9.pas and leastsqu.fil

Simulation

do one of 16-22 on pages 364-366

Numeric Computation

do one of 2-10 on pages 378-383

Hand in

copy your programs to the directory K:\CPTR\141\ASN6. The file names of your programs should be your user name followed by an S for the simulation and a N for the numeric program followed by the .pas suffix (for example, aabyanS.pas and aabyanN.pas).

Lecture 7: Data Types: Sets and Ordinals

This lecture takes 3 class periods.

Ordinal Data type

integer, boolean, char -- compare, succ, pred, inc, dec, typecast

- Enumerated Data Type
- Subrange

Sets

uniform distribution; normal distribution

Parser

```
program Parser (input,output);

{       A grammar for a Pascal like language
        ( Upper case symbols are nonterminals, lower case symbols and
          punctuation symbols are terminals.)

        P --> p i ; D B .           Program
        D --> v i : t ; D           Declaration
        D --> empty
        B --> b SS e                Body
        SS --> S RS                 Statement Sequence
        RS --> ; SS
        RS --> empty
        S --> i := E                Statement

        E --> T Elist               Expression
        Elist --> + T Elist
        Elist --> empty
        T --> F Tlist               Term
        Tlist --> * F Tlist
        Tlist --> empty
        F --> ( E )                 Factor
        F --> i
        F --> c
}
const
   debug = true;
type
              {A token type for each non-terminal in the grammar}
   tokenT = (progsym, vardec, typesym,
             beginsym, endsym,
             constsym, ident,
             assignop,
             plus, times, lparen, rparen,
             colon, semicolon, period,
             eofsym, errorsym);

var
```

```
   ch : char;
   token : tokenT;

procedure getch; { returns the next character except at eof in which
                   case returns a blank }
  begin
    if eof then ch := ' '
    else if eoln then begin readln; getch end
    else read( ch );
  end;


procedure gettoken;   {The Scanner: returns the token types}
{ FINISH THIS -- SET token TO TOKEN TYPE }

   begin
      while (ch = ' ') and not eof do getch;
      if debug then writeln(ch);
      if (ch = ' ') and eof then token := eofsym
      else      {recognize each token}
         case ch of
           'p' : begin token := progsym;   getch end;
           'i' : begin token := ident;     getch end;
           ';' : begin token := semicolon; getch end;
           '.' : begin token := period;    getch end;
           ':' : begin
                    getch;
                    if ch = '=' then begin token := assignop; getch end
                    else token := colon
                 end
            else token := errorsym
         end
   end;

{ FINISH ADDING FORWARD REFERENCES FOR EACH NONTERMINAL }

procedure P; forward;
procedure D; forward;
procedure B; forward;
procedure SS; forward;
procedure S; forward;
procedure RS; forward;
procedure E; forward;

procedure Error( s:string; t : tokenT );    { Error handler }
  begin
     write('Expected a ', s, ' found a ');
     case t of
       progsym  : write( 'program' );
       vardec   : write( 'variable declaration' );
       typesym  : write('type symbol');
       constsym : write('constant');
```

```
        ident    : write('identifier');
        assignop : write('assignment operator');
        plus     : write('plus operator');
        times    : write('times operator');
        lparen   : write('left parentheses');
        rparen   : write('right parentheses');
        colon    : write('colon');
        semicolon: write('semicolon');
        period   : write('period');
        eofsym   : write('end of file');
        errorsym : write('indeciferable symbol')
      end;
      writeln(' instead')
  end;

{  FINISH BY ADDING THE REMAINING PROCEDURES }

procedure P;
  begin
    if token = progsym   then begin gettoken;
    if token = ident     then begin gettoken;
    if token = semicolon then begin gettoken;
               D;
               B;
    if token = period    then       gettoken
    else      error('period',token)
    end else error('semicolon', token)
    end else error('identifier',token)
    end else error('program symbol',token)
  end;

procedure D;
  begin
    if token = vardec    then begin gettoken;
    if token = ident     then begin gettoken;
    if token = colon     then begin gettoken;
    if token = typesym   then begin gettoken;
    if token = semicolon then begin gettoken;
               D
    end else error('semicolon',    token)
    end else error('type symbol',  token)
    end else error('colon',        token)
    end else error('identifier',   token)
    end else error('variable sym', token)
  end;

procedure B;
  begin
    if token = beginsym then begin gettoken;
               SS;
    if token = endsym   then       gettoken
        else error('end symbol',   token)
```

```
    end else error('begin symbol', token)
  end;

procedure SS;
  begin
    S; RS
  end;

procedure S;
  begin
    if token = ident    then begin gettoken;
    if token = assignop then begin gettoken;
               E
    end else error('assignment operator', token)
    end else error('identifier',          token)
  end;

procedure RS;
  begin
    if token = semicolon then begin gettoken;
               SS
    end
  end;

procedure E;
   begin
   end;

begin
   ch := ' ';    { One character look ahead }
   getch;
   gettoken;     { One token look ahead     }
   P;
   if token = eofsym then writeln('Proper Syntax')
   else error('end of file', token)
end.
```

Lecture 9: Data Types: Multi-Dimensional Arrays

This lecture takes 2 class periods.

Array Declaration
    index type, element type
Array Initialization
Example
    Cellular automata, graphics, matrix multiplication, gaussian elimination, transpose, graph theory, finite state machines.

Classes and Objects

Assignment

Develop a program which maintains an address book. The address book should be stored in a file.

1. Insert an item into the l
2. Display the data in the .
3. Delete an item from
4. Find an item in the
5. Write data from the file. You will need to define a file of the appropriate record type. You can then write an entire record to the file with a single write statement.
6. Read data from a file into .
7. Modify an item in the

Students may work in pairs, submitting one program which contains both of their names.

Hand in

copy your program to the directory K:\CPTR\141\ASN10. The file name of your program should be your user name followed by the .pas suffix (for example, aabyan.pas).

Lecture 10: Data Types: Records and Pointers

This lecture takes 2 class periods.

Record Declaration
    Fields
Record Initialization
Example
    Address book.
Pointer Declaration
Linked List
Example

```
program Links (input, output);

type
     ItemT = integer;

     List = ^Node;
     Node = record
                Head : ItemT;
                Tail : List
            end;
var
    Numbers, L1 : List;

{  may be  initialized with a declaration of the form:
           const <var> : <type> = <value>
}


procedure empty (  var L : List );    begin
       L := NIL;
   end;

procedure cons( Item : ItemT; var L : list );
   var I : List;

   begin
      new(I);              {create a cell to hold the item}
      I^.Head := Item;     {copy item into cell           }
      I^.Tail := L;        {set cell to point to list     }
      L := I               {reset list to point to cell   }
```

```
  end;

procedure display( L : List);
   begin
      if L <> Nil then begin writeln(L^.Head); display(L^.Tail) end
   end;

procedure insert( Item : itemT; var L : List );
   var H : itemT; I, T : List;
   begin
      if L = nil then cons( Item, L)
      else begin
         if Item < L^.Head then cons(Item, L)
         else insert( Item, L^.Tail )
      end
   end;

procedure append( var L1, L2 : List );
   begin
      if L1 = Nil then  L1 := L2
      else append( L1^.Tail, L2 )
   end;

procedure find( Item : itemT; var L : List );
   begin
      if L = Nil then writeln(Item, ' is not in the list')
      else if L^.Head = Item then writeln(Item, ' is in the list' )
      else find( Item, L^.tail )
   end;

begin
   empty(Numbers);
   insert(3,Numbers);
   insert(5,Numbers);
   insert(1,Numbers);
   insert(4,Numbers);
   display(Numbers);
   readln;
   empty( L1 ); insert( 9, L1 ); insert( 8, L1 ); display( L1 );
   readln;
   append(Numbers, L1); display(Numbers);
   readln;
   find( 10, Numbers );
   find(  5, Numbers );
   readln
end.
```

Lab 9: Records and Linked Lists

Assignment

Develop a program which maintains an address book. You should modify the code from the program links. You will need to define a **record** data type to contain your data items. Your program should have modules to:

1. Insert an item into the linked list.
2. Display the data in the linked list.
3. Delete an item from the linked list.
4. Find an item in the linked list.
5. Write data from the linked list to a file. You will need to define a file of the appropriate record type. You can then write an entire record to the file with a single write statement.
6. Read data from a file into the linked list.
7. Modify an item in the linked list.

Students may work in pairs, submitting one program which contains both of their names.

Hand in

copy your program to the directory K:\CPTR\141\ASN10. The file name of your program should be your user name followed by the .pas suffix (for example, aabyan.pas).

An Overview of Pascal

Notational conventions for Pascal programs or code fragments:

- Actual Pascal code is presented in `typewriter font`.
- Comments or user defined code is presented in *`italicized typewriter font`*.

Programs

- Program structure

```
PROGRAM program name ( external files );

Constant definitions
Type definitions
Variable declarations
Procedure and Function definitions

BEGIN

   Statements

END.
```

- Names must be defined or declared before they are referenced.
- Comments may be placed anywhere a blank may appear and are of the form

```
{ This is a comment.  It includes the opening
  and closing braces and may extend over several
  lines.  Comments may not be nested i.e. braces
  not be nested.
}
```

Constants

The constant definitions section has the form:

```
CONST
   constant name = literal;
   ...
   constant name = literal;
```

Literals

Literals include

- Integer: `345`
- Real: `3.14159`
- String: `'This is a string constant'`
- Boolean: `true` and `false`

Types

The type definitions section has the form:

```
TYPE
   type name = type expression;
   ...
   type name = type expression;
```

The predefined types include

- `char` -- character
- `integer` -- integers
- `real` -- a subset of the rationals
- `boolean` --
- `text`

and the user defined types are built from

- `array`
- `record`
- `file of`
- sets `set`
- subranges
- enumerations

Type names are identifiers.

Files

The names: `Input, Output` are part of the environment. They denote files (stdin, stdout) whose values are part of the state. Type: name, set of constants, set of operations

assign(file-variable, file-name);
reset(file-variable);

read(file-variable, variable list)

readln(file-variable)
readln(file-variable, variable list)

assign(file-variable, file-name);
rewrite(file-variable);

write(file-variable)
write(file-variable, variable list);
writeln(file-variable)
writeln(file-variable, variable list);

close(file-variable);

Integer

Integers have the form: [+|-]D^+

Real

Real numbers have the form: [+|-]D^+.D^+E[+|-]D*

Other scalar types

Expressions

char, boolean, enumerated, subrange operations: +, -, $*$, /, div, mod mixed types \& overloading type conversion: round, trunc, ord, chr

Variables

The variable declaration section has the form:

```
VAR
   variable name = type name;
   ...
   constant name = literal;
```

Procedures and Functions

A procedure definition has the form:

```
procedure Name ( Foramal Parameters );
{ Pre: a comment reguarding the conditions for use
  Post: a comment reguarding the results returned
```

```
}
Constant definitions
Type definitions
Variable declarations
Procedure and Function definitions

Begin
   Statements
End;
```

The parameters and definitions introduce an local to the procedure.

User defined procedures are used just like statements. They may appear in a statement sequence as follows:

```
...
Name ( Actual Parameters );
...
```

If the procedure is defined without formal parameters, then the parentheses are not used in either the definition or the reference.

A function definition has the form

```
function Name ( Foramal Parameters ) : Type;
{ Pre: a comment reguarding the conditions for use
  Post: a comment reguarding the result returned
}
Constant definitions
Type definitions
Variable declarations
Procedure and Function definitions

Begin
   Statements { which must include
     an assignment statement of the form

         Name := expression

     where Name is the name of the function.
   }

End;
```

The parameters and definitions introduce an local to the function.

User defined functions are used just like Pascal's built-in functions. They may appear in an expression as follows:

```
... Name ( Actual Parameters ) ...
```

If the function is defined without formal parameters, then the parentheses are not used in either the definition or the reference.

Formal Parameters

If a procedure or function does not have any parameters, then the parentheses must not be used.

The formal parameters are a **semicolon** separated list of elements of the forms:

```
comma separated list of names : type
var comma separated list of names : type
```

The first type of parameter is value parameter. Any assignment to the parameter in the body of the procedure or function is local in effect.

The second type of parameter is a variable parameter. Any assignment in the body of the procedure or funtion is global in effect.

Actual Parameters

If a procedure or function is defined without formal parameters, then the parentheses are not used.

The actual parameters are a **comma** separated list of variables and expressions. They must correspond in order and type to the formal parameters.

Actual parameters corresponding to value parameters may be expressions. Actual parameters corresponding to variable parameters must be variables.

If the actual parameter corresponds to a value parameter then any assignment to the parameter in the body of the procedure or function is local in effect.

If the actual parameter corresponds to a variable parameter then any assignment in the body of the procedure or function changes the assignment of the actual parameter.

Statements

The statements include

- assignment statement
- output statement
- input statement
- for statement

- while statement
- repeat statement
- compound statement
- if statement
- case statement
- procedure call

assignment statement

The assignment statement has the form:

```
identifer := expression
```

The *identifier* is assigned the value of the expression. The identifier and expression must be type compatible (matching types).

output statement

The output statement is of the forms

```
writeln
write( comma separated list of expressions )
writeln( comma separated list of expressions )

write( file variable, comma separated list of expressions )
writeln( file variable )
writeln( file variable, comma separated list of expressions )
```

The expressions must be of type: integer, real, char, or string. The arguments are evaluated in order from left to right and the values are appended to Output.

input statement

The input statement is of the forms:

```
read( comma separated list of variables )
readln
readln( comma separated list of variables )

read( file variable, comma separated list of variables )
readln( file variable )
readln( file variable, comma separated list of variables )
```

The variables must be of type: character, integer or real. They are assigned to values in sequence from Input. The values are removed from Input. For character input, one character is input and assigned (blanks are characters and end of lines are read as blanks). For numeric input, all leading blanks and end of lines are

consumed and the longest string that has the correct syntax is used as the input.

for statement

The for statement comes in two forms:

```
for index := low to high do
   statement

for index := high downto low do
   statement
```

index, *low* and *high* must be of the same enumerated type. *index* must be the name of a variable, *low* and *high* must be expressions.

while statement

The while statement has the form:

```
while condition do

   statement
```

and the *statement* in the body of the while statement is repeatedly executed while the *condition* is true. The *condition* is checked upon entry to the while statement and after the execution of the body.

repeat statement

The repeat statement has the form:

```
repeat

   statement0;
   statement1;
   ...
   statementn

until condition
```

and each statement in the body of the repeat statement is executed in the order it appears in the sequence. The entire sequence is repeated until the *condition* is true. The *condition* is checked only after the execution of the last statement in the sequence.

compound statement

The compound statement has the form:

```
begin
   statement0;
   statement1;
   ...
   statementn

end
```

and each statement is executed in the order it appears in the sequence.

if statement

The if statement has the forms:

```
if condition then
   statement

if condition then
   statement
else
   statement
```

case statement

The case statement is of the form:

```
case expression of
   label list0   : statement0;
   label list1   : statement1;
   ...
   label listn-1 : statementn-1
   else            statementn

end
```

The label lists are comma separated lists of constants of an enumerated type. The else clause is optional.

Names i.e. Identifiers

Pascal's reserved words and user defined names are identifiers which are sequences of alphabetic charactors and digits. The first charactor must be an alphabetic character. Case does not matter so the following are the

same

```
HELLO123THERE Hello123There
```

Sample Programs

```
PROGRAM Circle( input, output );

CONST  pi : 3.1415;

VAR Radius : integer;
Circumference : real;

BEGIN

   Writeln ( 'Please enter the radius of the circle.' );
   Readln ( Radius );
   Circumference := 2*pi*Radius;
   Writeln ( 'The circumference of the circle is: ', Circumference )
   Writeln ( 'The area of the circle is: ', pi*Radius*Radius );

END.

PROGRAM ReverseFourChars ( Input, Output );

VAR  First, Second, Third, Fourth : char;

BEGIN

   Writeln( 'Please enter four characters.' );
   Readln( First, Second, Third, Fourth );
   Writeln;
   Write( 'Your four characters in reverse order are: ' );
   Writeln( Fourth, Third, Second, First )

END.
```

Model Pascal Program from *Condensed Pascal* by Cooper

```
program SoTypical (input, output);   {heading}

const
   LIMIT = 10;           {integer constant}
   POUNDSIGN = '#';      {char constant}
   AMORCITA = 'llana';   {string constant}

type
   Hues =(Red, Blue, Green, Orange, Violet);  {enumerated ordinal type}

   Shades = Blue. . Orange;                   {subrange}
   SmallNumbers = 1..10;                      {subrange}

   String = packed array [l..LIMIT] of char;  {string}

   Class = record                             {record-type}
              Name: String;
              Units: integer;
              Grade: char
           end;

   Grades = array [SmallNumbers] of Class;    {array type}
   ColorCount = array[1..10, 'A'..Z'] of Hues;{array type}

   ClassFile = file of Class;                 {file type}

   Pastels = set of Shades;                   {set type}

   NextWord = ^Sentence;                      {pointer}
   Sentence = record                          {dynamically allocable record}
                 Current Word: String;
                 Coming Word: NextWord
              end;


var High Low, Counter: integer;   {integer}
    First, Last: char;            {char}
    Heights, Weight: real;        {real}
    Testing, DeBugging: boolean;  {boolean}
    Colors: Hues;                 {enumerated type}

    Shorts: Small Numbers;        {subrange}
    Name: String;                 {string}

    OneCourse: Class;             {record}
```

```
    Curriculum: Grades;           {array}
    ColorSquares: ColorCount;     {array}
    Schedule: ClassFile;          {file}
    Source, Results: text;        {text file}
    Crayons: Pastals;             {set}
    List, Pointer: Next Word;     {pointer}

Procedure VeryBusy (Incoming: integer; var Outgoing: integer); {procedure
declaration}
    {A procedure with value and variable parameters.)
   var Local: integer ;

   begin
      readln (Local );
      Outgoing := Incoming * Local
   end; {Very Busy }

Function Capital (Parameter: char): boolean;   {function declaration}
   {Decides if its argument is a capital letter.}
   begin
      Capital := Parameter in ['A'.. 'Z']
   end; {Capital}

begin    {main program}

   writeln ('Let''s start demonstrating things.'); {output statement}
   readln (Frst, Last);                            {input statement}

   if First <= Last then                           {if statement}
      begin                                        {compound statement}
         write (First, ' and ', Second, ' are');
         writeln ('in alphabetical order.')
      end; {if}

   if first = POUNDSIGN then                       {if with an else part}
      High := 10                                   {assignment statement}
   else High := 20;

   for Counter := 1 to LIMIT do                    {for statement}
      read (Name[Counter]);

   case LIMIT div 2 of                             {case statement}
      0, 1, 2, 3,
      4, 5 :                                       {empty statement}
      6, 7, 8, 9: writeln (Within range.)
   end; {case}

   repeat                                         {repeat statement}
      read (Shorts)
   until (Shorts=l) or (Shorts=10);
```

```
   while not eoln do begin                          {while statement}
      read (First);
      writeln (First)
   end; (while }

   with OneCourse do                                {with statement}
      begin
         Name := 'Study Hall';                      {string assignment}
         Units:= 5;
         Grade := 'P'
      end;

   Testing := Capital (First);                       {function call}
   VeryBusy (High, Low);                             {procedure call}

   reset (Source);                                   {file handling}
   read (Source, Last);
   rewrite (Results);
   write (Results, Last);

   new (List );                                      {pointer allocation}
   List^NextWord:= nil;
   Pointer := List

end. {SoTypical}
```

Last update:

Send comments to: webmaster@cs.wwc.edu

CPTR215 Assembly Language Programming 3

Disclamer: these pages are subject to change.

Syllabus

Lecture Notes

	Topic	Reference	Assign't	Due
1	Introduction(4)	Chapter 1	#1, 2 - group	#1 10/5, #2 10/12
2	Data Definition and Transfer(4)	Chapter 2	# 3, 4 - group #5-7 - group	10/15 10/17
	Lab	Experiments Programs	#1-9 individual #1-3 individual	
3	Integer Arithmetic(3)	Chapter 3	#8-21 individual	ASAP
4	Control Structures(3)	Chapter 4		
	MID-TERM EXAM			
5	Procedures, Subprocedures and Macros(4)	Chapter 5		
6	Bit Manipulation(1)	Chapter 6		
7	Arrays and Character Strings(3)	Chapter 8		
8	Interrupts and I/O(4)	Chapter 9		
9	Recursion(2)	Chapter 10		
10	Segment Linking(2)	Chapter 12		
11	FINAL EXAM			

The final date and time of the final exam is listed in the class schedule.

inline stuff

95.6.5 a. aaby

CPTR215 Assembly Language Programming -- 3

Description

Introduction to computer architecture, machine language, and assembly language. Three lectures per week. Prerequisites: Programming in a high-level structured language such as Pascal, C, Modula-2 or Ada

Course Goal

Upon completion of this course you will

- know the basic machine level representations of numeric and non-numeric data,
- have a basic understanding of assembly level machine organization,
- be able to design and code assembly language programs, and
- be able to access assembly language programs from a high-level language.

Resources

Textbook:

Brey, Barry B. **8086/8088, 80286, 80386, and 80486 Assembly Language Programming** Merrill 1994

Evaluation

The course grade is determined by the quantity and quality of work completed in the areas indicated in the following table. The percentages listed are a rule of thumb. The actual percentages used may be lower, depending on the distribution of scores in the class.

GRADING WEIGHTS		LETTER GRADES	
Labs & Homework	50%	A	90 - 100%
Tests	50%	B	80 - 89%
		C	70 - 79%
		D	60 - 69%

95.9.18 a.aaby

Introduction

Four(4) lectures are allocated to this topic. Brey chapter 1

Overview of Computing Systems

- CPU
  - Control
  - ALU
  - Registers
  - Machine language
- Memory
- I/O Subsystem
  - I/O ports: serial, parallel
  - Send/receive
- Peripherals
  - Video Display
  - Keyboard
  - Diskdrives
  - Mouse
- System Unit:
  - Motherboard
  - Support Processors
  - ROM BIOS
  - RAM (Random Access Memory)
  - CMOS RAM
  - Expansion Slots
  - Power Supply
- Intel Microprocessor Family:
  - 80x87
  - 80186
  - 80286
  - 80386
  - 80486
  - P5, P6, P7
  - Compatibility

- System Software

  - Booting
  - OS

- Editors
- Assembler/Compilers
- BIOS/I/O drivers

Microcomputer Architecture

A microcomputer consists of a CPU, memory, and I/O subsystem connected by a bus.

- CPU (Microprocessor): control unit, ALU,
    - registers
        - Instruction Register
        - Program Counter
        - Accumulator registers
        - Index registers
        - Processor status word
    - Instructions
        - Turing Complete (Turing, Church, Post, Markov, ...)
            - Arithmetic operations
            - Logical operations
            - Data transfer operations
            - Transfer of control operations
        - Programming language support
            - Code segmemt -- pc, base
            - Data segment -- base
            - Stack -- stack top, stack frame, stack base
        - Operating system support
            - Memory management
            - supervisor mode
        - Architectural Realization
            - Stack machine -- instructions reference the stack
            - Accumulator machine -- instructions reference the accumulator and memory
            - Register machine -- instructions reference the registers
        - Fetch-execute cycle
- Memory
    - Linear array of cells
    - Fetch/Store

Introducing Assembly Language

- Hierarchy of abstract machines.
    - Microprocessor: interprets and executes machine language
    - Machine language: a sequence of numbers that represents data and instructions.
    - Assembly language: a symbolic form of machine language
    - *Instruction* symbolic representation of single machine instruction

    - format: instruction, operands, comments
    - *Operands* register, variable, memory location, immediate value
    - Assembler: performs translation from assembly language to machine language h-level languages; compilers
    - Why learn assembly language? computer architecture, operating system, utility, freedom, learning tool
- Assembly language applications: Specialized subroutines for highlevel programs. Writing assembly language programs requires attention to detail

Positional Number Systems

- positional number system
    - radix (base)
    - radix point
- decimal number system
    - radix 10 (base 10)
    - decimal point
- binary number system
    - radix 2 (base 2)
    - binary point
    - bit
    - binary to decimal conversion
    - decimial to binary conversion
- octal number system
    - radix 8 (base 8)
    - octal point
    - octal to decimal conversion
    - decimal to octal conversion
    - binary to octal conversion
    - octal to binary conversion
- Hexadecimal nunber system
    - radix 16
    - hex point
    - hexadecimal to decimal conversion
    - decimal to hexadecimal conversion
    - binary to hexadecimal conversion
    - hexadecimal to binary conversion

IBM PC Architecture

Memory Organization

- Memory Architecture]

Micro Processor Architecture

- Central Processing Unit (CPU):
    - Data buss
    - Registers
    - Clock
- Registers:
    - Data Registers
        - AX (accumulator)
        - BX (base)
        - CX (counter)
        - DX (data)
    - Segment Registers
        - CS (code segment)
        - DS (data segment)
        - SS (stack segment)
        - EX (extra segment)
    - Index Registers
        - SI (source index)
        - DI (destination index)
    - Special Registers
        - IP (instruction pointer)
        - BP (base pointer)
        - SP (stack pointer)
    - 80386 Extended Registers
- Flags:
    - Control Flags
    - Status Flags
- Instruction Execution Cycle

MASM: Microsoft Assembler

The Assembly Process

- Edit the Program:
- Assemble the Program:
- Link and Run the Program:

Related Files

- Listing file:
- Map file:
- Batch files:

95.9.18 a. aaby

Assignments: Assembly Language Programming

1. Implement (in C/C++) the Itty Bitty Machine described in the handout. Your implementation must include,
    - a control module (with user interface) with facilities for loading a program into memory, an address in the the PC, and a "start button" to start execution of the program
    - a display of program execution (display of the contents of the registers)
    - EXTRA CREDIT: will be given in addition, if you provide an assembler for your project.
2. Construct a web page with links to resource on the internet for assembly language programming. Your page must include links to free software (assemblers & debuggers) and programming helps (including courses). Especially helpful will be links to inline assembly and calling conventions from high-level languages. Since we will make use of GAS (the GNU assembler) and GDB (the GNU debugger and xxgdb the X window system interface) you should include helpful links to these tools.
3. Write the following programs for the Itty Bitty Machine.
    a. Input a 2-digit number, add 1 to it and display the result on the video screen. Remember that a 2-digit number must be entered as two separate characters, then combined in an appropriate way.
    b. Input three digits and display the largest on the video screen.
4. Construct a C/C++ program with two functions, a main function which prompts the user for input then passes the input to the second function which returns a result to the first function where the result is displayed to the user. This program will be used in a variety of future programming assignments.
5. Show how to convert numbers between two bases.
6. Show how to convert numbers between other bases and twos complement.
7. Show how to do arithmetic with twos complement numbers.

Do at least 10 of 8 - 21 (10 points each):

8. Use rotate to encript a file. Hint: rotate 4 bytes at-a-time.
9. Write a function to convert a 4 digit number in ASCII to an integer
10. Write a function to compute the factorial - recursively
11. Write a function to compute the factorial - iteratively
12. Write a function to compute the factorial and return the result in a reference parameter.
13. Repeat the previous three exercises for fibonacci numbers (1, 1, 2, 3, 5, 8, 13, 21, ...)
14. Write a function to find the largest integer in an array
15. Write a function to multiply two integers that, in turn, calls an add function , the apprpriate number of times, to perform multiplication.
16. Write a function to compute the number of neighbors of a cell (used in life).
17. Write a function to multiply two 4x4 matrices.

18. Write a function max, with up to eight arguments and an argument count that returns the maximum of the given arguments.
19. Write a function to determine if a string is a palindrome.
20. Write a function to convert a string to upper case.
21. Write a function to reverse a string.
22. Show how the stuctured control structures are implemented in assembly code

---

95.9.25 a. aaby

x86 Assembly Programming

Programming Model

Memory

2^{32}- bytes

Registers

8 32-bit General Purpose Registers

Register	**Function**	16-bit low end	8-bit
eax	Accumulator	ax	ah, al
ebx	(base index)	bx	bh, bl
ecx	(count)	cx	ch, cl
edx	(data)	dx	dh, dl
edi	(destination index)	do	
esi	(source index)	si	
ebp	Frame pointer	bp	
esp	Stack top pointer	sp	

6 16-bit Section Registers

Register	**Function**
cs	Code section
ds	Data section
ss	Stack section
es	(extra section)
fs	(supplemental section)
gs	(supplemental section)

EFLAGS Register

S	Sign
Z	Zero
C	Carry
P	Parity
O	Overflow

32-bit EFLAGS Register

32-bit EIP (Instruction Pointer Register)

AT&T Style Syntax (GNU C/C++ compiler and GAS)

- Instruction: **opcode[b+w+l] src, dest**
- Register: **%reg**
- Memory operand size: **[b+w+l]** for byte, word, longword - 8, 16, 32 bits
- Memory references: `section`:`disp`(`base`, `index`, `scale`) where *base* and *index* are optional 32-bit base and index registers, *disp* is the optional displacement, and *scale*, taking the values 1, 2, 4, and 8, multiplies *index* to calculate the address of the operand. -- address is relative to section and is calculated by the expression: base + index*scale + disp
- Constants (immediate operands)
  - 74 - decimal
  - 0112 - binary
  - 0x4A - hexadecimal
  - 0f-395.667e-36 - floating point
  - 'J - character
  - "string" - string

Operand Addressing

- Code: CS + IP (Code segment + Offset)
- Stack: SS + SP (Stack segment + Offset (stack top))
- Immediate Operand: $*constant_expression*
- Register Operand: %*register_name*
- Memory Operand: `section`:`displacement`(`base`, `index`, `scale`) The section register is often selected by default. cs for code, ss for stack instructions, ds for data references, es for strings.

Base	+(	Index	*	Scale	)+	Displacement
eax ebx ecx edx esp ebp esi edi		eax ebx ecx edx ebp esi edi		1 2 3 4		Name Number

- **Direct** Operand: displacement (often just the symbolic name for a memory location)
- **Indirect** Operand: (base)
- **Base+displacement**: displacement(base)
  - index into an array
  - access a field of a record
- **(index*scale)+displacement**: displacement(,index,scale)
  - index into an array
- B**ase + index + displacement**: displacement(base,index)
  - two dimensional array
  - one dimensional array of records
- **Base+(index*scale)+ displacement**: displacement(base, index,scale)
  - two dimensional array

Subroutines

- Function -- returns an explicit value
- Procedure -- does not return and explicit value

The flow of control and the interface between a subroutine and its caller is described by the following:

Caller ...	
call *target*	Transfer of control from caller to the subroutine by 1. saving the contents of the program counter and 2. the program counter (CS:IP) register to the entry point of the subroutine.
Subroutine	
pushl %ebp **movl %esp, %ebp**	Save base pointer of the caller New base pointer (activation record/frame)
Callee ...	Body of Subroutine

movl %ebp,%esp **popl %ebp**	Restore the callers stack top pointer Restore the callers base pointer
ret	Return of control from the subroutine to the caller by alter the program counter (CS:IP) register to the saved address of the caller.
Caller ...	

An alternative is to have the caller save and restore the values in the registers. (Prior to the call, the caller saves the registers it needs and after the return, restores the values of the registers)

Data

Data Representation

- Bits, Bytes, Wyde, word, double word -- modulo 2^n
- Sign magnitude -- sign bit 0=+, 1=-; magnitude
- One's complement -- negative numbers are complement of positive numbers - problem: two representations for zero
- Twos complement (used by Intel) -- to negate:
  - Invert (complement)
  - add 1
- Excess 2^(n-1) (often used for exponent)
- ASCII - character data
- EBCDIC
- BCD

Data Definition Directives

Description provided to the assembler of how static data is to be organized.

- Symbolic name (variables and constants)
- Size (number of bytes)
- Initial value
- **.data**

- Define Byte (DB): (8-bit values) [*name*] DB *initial value* [, *initial value*] see key examples in text; multiple values, undefined, expression, C and Pascal strings, one or more lines of text, $ for length of string
- Define Word (DW): (16-bit words) [*name*] DW *initial value* [, *initial value*] see key examples in text; reversed storage format, pointers
- Define Double Word (DD): (32-bit double words) [*name*] DW *initial value* [, *initial value*]
- Example: p. 80
- DUP Operator: *n* dup(*value*) see key examples in text; type checking

Constant Definitions

- .CONST
- EQU: *name* EQU *constant expression*

Data Transfer Instructions

- **mov *src, dest***
  - src: immediate value, register, memory
  - dest: register, memory
  - except memory, memory
- **xchg *sd1, sd2***
  - Memory, Register
  - Register, Memory
  - Register, Register
- **push *src***
  - src: immediate, register, or memory
- **pop *dest***
  - dest: register or memory
- **pusha** - save all registers on the stack
- **popa** - restore all registers from the stack

Arithmetic Instructions

- **add *src, dest***; **subl *src, dest*** - src +- dest, result in dest
  - Memory, Register
  - Register, Memory
  - Register, Register
- Flags Affected by add and sub: OF (overflow), SF (sign), ZF (zero), PF (parity), CF (carry), AF (borrow)
- i**nc *dest***; **decl *dest*** faster than add/subtract
  - Memory
  - Register
- Flags Affected by inc and dec: OF (overflow), SF (sign), ZF (zero), PF (parity), AF (borrow)
- **adc** & **sbb** add with carry/subtract with borrow - used for adding numbers with more than 32-bits
- **cmp *src, dest*** *computes src - dest* (neither src or dest changes) but may change flags.
  - Memory, Register
  - Register, Memory
  - Register, Register
- **cmpxchg *src, dest*** - compares dest with accumulator and if equal, src is copied into destination. If not equal, destination is copied to the accumlator.
- **neg *dest*** - change sign or two's complement
  - Memory

    - Register
- Flags Affected by NEG: SF (sign), ZF (zero), PF (parity), CF (carry), AF (borrow)
- **mul *src*** - unsigned multiplication EDX:EAX = src * eax
- **imul *src*** - signed multiplication EDX:EAX = src * eax
- Flags Affected by MUL, IMUL:
    - undefined: SF, ZF, AF, PF
    - OF, CF set if upper half is nonzero, set otherwise
- **div src** (unsigned) src is general register or memory quotient eax = edx:eax/src; remainder edx = edx:eax mod src
- **idiv src** (signed) src is general register or memory quotient eax = edx:eax/src; remainder edx = edx:eax mod src
    - Flags Affected by DIV, IDIV:
        - undefined: OF, SF, ZF, AF, PF
        - Type 0 interrupt if quotient is too large for destination register.
- CBW (change byte to word) expands AL to AX - signed arithmetic
- CWD (change word to double word) expands AX to DX:AX - signed arithmetic
- BCD Arithmetic - often used in point of sale terminals
- ASCII Arithmetic - rarely used

Logic Instructions

- **andl *src, dest*** - dest = src and dest
- **orl *src, dest***
- **xorl *src, dest***
- **notl *dest*** - logical inversion or one's complement
- **neg *dest*** - change sign or two's complement
    - Memory
    - Register
- **testl src, dest** (an AND that does not change dest, only flags)

Shift and Rotate Instructions

- Logical Shift
    - **shr *count, dest*** - shift dest count bits to the right
    - **shl *count, dest*** - shift dest count bits to the left
- Arithmetic Shift(preserves sign)
    - **sar *count, dest*** - shift dest count bits to the right
    - **sal *count, dest*** - shift dest count bits to the left
- Rotate without/With carry flag
    - **ror *count, dest*** - rotate dest count bits to the right
    - **rol *count, dest*** - rotate dest count bits to the left
    - **rcr *count, dest*** - rotate dest count bits to the right
    - **rcl *count, dest*** - rotate dest count bits to the left
- **test *arg, arg*** (an AND that does not change dest, only flags)

- **cmp *src, dest*** subtract src from dest (neither src or dest changes) but may change flags.
  - Memory, Register
  - Register, Memory
  - Register, Register
  - CMP
- Flag Bit Operations
  - Complement CF: CMC
  - Clear CF, DF, and IF: CLC,CLD,CLI,
- Set CF, DF, and IF: STC, STD, STI

Control Transfer Instructions

- **cmp *src, dest*** - compute dest - src and set flags accordingly
- Jump instructions: the transfer is one-way; that is, a return address is not saved.

```
NEXT:...
     ...
     jmp NEXT  ;GOTO NEXT
```

Jump Instructions

jmp *dest*		unconditional	NEXT:... ... **jmp NEXT** ;GOTO NEXT
Unsigned conditional jumps			
jcc dest			
ja/jnbe	C=0 and Z=0	Jump if above	
jae/jnb	C=0	Jump if above or equal to	
jb/jnae	C=1	Jump if below	
jbe/jna	C=1 or Z=1	Jump if below or equal to	
jc	C=1	Jump if carry set	
je/jz	Z=1	Jump if equal to	
jnc	C=0	jump if carry cleared	
jne/jnz	Z=0	jump if not equal	
jnp/jpo	P=0	jump if no parity	
jp/jpe	P=1	jump on parity	
jcxz	cx=0	jump if cx=0	gcc does not use
jecxz	ecx=0	jump if ecx=0	gcc does not use
Signed conditional jumps			

jcc dest		
jg/jnle	Z=0 and S=0	jump if greater than
jge/jnl	S=0	jump if greater than or equal
jl/jnge	S=1	jump if less than
jle/jng	Z=1 or S=1	jump if less than or equal
jno	O=0	jump if no overflow
jns	S=0	jump on no sign
jo	O=1	jump on overflow
js	S=1	jump on sign

- Loop instructions: The loop instruction decrements the ecx register then jumps to the label if the termination condition is not satisfied.

```
    movl count, %ecx
LABLE:
            ...
            loop LABEL
```

	Termination condition	
loop *label*	ecx = 0	gcc does not use
loopz/loope *lab el*	ecx =0 or ZF = 0	gcc does not use
loopnz/loopne *label*	ecx = 0 or ZF = 1	gcc does not use

- **call *name*** - call subroutine ***name***
- **ret** - return from subroutine
- **enter**
- **leave**
- **int *n*** - interrupt
- **into** - interrupt on overflow
- **iret** - interrupt return
- **bound** - value out of range
- IF C THEN S;
- IF C THEN S1 ELSE S2;
- CASE E DO c1 : S1; c2 : S2; ... cn : Sn end;
- WHILE C DO S;
- REPEAT S UNTIL C;
- FOR I from J to K by L DO S;

String Instructions

The sring instructions assume that by default, the address of the source string is in ds:esi (section register may be any of cs, ss, es, fs, or gs) and the address of the destination string is in es:edi (no override on the destination section). Typical code follow the scheme

initialize esi and edi with addresses for source and destination strings
initialize ecx with count
Set the direction flag with **cld** to count up, with **std** to cound down
prefix string-operation

- [prefix]**movs** - move string
- [prefix]**cmps** - compare string WARNING: subtraction is dest - source, the reverse of the cmp instruction
- [prefix]**scas** - scan string
- [prefix]**lods** - load string
- [prefix]**stos** - store string
- String instruction prefixes: The ecx register must be initialized and the DF flag in initialized to control the increment or decrement of the ecx register. Unlike the loop instruction, the test is performed before the instruction is executed.
    - **rep** - repeat while ecx not zero
    - **repe** - repeat while equal or zero (used only with cmps and scas)
    - **repne** - repeat while not equal or not zero (used only with cmps and scas)

Miscellaneous Instructions

- **leal *src, dest***(load effective address -- the address of src into dest)
    - Memory, Register
- **nop**
- **xlat/xlatb**
- **cpuid**

Floating Point Instructions

Floating Point
8 32-bit registers

Register	Function
st	
st(0)	
st(1)	
...	
st(7)	

MMX Instructions

System Instructions

- **hlt**
- **lock**
- **esc**
- **bound**
- **enter leave**

Interrupts

- **int**
- **into**

Memory Management Unit

- **invlpg**

Cache

References

- **www.x86.org**

Assembly Laboratory

One(1) lecture is allocated to this topic.

Edit

You may use any editor that produces ASCII text files.

G++ Options

Source program: **program.cpp**

- Compile to a.out

  -> g++ program.cpp
- Compile to named file

  -> g++ program.cpp -o program
- Generate assembly program

  -> g++ -S program.cpp
- Optimize a program
  -> g++ -O program.cpp
- Generate and optimize an assembly program
  -> g++ -O -S program.cpp

GAS

Source program: **program.s**

- Assemble

  -> as program.s -o program.o
- Compile

  -> gcc program.o -o program

Run

To execute the file, just enter the name of the executable file produced (or a.out if the naming option is not used).

Activities

Experiment: Do the following assembly experiments and hand in appropriately documented assembly source programs

1. Global data declarations
2. Character, string, integer, and floating point representations.
3. Assignment operations
4. Arithmetic operations
5. Function prototype declarations.
6. Calling conventions: stack manipulations
7. Calling conventions: return value
8. Calling conventions: pass by value
9. Calling conventions: pass by reference

Programming: Without looking at an assembly version of a C/C++ program: Use inline assembly code to --

1. f(x) = 3x+4
2. swap the values of two integers
3. sort an array of integers

References

- DJGPP - www.delorie.com
- Brennan's DJGPP & guide to inline assembly
- George Foot's functions in assembly language
- Assembly HOWTO
- GNU - gcc, as, gdb & documentation (also found at delorie.com)
- FAQ - RayMoon's x86 Assembly Language FAQ

Arrays and Character Strings

One(3) lectures are allocated to this topic. Runnion chapter 8

Arrays and Strings

- Defining and initializing arrays. see Runnion p. 372

```
DB 100 DUP(?)
DW 100 DUP(?)
```

- Accessing Array Elements: MOV TABLE[BX], AL
- String Instructions
  - Source is DS:SI
  - Dest is ES:DI
  - SI/DI are automatically incremented/decremented
  - if DF is 0/1, then increment/decrement
  - CLD sets DF to 0; STD sets DF to 1
  - MOVS (moves a byte/word)
  - LODS (loads AL/AX with a byte/word from DS:SI)
  - STOS (stores AL/AX with a byte/word in ES:DI)
  - CMPS (compares byte/word in DS:SI with byte/word in ES:DI): src-dst
  - SCAS (scans byte/word in ES:DI) src-AL/AX
- Repeat prefixs

```
Repeat
        Execute string instruction
        Decrement CX by 1
Until CX = 0 (or ??)
```

  - REP *SringOp, operand*
  - REPE/REPZ *SringOp, operand*
  - REPNE/REPNZ *SringOp, operand*

Addressing Modes

- Register (operand is contents of register): INC AL; MOV BX,DX
- Direct (operand is contents of memory): INC COUNT; MOV SUM,0
- Immediate (operand is a literal): MOV COUNT,16; CMP CHAR,'*'

Memory addressing modes

- Direct: (see above)
- Register Indirect (address is in register): MOV [DI],AX
- Indexed (operand = base + displacement): MOV TABLE[DI], AX
- Base addressing (operand = base register + displacement): MOV CX,6[BP]
- Base Indexed addressing: see Runnion p. 404

95.11.3 a. aaby

Interrupts and I/O

Four(4) lectures are allocated to this topic. Runnion chapter 12

Interrupts

An **interrupt** is an external request for service -- stop executing current procedure and begin executing an interrupt service procedure. It is **maskable** if it can be ignored and is **nonmaskable** if it must be acknowledged.

Required hardware

1. Recognize interrupt
2. Stop currently executing procedure and initiate designated procedure
3. Save and restore processor state.

Each interrupt is associated with an integer (interrupt type) in the range of 0-255 which is an offset into the **interrupt vector** (at addresses 00000-003FF) which contains the address of corresponding interrupt service procedure (interrupt handler).

The processor checks for pending interrupts at the end of most instruction executions. When an interrupt is detected, the following steps are performed by the processor.

1. The flags register is pushed onto the stack
2. The trap flag and the interrupt flag are cleared
3. The CS-register is pushed onto the stack
4. The location of the interrupt vector is computed from the interrupt type
5. The IP-register is pushed onto the stack
6. The first word of the interrupt vector is loaded into the IP-register.

External Interrupt (most are maskable)

- INTR line (maskable interrupts)
    - To disable interrupts, clear the IF bit: CLI
    - To enable interrupts, set the IF bit: STI
- NMI (non-maskable interrupt line) from controllers.

Internal Interrupts

1. Divide overflow

2. Single step mode
3. INT instruction used for
    - DOS Function Calls
    - BIOS-Level Video Control (INT 10h)

See page 433 for interrupt types and vector offest

Some predefined interrupts

- Type 0: Divide overflow
- Type 2: Non-maskable (memory or parity errors)
- Type 4: Overflow
- Type 8: System timer (DOS time of day update)
- Type 9: Keypress

I/O interrupts

- Type 10H: Video I/O; see p 448
- Type 13H: Diskette I/O
- Type 16H: Keyboard I/O
- Type 19H: Power-on reset
- Type 1BH: Control break
- Type 21H: DOS user services; see p 463

95.11.9 a. aaby

Segment Linking

Two(2) lectures are allocated to this topic. Runnion chapter 12

High-Level Language Interface

Assembly routines are frequently used for time-critical operations, bit manipulations, and interrupt control operations. The interface between an assembly language subroutine and a high-level language is specific to the particular high-level language.

General Conventions

- Considerations: naming convention, memory model, calling conventions
- External Identifiers: must be consistent with high-level language -- Pascal: all uppercase: C: case sensitive, begins with an underscore
- Segment names: must be compatible
- Memory Models: use default model for the calling program

The Turbo-Pascal Interface

1. All functions and procedures must be defined in a segment named CODE.
2. All variables must be defined in a segment named DATA and must be defined without initial value.
3. See p 591 for corresponding TurboPascal and MASM data types
4. Arguments are passed via the stack and are pushed onto the stack in left-to-right order.
5. Identifiers are case insensitive
6. Arguments are popped from the stack as part of the procedure return.
7. Function values are returned in registers as follows:
    - Double word values in DX:AX
    - Word value in AX
    - Byte value in AL

The Turbo C++ Interface

- Identifiers: Identifiers are case sensitive and all exported assembly subroutine names must begin with an underscore.
- Defining the function (example: extern int sub1(void)
- Saving registers: Assembly subroutines must save and restore the registers (BP, CS, DS, SS, ES, SI and DI).
- Arguments are passed via the stack and are pushed onto the stack in right-to-left order.
- Arguments are popped from the stack as part of the procedure return.

- Function results: returned in AX or DX:AX registers. Structures and arrays are stored in static data areas. Function values are returned in registers as follows:
    - Double word values in DX:AX
    - Word value in AX
    - Byte value in AL

95.9.18 a. aaby

x86 Inline Assembly Programming

Assumptions

The program is in a single file. All variables are 32-bit.

Inline Assembly

Inline assembly code may be included as a string parameter, one instruction per line, to the asm function in a C/C++ source program.

```
...
asm("incl x;
        movl 8(%ebp), %eax
      ");
```

Where the basic syntax is: **asm** [**volatile**] **(** /*asm statements*/
[**:** /* outputs - comma separated list of constraint name pairs */
[**:** /* inputs - comma separated list of constraint name pairs*/
[**:** /* clobbered space - registers, memory*/
]]]**);**

- asm statements - enclosed with quotes, at&t syntax, separated by new lines
- outputs & inputs - constraint-name pairs "constraint" **(**name**),** separated by commas
- registers-modified - names separated by commas

Constraints are

- **g** - let the compiler decide which register to use for the variable
- **r** - load into any avaliable register
- **a** - load into the eax register
- **b** - load into the ebx register
- **c** - load into the ecx register
- **d** - load into the edx register
- **f** - load into the floating point register
- **D** - load into the edi register
- **S** - load into the esi register

The outputs and inputs are referenced by numbers beginning with %0 inside asm statements.

Example:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

int f( int );

int main (void)
{
  int x;
  asm volatile("movl $3,%0" :"=g"(x): :"memory"); // x = 3;
  printf("%d  ->  %d\n",x,f(x));
}

/*END   Main    */

int f( int x )
{
   asm volatile("movl %0,%%eax
                 imull $3, %%eax
                 addl $4,%%eax"
          :
          :"a" (x)
          : "eax", "memory"
         ); //return (3*x + 4);
}
```

Global Variables

Assuming that x and y are global variables, the following code implements x = y*(x+1)

```
asm("incl x
        movl x, %eax
        imull y
       movl %eax,x
      ");
```

Local variables

Space for local variables is reserved on the stack in the order that they are declared. So given the declaration: int x, y;

x is at -4(%ebp)
y is at -8(%ebp)

Value Parameters

Parameters are pushed onto the stack from right to left and are referenced relative to the base pointer (ebp) at four byte intervals beginning with a displacement of 8. So in the body of p(int x, int y, int z)

x is at 8(%ebp)
y is at 12(%ebp)
z is at 16(%ebp)

Reference parameters

Reference parameters are pushed onto the stack in the same order that value parameters are pushed onto the stack. The difference is that access to the value to which the parameter points is as follows

p(int& x,...

```
movl 8(%ebp), %eax  # reference to x copied to eax
movl  $5, (%eax)        # x = 5
```

References

- **www.x86.org**
- DJGPP
- DJGPP QuickAsm Programming Guide

CPTR 221-222 Programming Languages -- 3, 3

Syllabus

Schedule (subject to change, check back often)

CPTR 221 (Theory, Design, & Implementation)			
Lecture Notes	**Textbook/Reading Material**	**Assignment**	**Due**
Introduction - 2	Introduction	#14 Paper	**Jan 19**
Syntax - 10	Syntax	#4, 10, 12, 18	Jan 23
Semantics - 3	Semantics	# 1c, 2a, 2e, 2f, 3b, 3c	Feb 9
Translation - 3	Translation Stack Machine	# 5, 6, 7 #6	Feb 27
Imperative Programming - 3	Imperative Programming	#16, 17	March 6
Data and Data Structuring - 3	Data and Data Structuring	#	
CPTR 222 (Theory, Design, & Paradigms)			
Lecture Notes	**Textbook/Reading Material**	**Assignment**	**Due**
Abstraction and Generalization - 3	Abstraction and Generalization	#	
Object-Oriented Programming	Object-Oriented Programming	Chapter review	
Database & Information Retrieval - 9	Overview, Models & Applications - 4 The Relational Model - 5	Exercises	
Logic Programming	Logic Programming Prolog Tutorial & more Godel Tutorial	Exercises none RDMS # 1, 4c FamilyDB #1 Parser none	

Artificial Intelligence - 9	History and Applications - 3 Problems, State Spaces, & Search - 6	
Functional Programming	Functional Programming Scheme Tutorial Haskell Tutoral SML Tutorial	Arithmetic Functions # 1,4 Numeric Lists # 3,5,6 Polymorphic lists # 6,11,15 Sorting # 2,4,7 Higher Order # 1,3,6
Concurrent Programming - 3	Concurrent Programming PCN Tutorial MPI Tutorial	Producer-consumer Dining Philosophers # 1 Miscellaneous # 2
Pragmatics	Pragmatics	Exercises Chapter review

Exams

- 221 Midterm Exam Information
- 221 Final Exam Information
- 222 Midterm Exam Information
- 222 Final Exam Information

OLD Stuff

Projects

- Paper: Language description
- Compiler (scanner, parser, symbol table, stack-machine code)
- Virtual machine
- File programming: transaction processing (update, merge)
- DBMS (product, selection, projection, natural join)
- Expert System
- Natural Language Interface
- 121

  Concurrent Programming: Parameter Passing
- 122

Logic programming: exercises
AI: Natural Language Processing

Forms

CPTR221, 222 Programming Languages -- 3, 3

Winter-Spring 98

Description

History; virtual machines; representation of data types; sequence control, sharing and type checking; run-time storage management; finite state automata and regular expressions; context-free grammars and pushdown automata; language translation systems; semantics; programmming paradigms; and distributed and parallel programming constructs. Prerequisite CPTR 143; CPTR 215 strongly recommended.

The course provides three lectures per week.

Goals

Upon completion of this course you will

- ...

Resources

Textbooks:
Aaby, Anthony (1996) *Introduction to Programming Languages*
Thompson, Simon (1996) *The Craft of Functional Programming* Addison-Wesley
Other Books:
Pratt, T.W. *Programming Languages: Design and Implementation* Prentice-Hall 1975.
Watt, D.A., *Programming Language Concepts and Paradigms* Prentice-Hall International 1990
Watt, D.A., *Programming Language Processors* Prentice-Hall International 1993.
Watt, D.A., *Programming Language Syntax and Semantics* Prentice-Hall International 1991.
Ullman, Jeffery *Elements of ML Programming* Prentice-Hall
Clocksin and Mellish, *Programming in Prolog* 4th Ed. Springer-Verlag 1994.
Lab Manual:
Handouts
Journals:
Communications of the ACM, Computing Surveys, Letters on Programming Languages and Systems, Transactions of Programming Languages and Systems, Transactions of Software Engineering and Methodology, Journal of the ACM

Internet:

comp.lang.* -- esp. functional, ml, prolog;

Haskell

Term Projects

- A review of the textbook.
- Chapter reviews
- A written and oral report on a journal article (one each quarter). Sources include:
  - ACM Journal, TOPLAS, TOSEAM
  - ACM SIGPLAN, Software Engineering Notes, OOPS Messinger
- And either

  install a programming language and prepare a tutorial introduction (in html) to the language with sample programs, e.g.
  - Logic: Prolog, Gödel
  - Functional: Scheme, Haskell (Hugs), Sisal ...
  - OO: Modula-3, ...

  (See the descriptions and tutorials in the Software home page.)

  or

  produce a significant program in a functional or logic programming language.

  The choices must be made within the first three weeks of the begining of winter term.

Evaluation

The course grade is determined by the quantity and quality of work completed in the areas indicated in the following table. The percentages listed are a rule of thumb. The actual percentages used may be lower, depending on the distribution of scores in the class. The grade expectations document helps to explain the different grades. The grading criteria are used to grade programs. The course grade is determined by the quantity and quality of work completed on the project and the assigned homework problems.

Grading Weights	**Letter Grades**
Homework: 50%	A: 90 ~ 100%
Tests: 50%	B: 80 ~ 89%
	C: 70 - 79%
	D: 60 - 69%

Each student will participate in the evaluation process.

Last Modified

Send comments to aabyan@wwc.edu

LN: PL -- Introduction

What is a complete description of a programming language?

Motivational example:

- constant
- expression
- abstraction (naming)
- local environment, block
- scope
- generalization
- variable
- free and bound variables
- parameterization

Complete description of a programming language

- syntax
- computational model
- semantics
- pragmatics

Data

- Simple data types and their implementation
- Compound data types and their implementation
- Abstract data types

Models of Computation

- Functional
- Logical
- State transition

Computability & the equivalence of the models

Syntax and Semantics

Pragmatics

Language Design Principles

Last Modified

Send comments to aabyan@wwc.edu

Term Paper

Read Peter Naur's paper *Report on the Algorithmic Language ALGOL 60* and one other paper on some other programming language of your choice in the listed references. Write a paper of at least two pages in length summarizing/evaluating/reacting to the papers.

References

Horowitz, Ellis
Programming Languages: A Grand Tour Computer Science Press 1983
Laplante, Phillip
Great Papers in Computer Science West Publising Company 1996

Last Modified
Send comments to aabyan@wwc.edu

LN: PL -- Syntax

How should we describe the structure of a programming language?

Context-free Grammars

- Grammars and Languages
- Abstract Syntax
- Parsing
- Table-driven and recursive descent parsing

Nondeterministic PDAs

Regular expressions

- concatenate, alternative, repetition, grouping

FSA

- deterministic and non-deterministic
- fsa & regular expressions
- transition function -- graph, table
- implementation -- case statement, procedures, table-driven

Pragmatics

- single pass
- semicolons
- case
- keywords
- assignment
- function calls
- return value

Last Modified
Send comments to aabyan@wwc.edu

LN: PL - Semantics

Algebraic Semantics

A many-sorted algebra

- Domain, Functions, Axioms/Equations, Errors

Useful for defining abstract data types and objects

Figure N.2: Algebraic definition of an Integer Stack ADT

Domains:

Nat (the natural numbers
Stack (of natural numbers)
Bool (boolean values)

Functions:

newStack: () -> Stack
push : (Nat, Stack) -> Stack
pop: Stack -> Stack
top: Stack -> Nat
empty : Stack -> Bool

Axioms:

pop(push(N,S)) = S
top(push(N,S)) = N
empty(push(N,S)) = false
empty(newStack()) = true

Errors:

pop(newStack())
top(newStack()) where N in Nat and S in Stack.

Axiomatic Semantics

- Assertions
- Hoare triples
- Inference rules
- Verification

Program construction**Figure N.4:** Verification of S = $sum_{i=1}^{n}A[i]$

	Pre/Post-conditions	**Code**
1.	{ 0 = $Sum_{i=1}^{0}$`A[i]`, 0 < |A| = n }	
2.		`S,I := 0,0`
3.	{`S` = $Sum_{i=1}^{I}$A[i], I <= n }	
4.		`while I < n do`
5.	{`S` = $Sum_{i=1}^{I}$A[i], I < n }	
6.	{`S+A[I+1]` = $Sum_{i=1}^{I+1}$A[i], I+1 <= n }	
7.		`S,I := S+A[I+1],I+1`
8.	{ S = $Sum_{i=1}^{I}$A[i], I <= n }	
9.		`end`
10.	{`S` = $Sum_{i=1}^{I}$A[i], I <= n, I >= n }	
11.	{`S` = $Sum_{i=1}^{n}$A[i] }	

- Program construction

Assignment Axiom
{P[x:E]} `x:= E` {P}

If after the execution of the assignment command the environment satisfies the condition P, then the environment prior to the execution of the assignment command also satisfies the condition P but with E substituted for x (In this and the following axioms we assume that the evaluation of expressions does not produce side effects.).

Loop Axiom:

$$\frac{\{I \wedge B \wedge V > 0\}\ C\ \{I \wedge V > V' >= 0\}}{\{I\}\ \texttt{while B do C end}\ \{I \wedge \neg B\}}$$

To verify a loop, there must be a loop invariant I which is part of both the pre- and post-conditions of the body of the loop and the conditional expression of the loop must be

true to execute the body of the loop and false upon exit from the loop.

Loop Correctness Principle: Each loop must have both an *invariant* and a *variant*.

Rule of Consequence:

$$\frac{P \rightarrow Q,\ \{Q\}\ C\ \{R\},\ R \rightarrow S}{\{P\}\ C\ \{S\}}$$

Sequential Composition Axiom:

$$\frac{\{P\}\ C0\ \{Q\},\ \{Q\}\ C1\ \{R\}}{\{P\}\ C0;\ C1\ \{R\}}$$

The sequential composition of two commands is permitted when the post-condition of the first command is the pre-condition of the second command.

Selection Axiom:

$$\frac{\{P \wedge B\}\ C0\ \{Q\},\ \{P \wedge \neg B\}\ C1\ \{Q\}}{\{P\}\ \text{if } B \text{ then } C0 \text{ else } C1 \text{ fi } \{Q\}}$$

Conjunction Axiom:

$$\frac{\{P\}\ C\ \{Q\},\ \{P'\}\ C\ \{Q'\}}{\{P \wedge P'\}\ C\ \{Q \wedge Q'\}}$$

Disjunction Axiom:

$$\frac{\{P\}\ C\ \{Q\},\ \{P'\}\ C\ \{Q'\}}{\{P \vee P'\}\ C\ \{Q \vee Q'\}}$$

Denotational Semantics
Operational Semantics

Last Modified
Send comments to aabyan@wwc.edu

LN: PL -- Translation

Introduction

- virtual machines
- compilers & assemblers
- interpreters & virtual machines
- linkers & loaders
- Compiler phases
    - Scanner
    - Parser
    - Symbol table and error handling
    - Semantic checking
    - Intermediate representation
    - Optimization
    - Code generation

Recursive Descent Parser

- Transform the grammar
- Compute first and follow sets
- Construct parsing procedures
- Complete the parser

Scanner Construction

- Transform the grammar
- Translate production rules to procedures
- Complete the scanner

Attribute Grammars

Last Modified
Send comments to aabyan@wwc.edu

LN: PL -- Imperative Programming

- State sequence
- Variables, Assignment & binding
- Unstructured commands
- Structured commands
- Sequential expressions
- Subprograms, procedures, and functions
- Other control structures -- cooroutines and processes
- Reasoning about imperative programs
    - Sequencers: goto, return, exit, exceptions
        - Domain failures
        - Range failures
    - Side effects
    - Alasing & dangling references

LN: PL -- Abstraction and Generaliztion

The management of complexity

- Abstraction
- Invocation
- Substitution
- Generalization
- Specialization
- Binding
    - collateral binding (independent)
    - sequential binding
    - recursive binding
- Encapsulation
- Block structure
    - monolithic
    - flat
    - nested (hierarchical)
- Scope rules - name visibility & reus
    - static
    - dynamic
- Environment
- ADTs
- Pragmatics
    - Binding times
        - Language design time
        - Language implementation time
        - Program translation time
        - Program execution time
    - Procedures and functions
        - Activation records
        - Parameters and arguments
            - strict
            - non-strict
            - eager
            - lazy
            - passing
                - copy
                - definitional
    - Scope and blocks

- Partitions
- Modules

Chapter Review: Programming Languages

Name: **Chapter:**

Comment on the following:

1. Organization: Is the chapter organization *logical*? If no, how would you re-order?

2. Examples: Do the examples *effectively* to support the concepts in this chapter?

3. Writing Style: Is the chapter *clear* and *readable* to you? If no

4. Exercises: Are the exercises of *sufficient number, type* and c*learly written?*

DB1: Overview, Models, and Applications of Database Systems

Introduction to the basic goals, functions, models, components, applications, and social impact of database systems

Recurring Concepts: complexity of large problems, conceptual and formal models, evolution, trade-offs and consequences.

Lecture Topics: (four hours minimum)

1. History and motivation for database systems
2. Components of database systems; data, dictionary, database management system, application programs, users, administration
3. Conceptual modeling (e.g., entity-relationship, object-oriented)
4. Functions supported by a typical database system; access methods, security, deadlock and concurrency problems, fourth generation environments
5. Recent developments and applications (e.g., hypertext, hypermedia, optical disks)
6. Social impact of database systems; security and privacy

Suggested Laboratories:

1. (closed) Interaction with a database management system. Students create a small database and evaluate how the system supports functions introduced in lectures.
2. (closed) Work in small teams on a conceptual modeling problem. Students evaluate the advantages and disadvantages of alternative solutions

Connections:

- Related to: OS8, SE4, SP1, SP2, SP3
- Prerequisites:
- Requisite for: DB2

Last update:

Send comments to: webmaster@cs.wwc.edu

DB2: The Relational Data Model

Introduction to the relational data model and the concept of a non-procedural query language. Mapping a conceptual model to a relational schema.

Recurring Concepts: complexity of large problems, conceptual and formal models, consistency and completeness, levels of abstraction, trade-offs and consequences.

Lecture Topics: (five hours minimum)

1. Relational data model terminology; mapping conceptual schema to a relational schema
2. Representing relationships, entity and referential integrity
3. Overview of relational algebra
4. Representing database relationships
5. Database query language; data definition, query formulation, update sublanguage, expressing constraints, referential integrity, embedding in a procedural language

Suggested Laboratories: (closed) Using a procedural language, students will implement the join operation of the relational algebra. At least two diverse implementations will be attempted in order to demonstrate the relative efficiency of the various techniques. The goal of this lab is to introduce students to the computational aspects of relational algebra.

Connections:

- Related to: OS7
- Prerequisites: DB1, SE1, Discrete Mathematics
- Requisite for:

Last update:

Send comments to: webmaster@cs.wwc.edu

Database

Objectives

- To introduce the programming language Prolog.
- To introduce the Prolog approach to relational databases,

Background

The mathematical concept underlying the relational database model is the set-theoretic *relation*, which is a subset of the Cartesian product of a list of domains. A domain is a set of values. A *relation* is any subset of the Cartesian product of one or more domains. The members of a relation are called *tuples*. In relational databases, a relation is viewed as a table. The Prolog view of a relation is that of a set of named tuples. For example, in Prolog form, here are some unexpected entries in a city-state-population relation.

```
city_state_population('San Diego','Texas',4490).
city_state_population('Miami','Oklahoma',13880).
city_state_population('Pittsburg','Iowa',509).
```

Items in this form are called **facts**. In addition to defining relations as a set of tuples, a relational database management system (DBMS) permits new relations to be defined via a query language. In Prolog this means defining a **rule**. Rules take the form of an *If-then* statement. For example, the sub-relation consisting of those entries where the population is less than 1000 can be defined as follows:

```
smalltown(Town,State,Pop) :- city_state_population(Town,State,Pop),
                             Pop < 1000.
```

The operator (**:-**) is read as **if** and the comma is read as **and**. The semicolon (;) is read as logical **or** but it is best to provide another rule instead. Negation is provided with by **not** (it is implemented by failure). The built-in relational operators are:

```
=:=             Values are equal
=               Unified
<               Less than
>               Greater than
=<              Equal or Less than
>=              Greater than or equal
=\=             Values are not equal
```

```
\=              Not unifiable equal
```

Atoms (literals or constants) are either numbers, quoted strings or identifiers beginning with a lowercase alphabetic symbol. **Variables** are single assignment and are identifiers beginning with an uppercase alphabetic symbol. Interaction with an interactive Prolog environment occurs through a **query**. Querys have the form:

```
?- goal-list.
```

For example, if we wanted to see if there was a small town in Texas in our database, we would use the query:

```
?- smalltown(Town, 'Texas', Pop).
```

In an interactive Prolog environment, a database (Prolog program) is loaded using the **consult** predicate as follows:

```
?- consult( filename ).
```

Note that facts, rules and querys all terminate with a period (.).

Assignment

1. Construct a family data base of entries of the form:

   ```
   male( Name ).
   female( Name ).
   parent_of( Parent, Child ).
   ```

   and define the following relations,

   ```
   father_of(F,C)
   mother_of(M,C)
   son_of(P,C)
   dau_of(P,C)
   grandfather_of(GF,GC)
   grandmother_of(GM,GC)
   aunt_of(A,N)
   uncle_of(U,N)
   ancestor_of(A,D)
   half_sis(HS,Sib)
   half_bro(HB,Sib)
   ```

2. Suppose we informally define the data in the database of a department store as follows.

1. Each employee is represented, his name, employee number, address and department he works for.
2. Each department is represented, its name, employees, manager, and items sold.
3. Each item sold is represented, its name, manufacturer, price, model number, and store supplied stock number.
4. Each manufacturer is represented, its name, address, items supplied to the store, and their prices.

Design a simple database containing this information.

Last update:

Send comments to: webmaster@cs.wwc.edu

Parsers

Objectives

To construct recursive descent and table driven parsers using Prolog.

Background

We use the following grammar for our examples.

```
program --> statementsequence .
statementsequence --> statement | statement ; statementsequence
statement --> if condition then statementsequence else statementsequence endif
              | variable := expression
condition --> variable = expression
```

Recursive Desent

```
program( Input ) :- statementsequence( Input, ['.'] ).

statementsequence( Input, Output ) :- statement( Input, Rest ),
                                      statementsequence( Rest, Output ).

statement( [if|Input], Output ) :- condition(Input,[then|R1]),
                                   statementsequence(R1,[else|R2]),
                                   statementsequence(R2,[endif|Output]).

statement( [V, ':=', E | Output], Output) :- variable(V), expression(E).

conditition([V, '=', E | Output], Output) :- variable(V), expression(E).

variable(V) :- atom(V), not rw(V).

expression(E) :- atomic(E), not rw(E).

rw(X) :- in(X,[if,then,else,endif,':=',=,variable,expression]).
```

Table Driven

We use the following table representation of our grammar

```
start(program).
p(program,[statementsequence,' .']).

p(statementsequence,[statement]).
p(statementsequence,[statement,statementsequence]).

p(statement,[if,condition,then,statementsequence,else,statementsequence,endif]).
p(statement,[variable,':=',expression]).
```

```
p(condition,[variable,=,expression]).

token(X,variable)   :- atom(X), not rw(X), not nt(X).
token(X,expression) :- atomic(X), not rw(X), not nt(X).

rw(X) :- in(X,[if,then,else,endif,':=',=,variable,expression]).
nt(X) :- in(X,[program,statement,statementsequence,condition,expression,variable]).

%% Alternate grammar for bottom up parser
p(statementsequence,[statement]).
p(statementsequence,[statementsequence,statement]).
```

Top Down

```
top_down(Input) :- start(Start), top_down([Start],Input).

top_down([],[]) :- print('Parsing complete -- Program Accepted'), nl.
top_down([X|Stack],[X|Input]  ) :- top_down(Stack,Input).
top_down([X|Stack],[Top|Input]) :- token(X,TokenType), Top = TokenType,
                                   top_down(Stack,Input).
top_down([Top|Stack],Input) :- p(Top,RHS), concat(RHS,Stack,NewStack),
                               top_down(Input,NewStack).

t :- top_down([x,':=',3,if,x,=,3,then,x,':=',4,else,x,':=',5,endif]).
```

Bottom up

```
bup(Input) :- bup(Input,[]).

bup([],[S]) :- start(S), print('Parsing complete -- Program Accepted'), nl.
bup(Input,S) :- S  [], split(F,Handle,R,S),
                p(N,W),
                reverse(W,Handle),
                concat(F,[N|R],Ns),
                bup(Input,Ns).
bup(Input,S) :- S  [], append(F,[Token|R],S),
                token(Token,TokenType),
                concat(F,[TokenType|R],Ns),
                bup(Input,Ns).
bup([I|Input],S) :- bup(Input,[I|S]).

split(F,Handle,R,L) :- append(F,T,L), append(Handle,R,T).

t :- bup([x,':=',3,if,x,=,3,then,x,':=',4,else,x,':=',5,endif,'.']).
```

Study the parsers and their execution on the sample program. Notice that top-down parsing is driven by the grammar for the language. The parsing is goal driven, that is, the grammar is used to tell the parser what to look for in the input. The grammar rules are used in a left to right manner. In contrast, the bottom-up parser is driven by the input, it compares the input to the right hand sides of the grammar rules and attempts to reduce the input to the left hand side of the grammar rules.

Assignment

1. Finish the recursive descent parser for the programming language Simple.
2. Finish adding the production rules for the programming language Simple to the top-down parser.
3. Finish adding the production rules for the programming language Simple to the bottom up parser .
4. Modify one of the parsers so that it generates code.

Last update:

Send comments to: webmaster@cs.wwc.edu

AI1: History and Applications of Artificial Intelligence

Introduction to the basic history, premises, goals, social impact, and philosophical implications of artificial intelligence. Capabilities and limitations; applications in expert systems, natural language, robotics, planning, speech, and vision.

Recurring Concepts: conceptual and formal models, evolution, levels of abstraction trade-offs and consequences.

Lecture Topics: (three hours minimum)

1. History, scope, and limits of artificial intelligence; the Turing test, games
2. Social, ethical, legal, and philosophical aspects
3. Expert systems and shells; effective applications
4. Natural language; Extended example
5. Speech and vision
6. Robotics and planning

Suggested Laboratories: (closed)

1. Interaction with an existing expert system; observing its behavior, capabilities, and limitations.
2. Construction of a small expert system using an expert system shell. Students will assess the advantages and disadvantages of using such a shell, compared with solving a similar problem in an Al language such as LISP or PROLOG.

Connections:

- Related to: HU1, PL1, SP1, SP2, SP3
- Prerequisites:
- Requisite for: AI2

Last update:

Send comments to: webmaster@cs.wwc.edu

AI2: Problems, State Spaces, and Search Strategies

Identification of fundamental classes of algorithms for artificial intelligence. Methods for developing appropriate state space representations for problems (where appropriate) and implementation of various search algorithms that are appropriate to each represen tation. Evaluation of candidate representations and search strategies in terms of their appropriateness and efficiency.

Recurring Concepts: conceptual and formal models, consistency and completeness, efficiency, levels of abstraction, ordering in space, ordering in time.

Lecture Topics: (six hours minimum)

1. Problems and state spaces, knowledge representation
2. Basic control strategies (e.g., depth-first, breadth-first)
3. Forward and backward reasoning
4. Heuristic search (e.g., generate and test, hill climbing, breadth-first search means-ends analysis, graph, search, minimax search)

Suggested Laboratories:

1. (open) Implementation of several of the search strategies discussed in lectures, using a suitable AI language.
2. (closed) Observation of the behavior of several heuristic search implementations applied to a particular problem or a set of problems. The goal of this lab is to allow students to collect data on the performance of various heuristic search algorithms.

Connections:

- Related to: AL8, PL11
- Prerequisites: AL6, AI1
- Requisite for:

Last update:

Send comments to: webmaster@cs.wwc.edu

Arithmetic and Lists

Objectives

To learn how mathematical functions are defined in Prolog and to practice recursive programming on list structures in logic and functional programming.

Background

The syntax of mathematical expressions in Prolog is similar to that found in other programming languages. However, mathematical expressions are treated as symbolic expressions. For example, the function *f(x) = 3x + 4* might be written in relational form as:

```
f(X,3*X + 4).
```

The query

```
?- f(3,Y).
```

returns Y = 3*3+4 rather than 13 as might be expected. To force evaluation of the expression, the predicate `is` is used as follows:

```
f(X,Y) :- Y is 3*X + 4.
```

Here are some examples of list representation, the first is the empty list.

```
Pair Syntax          Element Syntax
[ ]                     [ ]
[a|[ ]]                 [a]
[a|b|[ ]]               [a,b]
[a|X]                   [a|X]
[a|b|X]                 [a,b|X]
```

Predicates on lists are often written using multiple rules. One rule for the empty list (the base case) and a second rule for non empty lists. For example, here is the definition of the predicate for the length of a list.

```
length([],0).
length([H|T],N) :- length(T,M), N is M+1.
```

Assignment

Functions:

1. Define the factorial function

```
n ! =   1       if n = 0
        n(n-1)  if n > 0
```

2. Define the Fibonacii function

```
fib(n) =        1                       if n = 0
                1                       if n = 1
                fib(n-1) + fib(n-2)     if n > 1
```

3. Define Ackerman's function

```
f(m, n) =       n+1             if m = 0
                f(m-1, 1)       if m > 0 & n = 0
                f(m-1, f(m,n-1) if m > 0 & n > 0
```

4. Define the function which computes the distance between two points in the cartesian plane

```
DP = sqrt(x_2 - x_1)^2 + (y_2 - y_1)^2)
```

5. Define a function which implements the quadratic formula. It should return a list containing the two solutions.

```
x = (-b +- sqrt(b^2 - 4ac)/2a
```

6. Define and test a function to find the surface area of a box given the length(l), height(h), and width(w).
7. Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

Numeric lists: For the following exercises let *S = [a_1,...,a_n]*.

1. The function `sum` where `sum(`*S*`) =` *a_1+...+a_n*.
2. The function `prod` where `prod(`*S*`) =` *a_1*...*a_n*.
3. The function `max` where `max(S)`is the maximum element in S.
4. The function `min` where `min(S)`is the minimum element in S.
5. The function `mean` where `mean(S,n)=` $(sum_{i=1..n} a_i)/n$
6. The function `interval` where `interval(m,n)}` is the list [m, m+1,

m+2, ..., n].
7. The function interval redefined so that when m>n the sequence is the list [m, m-1, m-2,..., n].

Polymorphic Lists

For the following exercises let *S = [a_1,...,a_m]* and *T = [b_1,...,b_n]*.

1. The function hd where hd(S)is *a_1*.
2. The function tl where tl(S)is the list *[a_2,...,a_m]*.
3. The function last where last(S)is *a_m*.
4. The function init where init(S)is the list *[a_1,...,a_{m-1}]*.
5. The function ithelmt where {ithelmt}(*S,i)* = *a_i*}.
6. The function subseq where subseq(S,i,j)is *[a_i,...,a_j]*.
7. The function drop where drop(I,S)is the list *[a_{i+1},...,a_m]*.
8. The function take where take(I,S)is the list *[a_{1},...,a_i]*.
9. The function append2 where {append2}(*S,T*) is the list [*a_1,...,a_m,b_1,...,b_n]*}.
10. Let *S = [S_1,...,S_n]* where each *S_i* is a list. The function append where {append}(*S)*is the result of concatinating the elements of *S*.
11. The function zip2 where {zip2}(*S,T)* = *[[a_1,b_1],...,[a_n,b_n]]*}.
12. Let *S = [[a_1,b_1],...,[a_n,b_n]]*. The function assoc where {assoc}(*a_i,S)* = *b_i*}.
13. The function length where length(S)is *m*.
14. The function member where {member}(*x,S)*is **true** iff *x* is an element of the sequence *S*.
15. The function mkset where mkset(S)is the list *S* without any duplicates. That is, it is a set.
16. The function union where union(S,T)is the list consisting of the set union of lists *S* and *T*.
17. The function intersection where intersection(S,T)is the list consisting of the set intersection of lists *S* and *T*.
18. The function difference where difference(S,T)is the list consisting of the set difference of lists *S* and *T*.

Sorting:

1. The function insert where insert(X,L)is the result of inserting X into the ordered sequence L.
2. The function merge where merge(L1,L2)is the sorted sequence resulting from the merging two sorted sequences.
3. The function partition where partition(X,L)is the pair of lists

[LT,GT], such that LT is the list of elements of L < X and the second list is the list of elements of L >= X.
4. The function isort where isort(L)is the list L} sorted. Use insertion sort as your sorting model.
5. The function bsort where bsort(L)is the list L} sorted. Use bubble sort as your sorting model.
6. The function ssort where ssort(L)is the list L} sorted. Use selection sort as your sorting model.
7. The function msort where msort(L)is the list L sorted. Use mergesort as your sorting model.
8. The function qsort where qsort(L)is the list L sorted. Use quicksort as your sorting model.

Miscellaneous List Exercises: For the following exercises let *S* = *[a_1,...,a_n]*.

1. The function limit where limit(S)is the first value in the list L which is the same as its successor.
2. The function delete(X,S)is the list *S* with all top level occurences of Xdeleted.
3. The function replicate where replicate(N,V)is the list consisting of N instances of V.
4. The function subst where {subst}(*x,y,S)*is like *S* except all ``top level'' occurances of *y* have been replaced by *x*.
5. The function reverse(S) where reverse(S)is the list *[a_m,...,a_1]*.
6. The function pal where pal(S)is true iff *S* is a palindrome.
7. The function digits where digits(N)is a list of the digits of the integer N.

Last update:

Send comments to: webmaster@cs.wwc.edu

Second-Order Predicates

Higher-Order Functions

Objectives

To study the second-order predicates provided in Prolog and higher-order functions and to construct other second-order predicates and higher-order functions.

Background

Pure Prolog is an attempt to implement first-order logic. Only the arguments of predicates may be quantified. Second-order logic permits quantification of predicates.

setof(X,P,Set) Set is the set of all X such that P is provable. To prevent backtracking the only free variables of P should be those appearing in X. If X is a free variable appearing in Q then the expression X^Q binds X and the expression is read as there exists an X such that Q.
bagof(X,P,Bag) Same as setof/3 but Bag is returned unordered and may contain duplicates.
findall(X,P,L) Similar to bagof/3 except that the variables of P not occuring in X are treated as local.

Assignment

For the following exercises let S be the list [S_1,...,S_n].

1. What is the value of ``answer'' given the following definitions?

```
answer = twice twice twice suc 0
twice f x = f(f x)
suc x = x + 1
```

2. The function `map` where `map(f,S)` is [f(S_1),...,f(S_n)].
3. The function `filter` where `filter(P,S)` is the list of elements of S that satisfy the predicate P.
4. The function `foldl` where `foldl(Op,In,S)` which folds up S, using the given binary operator Op and start value In, in a left associative way, ie, foldl(op, r,[a,b,c]) = (((r op a) op b) op c).
5. The function `foldr` where `foldr(Op,In,S)` which folds up S, using the given binary

operator Op and start value In, in a right associative way, ie, foldr(op,r,[a,b,c]) = a op (b op (c op r)).

6. The function `map2` is similar to `map`, but takes a function of two arguments, and maps it along two argument lists.
7. The function `scan` where `scan(op, r, S)` applies `foldl op r` to every initial segment of a list. For example `scan (+) 0 x)` computes running sums.
8. The function `dropwhile` where `dropwhile(P,S)` which returns the suffix of S where each element of the prefex satisfies the predicate P.
9. The function `takewhile` where `takewhile(P,S)` returns the list of initial element of S which satisfy P.
10. The function `until` where `until(P,F,V)` returns the result of applying the function F to the value the smallest number of times necessary to satisfy the predicate. Example until (>1000) (2*) 1 = 1024
11. The function `iterate` where `iterate(f,x)` returns the infinite list [x, f x, f(f x), ...]
12. Use the function `foldr` to define the functions, sum, product and reverse.
13. Write a generic sort program, it should take a comparison function as a parameter.
14. Write a generic transitive closure program, it should take a binary relation as a parameter.

Last update:

Send comments to: webmaster@cs.wwc.edu

Final Exam Information

The final exam will cover the following topics:

- Programming in Prolog
- Context-free grammar (Develop one for HTML)
- Push-down automata
- Regular expressions, Linear grammars
- Finite state machines
- Virtual machines for Fortran, C, and Pascal
- Compilers: scaning, parsing, and code generation
- Machine level implementation of simple and compound data types
- Run-time storage management (static, stack and heap)

96.3.8

Parameter Passing Exercises

1. Write a C program with a subroutine which implements a definite integration algorithm such as the trapezoidal rule or Simpson's rule. The subroutine must be able to be used with any function i.e. the function must be passed as a parameter.
2.

Logic Programming Exercises

This chapter contains several sets of exercises. They are designed to help a novice learn how to program in a new programming language. Not all the exercises are suitable for all programming languages.

Logic Programming Exercises

Prolog

1. Construct rules for searching a binary search tree to determine if an item is present in the tree.
2. Construct rules for returning a list of the elements of a binary search tree resulting from an inorder traversal.

Graph Algorithms

1. Implement Dijkstra's solution to the single source shortest paths problem.
2. Implement Floyd's solution to the all pairs shortest paths problem.
3.

Miscellaneous Exercises

The exercises in this section are designed to assist the student in learning a programming. While the exercises are couched in functional terms, they may written as procedures or predicates.

Object-Oriented Programming

Inheritance

1. Define an abstract data type - set which inherits list operations.
2. Define an abstract data type - binary tree and an abstract data type - binary search tree where the binary search tree inherits operations from the binary tree.

Meta-Programming

Meta-Interpreter

1. Extend the meta interpreter to handle full Prolog. You will need to add rules to handle the built in predicates, the cut symbol, the setof predicate and not.

2. Modify the meta interpreter so that it performs, breadth first search rather than depth first search.
3. Modify the meta interpreter so that it performs, bottom up proofs, i.e., it starts with the facts and produces all possible deductions.
4. Extend the proof version of the meta interpreter to handle full Prolog. You will need to add rules to handle the built in predicates, the cut symbol, the setof predicate and not.
5. Extend the trace program so that it will trace only those predicates which have been previously selected as spy points. You will need to add a predicate {\tt spy} of one argument which asserts the predicate {\tt spypoint} of one argument into the data base. The argument is the functor of the predicate to be traced. You will also need to modify the downprint and upprint routines so that printing is done only when the predicate is a spypoint.
6. Extend the trace program so that it will not only trace only the spy points but also indicates the depth of the trace.
7. Extend the trace program so that it will work for full Prolog. You will need to add rules to handle the built in predicates, the cut symbol, the setof predicate and not.

Interpreter

1. Complete the interpreter for the programming language Simple.

Compiler

1. Complete the compiler for the programming language Simple.
2. Complete the stack code interpreter.

Parsing

1. Add the additional grammar rules so that the topdown parser can parse programs in Simple.
2. Add the additional grammar rules so that the bottom up parser can parse programs in Simple.

Infinite Data Structures

Prolog

1. Construct a generator for the natural numbers.
2. Construct a generator for squares of natural numbers.
3. Construct a generator for prime numbers (use the sieve).
4. Construct a generator for perfect numbers.
5. Construct a generator for the arithmetic series: [n,m,n+m-n,...]. Make sure that it works for m $<$ n as well as for m $>$ n.
6. Construct a generator for the elements of a binary search tree where the elements are generated through an in-order traversal.

Natural Language Processing

The language
: Define a context-free grammar for a simple database.

Parser
: Using the grammar, construct a DCG for parsing and generating strings in the langauge.

Grade Sheet

Assignment	Beeson	Bertschy	Buchheim	Cromwell	Hanson	Mueller	Reihardt	Sanders	Springer	Vliet
Chapter Review: Intro										
Chapter Review: Syntax										
Chapter Review: Semantics										
Chapter Review: Compiler										
Exercises: Chapter 1										
Exercises Chapter 2										
Exercises Chapter 3										
Exercises Chapter 4										
Paper										
Test 1										
Test 2										
Totals										

CPTR 350 Computer Architecture -- 4

Syllabus

Lecture Notes

Preface - 1/2

Week	Topic	Chapter	Assignment	Lab
1	Introduction - 1 1/2	1	1.1-1.52	Digital Logic Library
1,2	Performance	2	2.1-2.27; 2.32-2.33	Mips Assembly
	THE CPU			
2,3	The Instruction Set	3	3.1-3.24 (3.25-3.38 ec)	Compilation
	DATA PATH AND CONTROL			
4,5	Arithmetic	4		Digital Logic
6,7	The Processor	5		CPU Simulation
7,8	Pipelining	6		CPU Simulation
9,10	MEMORY and I/O			
	Memory Hierarchy	7		CPU Simulation
	Input/Output	8		
	Parallel Processors	9		

Projects

- Hardware Project Design and implement a CPU in software.
- OS Project Construct a multitasking executive for some system.
- Retargetable Assembler Project Construct an easily retargetable assembler.
- Retargetable Compiler Project Construct an easily retargetable compiler.
- Other Projects Construct an easily retargetable compiler.
- Final Information

95.11.22 a. aaby

CPTR350 Computer Architecture

Winter 96

Goals

Upon completion of the course you will

- be familiar with computer abstractions and technology,
- be familiar with complete and reliable methods for measuring computer performance,
- be familiar with the computer arithmetic,
- be familiar with the design and implementation of microprocessors,
- have designed and simulated a CPU,
- will have retargeted a compiler or designed an OS executive for the MIPS processor.

Description

Study of the organization and architecture of computer systems with emphasis on the classical von Newmann architecture. Topics include instruction processing, addressing, interrupt structures, memory management, microprogramming, procedure call implementation, and multiprocessing. Laboratory work required. Prerequisites: CPTR 215 and ENGR 354.

The three weekly class times will follow a seminar format with assigned readings, discussion and in class presentations by students on the readings and solutions to assigned problems. Students will be expected to work together to solve the assigned problems.

The weekly lab time will be used to work on the projects, the design and implementation of a CPU and its supporting simulation environmentand either retargeting a compiler or the design and implementation of an OS executive. There will be 3 to 5 students per team.

You can expect to put in 6-9 hours per week for the class (including scheduled class meeting times) and an additional 3-4 hours per week for the laboratory.

Resources

Textbook:

Paterson & Hennessy, (1993) *Computer Organization & Design: The hardware/software interface* Morgan Kaufmann (MIPS)

Other Books:

Stallings, William. (1996) *Computer Organization and Architecture: Designing for*

Performance Prentice-Hall (PowerPC, Pentium)
Maccabe, A. B. (1993) *Computer Systems: Architecture, Organization, and Programming* Irwin (SPARC)
Kogge, P. M. (1991) *The Architecture of Symbolic Computers* McGraw-Hill (Alternative)
Wilhelm, R. & Maurer, D., (1995) *Compiler Design* Addison-Wesley (Abstract machines)
Wilkes, M.V. (1995) *Computing Perspectives* Morgan Kaufmann
Usenet: comp.arch and other CPU specific news groups
Technical Journals: Journal of the ACM; ACM sigs: SIGArch, SIGOP, SIGPLAN

Evaluation

The course grade is determined by the quantity and quality of work completed on the project and the assigned homework problems.

Grading Weights	**Letter Grades**
Problems: 25%	A: 90 ~ 100%
Projects: 75%	B: 80 ~ 89%
	C: 70 ~ 79%
	D: 60 ~ 69%

Each student will participate in the evaluation process.

95.6.5 a.aaby

Preface

Computer architecture is instruction set design and computer organization is instruction set implementation.

Why study computer architecture?

- To appreciate the organizational paradigms that determine the
  - capabilities,
  - performance, and,
  - success

  of computer systems.
- To understand the interaction between hardware and software.
- To develop a framework for understanding the fundamentals of computing.

Understand the interdependencies between

- assembly langauge,
- computer organization, and
- computer design.

This course provides an essential background for

- compiler compiler writers,
- operating system designers,
- database programmers, and
- hardware designers who must understand clearly the effects of their work on software applications.

96.1.2 a. aaby

Computer Abstractions and Technology

Introduction

- Economic implications: 5%-10% of the gross national product.
- Social implications: travel coast to coast in 30 seconds for 50 cents. Revolutions: agriculture, industrial, information
- Recent advances: teller machines, automobiles, laptop computers, human genome project
- Future:
- Performance: small memory (programming tricks), slow memory

Below the program

- Numbers for both programs and data
- Assembler
- HLL
    - allow programmer to think in a more natural language
    - improved productivity
    - hardware independence
- Evolution of the OS: I/O routines, supervisor routines
- Virtual machines: hardware, instruction set, systems software, application software

Under the covers

- I/O devices: mouse, crt
- Motherboard: IC chips, memory, bus, cpu, I/O controllers
- CPU: data path, control
- Computer: control (tells data path, memory, and I/O devices what to do) , datapath (performs arithmetic and logic operations), memory, input, output
- Secondary storage
- Communication

Integrated Circuits

- transistor
- VLSI
- silicon
- DRAM: increase in capacity NOT SPEED
- Silicon ingot, wafer, doping, flaws, yield

Falicies and Pitfalls

95.11.22 a. aaby

Performance

1. What methods have been developed for measuring computer performance?
2. What are the are the pros and cons of each method?
3. What methods are effective for measuring relative performance within a processor line?
4. What methods are effective for measuring relative performance across processors?

Definitions

Response time
: the time between the start and the completion of a task

Throughput
: the total amount of work done per unit of time

Performance
: reciprocal of the execution time (Performance = 1/Execution time)

Elapsed time
: wall-clock time from start to finish

CPU time
: the time the CPU spent computing the task -- user and system (CPU time = CPU clock cycles * Clock cycle time)

Clock period
: time for a clock cycle (e.g., 10 nanoseconds)

Clock rate
: inverse of clock period (R = 1/P) (e.g., 100 MHz)

Bandwidth requirements for various peripheral technologies (Stallings p. 40)

Peripheral	**Technology**	**Required bandwidth**
Graphics	24-bit color	30 MBytes/sec
LAN	100BASEX or FDDI	12 MBytes/sec
Disk Controller	SCSI or P1394	10 MBytes/sec
Full-motion video	1024x768@30fps	67+ MBytes/sec
I/O peripherals	miscellaneous	5+ MBytes/sec

Chips

	P5	**P6**	**PPC**	**Alpha 21066/A**	**Alpha 21064**	**Alpha 21164**
Clock Mhz				233/275/300	250/300	

SPECint92	/189/	277.1/341.4
SPECfp92	/264/	410.4/512.9
MIPS	/555/	

95.11.22 a. aaby

Instruction Set

MIPS assembler

Find an instruction set that makes it easy to build the hardware and compiler while maximizing performance and minimizing cost.

Definitions

- Assembly language: symbolic representation
- Machine language: mumerical representation
- Assembly: translation from symbolic to numerical
- Assembler: the program that performs the assembly
- Loader: the program that places the machine language in memory
- Linker: fixes the cross references between separately assembled modules

Instruction format

- R-type: 32 bits = op code(6), dst reg(5), src reg(5), dst reg(5), shift amt(5), function(6)
- I-type: 32 bits = op code(6), idx reg(5), src/dst reg(5), address(16)

Arithmetic Expressions

The operands are registers of which there are 32.

- add a, b, c # R[a] := R[b] + R[c]
- sub a, b, c # R[a] := R[b] - R[c]

Data transfer/Assignment

Word = 4 bytes; register 0 always contains 0 i.e., it cannot be changed.

- Load Word: lw a, b(c) # R[a] := M[b+R[c]]
- Move: move a, b # R[a] := R[0] + R[b]
- Store Word: lw a, b(c) # M[b+R[c]] := R[a]

byte and half-word load and store instructions are also avaliable.

Test & Branch instructions

- Branch on equal: beq a, b, L # PC := if R[a]=R[b] then L
- Branch on not equal: bne a, b, L # PC := if R[a]!=R[b] then L
- Jump: j L # PC := L
- Set on less than: slt a, b, c # R[a] := if R[b] < R[c] then 1 else 0
- Branch on less than: blt a, b, L # implemented by assember with slt and bne
- Jump register: jr a # PC := R[a] (used for case/switch statement)

Subroutines

- Call: jal Address # R[31] := PC; PC := Address
- Return: jr $31 # PC := R[31]

Compiler Issues

- Register allocation
- Register spilling
- Implementation of
    - Conditional statement
    - Case statement
    - While statement
    - Repeat-until statement

95.11.22 a. aaby

Pipelining

Pipelined Datapath

Data Hazards

Branch Hazards

95.11.22 a. aaby

Memory Hierarchy

Caches

Virtual Memory

95.12.10 a. aaby

Interfacing Processors and Peripherals

IO Performance

Characteristics of I/O Devices

Busses

Interfacing

95.11.22 a. aaby

Instruction Set Project

Construct an OS executive to run on the MIPS simulator.

This project constitutes 25% of your grade.

Phase 1

Phase 2

95.12.9 a. aaby

Universal Assembler Project

Construct an universal assembler.

This project constitutes 25% of your grade.

Phase 1

Phase 2

95.12.9 a. aaby

Instruction Set Project

Retarget a compiler to produce MIPS assembly code.

This project constitutes 25% of your grade.

Phase 1

Phase 2

95.12.9 a. aaby

CPTR 350 Final Information

- Evaluate your performance in this class based on the *Grade Expectations* and place your name and estimate of your grade on the form.
- Evaluate your contribution to the programming project based on the *Program grading criteria* place your name and estimate of your grade on the form.
-

96.3.8

CPTR 351 Memory and I/O -- 3

Syllabus

Lecture Notes

Week	Topic	Chapter	Assignment	Lab
	MEMORY and I/O			
	Memory Hierarchy	CO&D 7		
	Input/Output	CO&D 8		
	Device Drivers	KHG 1		
	Networking			

Projects

- Device Driver Project
  - Kernel Hacker's Guide
  - Kernel HOWTO
  - I/O Port Programming mini-HOWTO
  - Kerneld mini-HOWTO
  - Inline Assembly with DJGPP
  - DJGPP optimization
- Final Information

96.3.26 a. aaby

CPTR351 Memory and I/O -- 3

Spring 96

Goals

Upon completion of the course you will

Description

Study of interfacing techniques used in computer systems. Topics include random, semi-random, sequential, and direct-access methods; caching; synchronous and asynchronous transfer; and characteristics of I/O devices. Laboratory work required. Prerequisites: CPTR 142 and CPTR 350.

The three weekly class times will follow a seminar format with assigned readings, discussion and in class presentations by students on the readings and solutions to assigned problems. Students will be expected to work together to solve the assigned problems.

You can expect to put in 6-9 hours per week for the class (including scheduled class meeting times).

Resources

Textbook:

Paterson & Hennessy, (1993) *Computer Organization & Design: The hardware/software interface* Morgan Kaufmann (MIPS)

Johnson, Michael K. (1995) *The Linux Kernel Hacker's Guide* Linux Documentation Project

Other Books:

Stallings, William. (1996) *Computer Organization and Architecture: Designing for Performance* Prentice-Hall (PowerPC, Pentium)

Tanenbaum, Andrew S. (1987) *Operating Systems: Design and Implmentation* Prentice-Hall

Technical Journals: Journal of the ACM; ACM sigs: SIGArch, SIGOP, SIGPLAN

Web resources

- Linux Kernel Hacker's Guide
- Linux SCSI HOWTO
- FreeBSD Device Driver Writer's Guide
- IRIX Device Driver Programming Guide

Evaluation

The course grade is determined by the quantity and quality of work completed on the project and the assigned homework problems.

Grading Weights	**Letter Grades**
Problems: 25%	A: 90 ~ 100%
Projects: 75%	B: 80 ~ 89%
	C: 70 ~ 79%
	D: 60 ~ 69%

Each student will participate in the evaluation process.

96.3.26 a.aaby

Memory Hierarchy

Introduction

Programmers want an unlimited amount of fast memory. In this chapter we focus on techniques for creating the illusion of unlimited fast memory.
Motivation:

- Decreasing cost of memory: registers, SRAM, DRAM, DISK ...
- Decreasing speed of memory: registers, SRAM, DRAM, DISK ...

Principle of locality
programs access a relatively small portion of their address space at any instant of time.
- Temporal locality: if an item is referenced, it will tend to be referenced again soon.
- Spatial locality: if an item is referenced, items which are near by will tend to be referenced soon.

Memory hierarchy
SRAM, DRAM, disk, tape
- Data is copied between adjacent levels
- Minimum unit of information copied is a *block*
- If the requested data appears in some block in the upper level, this is called a *hit*, otherwise a *miss* and a block containing the requested data is copied from a lower level.
- The *hit rate* or hit ratio, is the fraction of memory accesses found in the upper level. The *miss rate* (1.0 - hit rate) is the fraction not found at the upper level.
- *Hit time*: the time to access the upper level including the time to determine if the access is a hit or a miss.
- *Miss penalty* the time to replace a block in the upper level.

Memory systems affect the operating system, compiler code generation, and applications. The memory system is a major factor in determining performance.

Caches

Cache: a safe place for hiding or storing things.
Motivation:

- high processor cycle speed
- low memory cycle speed
- fast access to recently used portions of a program's code and data

direct mapped
: address of the block *modulo* number of blocks in the cache.

tag
: contains the information to identify whether a word in the cache corresponds to the requested word.

valid bit
: indicates whether an entry contains a valid address

Handling Cache Misses

Instruction cache miss

1. Compute the value of PC-4 (PC was incremented before the miss was detected)
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.
4. Restart the instruction execution at the first step, which wil re-fetch the instruction, this time finding it in the cache.

A cache miss creates a *stall* which may be handled by stalling the entire machine while waiting for memory.

Data cache miss

: on a read, stall the processor until memory responds with the data.

on writes there is no need to stall, but the cache and memory become *inconsistent*. One solution is called *write-through* (data is written to both cache and memory possibly using a *write buffer* so memory writes do not cause stalls).

Spatial Locality

Increase block size to multiple words

For fixed cache size, increasing block size leads to increased miss costs and miss rate increases.

Memory design

- One word wide memory
- Multiple word wide memory
- Interleaved memory organization (less attractive as memory depth increases)

Cache Performance

CPU time = (CPU execution clock cycles + Memory-stall clock cycles) x Clock cycle time

Memory-stall clock cycles = Read-stall cycles + Write-stall cycles

Read-stall cycles = (Reads/Program) x Read miss rate x Read miss penalty

Write-stall cycles = ((Writes/Program) x Write miss rate x Write miss penalty) + Write buffer stalls

Virtual Memory

The technique in which main memory acts as a "cache" for the secondary storage is called *virtual memory*.
Motivation:

- allow efficient sharing of memory among multiple programs
- remove the programming burdens of a small, limited amount of main memory

overlays
: program segments loaded and unloaded under programmer control during program execution.

virtual memory
: automatically manages main memory and secondary storage.

address space
: programs are compiled into an address space; programs must be protected from each other -- virtual memory provides individual address space and protection.

page
: virtual memory block

page fault
: virtual memory miss

virtual address
: provided by the CPU is translated by hardware and software to a *physical address*

memory mapping or *address translation*
: often implemented by a table
: virtual address = virtual page number & offset
: physical address = page table[f(virtual page number)] + offset

relocation
: is facilitated by virtual memory.

virtual page number and *offset*
physical page number

Key decision in designing virtual memory

- Pages should be large enough to amortize the high access time (4-16KB with 64 KB under consideration)
- Flexible placement of pages helps to reduce page fault rate
- Misses (page faults) can be handled in software in clever algorithms

- Write through is not worth while.

Placing a page and finding it again

- *fully associative* mapping: a page may be placed anywhere in memory
- Each program has its own page table
- Hardware requirement: *page table register*
- Page table: valid bit and page pointer (p. 486)
- Context switch: save/restore page table, program counter and registers

Page faults

- Page faults cause exceptions to be generated
- VMS must keep track of the location on disk of each page in virtual address space
- Page replacement algorithm: LRU
    - *use bit* or *reference bit* set whenever a page is accessed
    - periodically clear use bits
    - Size of the page table
        - may be limited by use of a limit register
        - multiple page tables: stack, heap, code
        - hash function (*inverted page table*)
        - page the page tables

Writes

Write back: write only when necessary to replace the page and only if *dirty bit* is set (page has been modified).

Fast address translation: the TLB

translation-lookaside buffer: special address translation cache which holds only page table mappings. (p 492, 494)

Protection and virtual memory

Protection is provided if only the OS is permitted to modify the page tables.

1. Support at least two modes
2. Provide a protion of the CPU state that a user process can read but not modify: user/supervisor mode bit(s) and the page table pointer
3. Provide mechanisms whereby the CPU can go from user mode to supervisor mode and vice versa.

Shared pages are protected via read and write protection bits

Page faults and TLB misses

A TLB miss occurs when no entry in the TLB matches a virtual address. A TLB miss indicates one of two possiblilities:

1. The page is present in memory (page table's valid bit is on), and we need only to create the missing TLB entry.
2. The page is not present in memory (page table's valid bit is off), and we need to transfer control to the OS to deal with a page fault.

Common Framework for Memory Hierarchies

Where can a block be placed?

- Direct mapped (Block number modulo number of blocks in cache)
- ...
- Fully associative (Anywhere in physical memory)

n-way set associative: a block can be placed in any of *n* locations.

Cache is often direct mapped
TLB is often n-way set associative
VM is fully associative

How is a block found?

Fully associative and direct mapped: index
n-way set associative: use an index to find the set and then search the set

Which block should be replaced on a miss?

- Random
- Least recently used (LRU)

What happens on a write?

- Write through
    - read misses are cheaper
    - simpler to implement
- Write back
    - data is transfered at the rate the cache, rather than memory, can accept them
    - multiple writes require only one write to the lower level in the memory hierarchy
    - lower level has a high through-put

Concluding Remarks

- Processor-DRAM performance gap
- Multilevel cache (L1 affects clock rate of cpu; L2 affects miss penalty)
- DRAM improvements
- Compiler technology: restructure program to improve locality; prefetching

---

Interfacing Processors and Peripherals

Introduction

The difficulties in assessing and designing I/O systems have often relegated I/O to second class status. But I/O is important since

- Users interact with the system through I/O
- I/O performance is what distinguishes one class of computing systems from another.
- Amdahl's law: the performance of a system is determined by its's slowest component.

I/O systems design considerations

- expandability
- resilience in the face of failure
- performance

I/O performance issues

- access latency
- throughput
- depencency on
    - device characteristics
    - connection between device and rest of the system
    - the memory hierarchy
    - the operating system

Assessment of I/O performance often depends on the application.

- System throughput (system bandwidth) measured by either
    - data per unit of time (critical in super computer applications) or
    - I/O operations (often unrelated) per unit of time (critical in transaction processing)
- Response time (depends on)
    - bandwidth
    - latency

  most importance in single user systems.
- high throughput and short response time required in ATMS, airline reservation systems, order entry, inventory tracking, file servers, time sharing environments.

Conflicting priorities:

- minimize response time: handle a request as early as possible
- maximize throughput: service requests related by location first.

I/O Performance Measures

- Supercomputer I/O benchmarks: data throughput -- bytes/second
- Transaction processing benchmarks: I/O rate -- disk accessess/second
  Reliability in the face of failure is an absolute requirement and both response time and throughput are critical to building cost-effective systems.

Characteristics of I/O Devices

- Behavior: input (read once), output (write only) or storage (read,write)
- Partner: human or machine (at other end)
- Data rate: peak rate of data transfer

see table p. 540

Mouse

- optical
- mechanical

System monitors mouse status via *polling*

Keyboard

Magnetic Disk

- Platters
- Surfaces
- Tracks
- Sectors
- Cylinder
- Seek, seek time
- Rotational latency (rotational delay)
- Transfer time
- Controller, controller time

Scheduling Algorithms

- FCFS - first come, first served (unefficient, fair)
- SSTF - shortest seek time first (efficient, unfair)

- SCAN - efficient and fair
- C-SCAN - circular scan
- LOOK Scheduling - Look for a request in that direction before moving

Algorithm Selection

- Rotational time
- Hardware implementation of algorithms
- Raid technology and disk striping

RAID

- Level 1 - replication (twice as much storage)
- Level 2
- Level 3
- Level 4
- Level 5

Optical Memory

Magnetic Tape

Network

- Terminal network (RS232 standard) 0.3~19.2 Kbit/sec
- LAN (local area network) usually Ethernet 10 Mbit/sec
- Ethernet CSMA/CD; packet size: 64~1518 bytes sent a 0.1

Buses

Bus: a shared communication link consisting of

- a set of control lines and
- a set of data lines.

Typical bus transactions

- send an address then receive or transmit data
- a *read* transfers data from memory to the processor or an I/O device
- a *write* transfers data to the memory

Advantages, disadvantages and performance limitations

- advantage: simplicity

- disadvantage: creates a communication bottleneck limiting I/O throughput
- performance: bus speed is limited by physical factors: length of bus and number of devices

Types of Buses

processor-memory bus
: short, highspeed, and matched to the memory system to maximize processor-memory bandwidth; often design specific

I/O bus
: long, many types of devices and wide range of bandwidth; often usable in different machines

backplane bus
: allow processors, memory, and I/O devices to coexist on a single bus (motherboard=backplane); often usable in different machines

High-performance systems often use all three bus types

Synchronous and Asynchronous Buses

Synchronous

Synchronous buses are clocked and are simple and fast but are limited in length and number and types of devices.

- Clock included in control lines
- fixed communication protocol relative to the clock
- easily implemented in a small finite state machine
- fast and simple interface logic
- all devices must run at the same clock rate
- fast buses must be short because of clock skew

Asynchronous

Asynchronous buses are not clocked and follow a *handshaking protocol*. They scale better with technology changes and can support a wider variety of device response speeds.

Example: device requests a word of data from the memory system. There are three control lines:

1. *ReadReq:* used to indicate a read request for memory. The address is put on the data lines at the same time.
2. *DataRdy:* used to indicate that the data word is now on the data lines. The data is placed on the data lines at the same time.
3. *Ack:* used to acknowledge the ReadReq or the DataRdy signal of the other party.

Increasing the Bus Bandwidth

- *Data bus width*: transfer multiple words
- *Separate versus multiplexed address and data lines*
- *Block transfers*

split transaction protocol allows overlapping transactions

Obtaining access to the Bus

bus master controls access to the system. In a single bus master system the process is the bus master.

Bus arbitration

Bus arbitration is a scheme for deciding which bus master gets to use the bus. Typically there is a bus *arbiter* that decides on a basis of priority and fairness. There are four broad schemes

1. *Daisy chain arbitration* simple but unfair
2. *Centralized, parallel arbitration* central arbiter may become a bottleneck
3. *Distributed arbitration by self-selection* (NuBus - Apple Macintosh)
4. *Distributed arbitration by collision detection* (Ethernet)

Bus Standards

- IBM PC-AT bus -- IBM
- *intelligent peripheral interface* (IPI) -- manufacturers
- *small computer system interface* (SCSI) -- manufacturers; SCSI-2 20 or 40 Mbytes/sec (dazy chain)
- Ethernet -- manufacturers 10 Mb/s (asynchronous, distributed arbitration by collision detection)
- Fast Ethernet -- manufacturers 100 Mb/s (asynchronous, distributed arbitration by collision detection)
- Futurebus+ -- IEEE; no technology imposed limit (high-performance asynchronous, centralized and distributed arbitration)
- *peripheral component interconect* (PCI) -- Intel; 264 Mbytes/sec (syncronous timing, centralized arbitration scheme)
- *fiber distributed data interface* (FDDI) -- 100 Mb/s
- P1394 Serial Bus -- ANSI; 25 to 400 Mb/sec (dasy chain or tree structure - upto 63 devices)

Interfacing

Giving commands to the I/O devices

Communicating with the processor

Transferring the data between a device and memory

Direct memory access (DMA) and the memory system

Device Drivers

Introduction

Character Device Drivers

Block Device Drivers

SCSI Device Drivers

Writing Device Drivers

96.3.26 a. aaby

Last Modified - .

Computing Programs at WWC

evaluation & proposals

Anthony Aaby

The online version of this document (http://cs.wwc.edu/~aabyan/Local/computing.html) contains links to professional organizations and accrediting organizations.

Abstract: A refocused CIS program and the addition of one faculty position in software engineering would allow the creation of a CIS concentration in the MBA program, a degree in software engineering and the creation of a new program in information science and technology.

1 Introduction

The recent the proposal by Southern Adventist University for the creation of the Adventist College of Computing, their promotion of their computing program and the discussions between Computer Science and Engineering regarding a software engineering concentration for the BSE degree are the most recent events prompting this paper. However, there are good reasons for reviewing WWC's computing programs and planning for the future.

Computing is all pervasive and there are severe shortages of computer literate employees in the job market. It is estimated that the number of software engineers with go from about 800,000 in 1996 to 3,000,000 in 2005. While the number of traditional engineers will remain at about 2,000,000. The internet economy has created whole new categories of jobs.

In the various sections the WWC CIS, CS, & CpE programs are compared with the recommendations of professional societies and accrediting organizations, "holes" in the curriculum are pointed out , recommendations are made for significant change, and suggestions are made for ways that WWC can take advantage of emerging opportunities.

Sections 2, 3, and 4, compare WWC's CIS, CS, and CpE programs with the curricular recommendations of accrediting organizations and professional societies and point out areas where efficiencies or changes could be made. Section 5 contains descriptions of emerging computing related disciplines as possible programs, CIS concentration in the MBA program, software engineering, computing infrastructure, and information science and technology. Some recommendations for WWC are presented in Section 6.

2 CIS - Computer Information Systems

The School of Business offers a BBA degree with a concentration in CIS and a BS degree with a major in CIS. Several professional organizations (ACM, AIS, & AITP) have constructed a recommended curriculum. An abridged copy of their curricular requirements is available (IS2000). The degree programs are described in the bulletin. The current program is staffed at 2 fte.

Recommendations

Based on a comparison of IS2000 and the WWC BBA-CIS & BS-CIS programs,

- the curriculum should consist of 10 courses and
- discrete mathematics should be add as a cognate.

A 10 course program with a

- load of 9 courses per instructor/year would require 1.11 teaching fte and a
- load of 6 courses per instructor/year would require 1.7 teaching fte.

3 CS - Computer Science

The Computer Science Department offers a BA degree and a BS degree which has hardware, standard, & software options. The accrediting agency is the Computer Science Accreditation Board (CSAB). An abridged copy of their curricular requirements is available. The BS-CS major is described in the online bulletin. The current program is staffed at 2.5 fte plus a contract instructor for INFO105 and provides at least one section as a service course.

The current responsibility for INFO 105 Personal Computing is useful for maintaining student credit hours but is not part of the CS program.

Recommendations

Based on a comparison of CSAB Criteria 2000 and the BS-CS program, the computer science program should

- increase its science requirement by 4-8 hours and
- decrease the workload on instructors to the recommended loading of a maximum of 12 hours and *two preparations per term*. With a minimum of 60 hours required by CSAB, 2.5 teaching fte are necessary. A minimal program with fewer teaching fte and using courses from CIS, math, and engineering as electives is possible but would reduce the depth of the program.

In addition, the CS program should

- drop the hardware and software options,
- increase the science requirements,
- drop
    - CPTR 324 Scientific Computer Applications,
    - CPTR 351 Computer I/O, and
    - possibly CPTR 374 Simulation and Modeling,
- broaden electives by utilizing courses from engineering, math, and CIS (for example, numerical analysis could be an elective rather than a cognate), and
- reduce current teaching load to a maximum of two preparations per term.

4 CpE - Computer Engineering

The School of Engineering offers a concentration in computer engineering in the BSE degree. The current requirements are found in the WWC bulletin. The BSE program is accredited by ABET. A copy of ABET's curricular requirements, Engineering Criteria 2000 is available.

Recommendations

Based on a comparison of the ABET's curricular requirements, Engineering Criteria 2000 and the current concentration in computer engineering, the CpE program

- should find ways to provide more flexibility in the program by reducing the engineering core by 4-12 hours,
- examine all EE courses for content and prerequisites with the goal of shortening the prerequisite sequences and dependence upon ODE, and
- consider creating a computer and electrical engineering (CpEE) option.

5 Possible new programs

5.1 MS-IS (MBA-CIS)

A joint ACM/AIS task force has produced a MS IS Model Curriculum.

- The suggested MSIS program contains 13 courses.
- Three of the courses are part of the recommended CIS curriculum.
- Three additional courses appear to be advanced versions of other CIS courses and could be taught concurrently.
- The 7-10 courses above BBA-CIS program
    - require an additional 0.78-1.1 teaching fte carrying a load of 9 courses per instructor/year or

- require an additional 1.2-1.7 teaching fte carrying a load of 6 courses per instructor/year.

5.2 Software engineering (SE)

Software engineering is an emerging discipline. One estimate places the number of software engineers in 1996 at over 800,000 and estimates that by the year 2005 the number will swell to 3,000,000. For comparison, the number of all other engineers in 1996 is estimated at 1,300,000 with little change expected by the year 2005.

A copy of ABET's curricular requirements, Engineering Criteria 2000 is available.
If SE were to become a concentration of the BSE degree, WWC would have the only accredited software program in the denomination providing a strong counter balance to Southern's initiative. The CS program has already taken steps to strengthen its commitment to software engineering by adding a 400 level course in software engineering and two lower division courses with primary emphasis in software engineering: system software and programming, Object-oriented systems programming.

5.3 Computing Infrastructure

The technologies for telecommunications and computer networking are merging. Industry needs technically trained workers who have a basic understanding of

- electricity and electronics,
- computer hardware,
- telecommunications, and
- networking, and

who can assemble and maintain the telecommunications and computing infrastructure. An appropriate technical core includes:

- Electricity and electronics,
- Computer hardware,
- Telecommunications,
- Networking, and
- System design and trouble shooting.

Appropriate technical electives include

- programming, and
- system administration.

5.4 Information science and technology (IST)

Industry needs a larger base of information technology literate employees. The Pennsylvania State

University's program in information science and technology is an excellent example of how the academic world can respond to fill this need. A primary goal of the program should be to be an advocate for information technology across the curriculum and to provide service courses to other programs and departments. The traditional courses found in CIS and CS should be part of the curriculum but other courses unique to the department should be created. The program of the previous section, computing infrastructure could be a part of this program as well as programs for preparing students for careers as

- web masters and
- system and network administrators.

A proposal for a webmaster curriculum is available.

The IST programs curricula should include

- service courses,
- short courses.
- and interdisciplinary programs.

5.5 Organizational issues

1. Given the range of proposals, is a Southern Adventist University style School of Computing appropriate for computing at WWC?

The short answer is that the current political climate makes in impossible. The long answer is that the range of computing activities is so broad and some areas to closely tied to applications that it is unrealistic to combine all computing into a single school. With rapid change in computing likely to continue, the optimal placement of programs and the distribution of computing courses will require frequent study.

2. With the creation of the computer engineering concentration and the possibility of a software engineering concentration, should the Computer Science Department merge with the School of Engineering?

It is essential that those participating in the CS program maintain close contact the the EE instructors so, a merger could help to facilitate that communication. Recruiting for the CS program would be enhanced. Under a merger CS program would gain secretarial help for a variety of administrative activities. There would still be the need for some support for curriculum development and scheduling. With less responsibility for the chair, the chair could assume additional teaching duties. A separate operating and equipment budget would still be necessary for the maintenance of the program. There would be an increased workload for the School of Engineering.

The effect on the affiliation program needs to be considered.

3. What are the economies available through sharing instructors?

The economies depend on faculty qualifications and insuring the programs follow curricula recommendations.

- If the CIS curriculum were based on the recommendations in IS2000, it would be possible to provide
  - anywhere from 2-8 courses to be taught outside the program or made available for a graduate CIS program,
  - a common database course for CIS & CS programs,
  - a common software engineering/systems analysis and design course for CIS & CS, and
  - a common project management course for CIS & CS.
- If the previous recommendation were put in place, it would provide significant relief to the CS program permitting realistic teaching loads and elective choices.

6 Recommendations

6.1 BBA/BS-CIS, & MBA-CIS

The CIS program

- should be refocused to reflect the recommendations of IS2000.
- MATH 250 Discrete Mathematics should be add as a cognate.

It has the resources to provide both a BBA/BS-CIS degrees and a MBA-CIS concentration. To offer both programs requires between 1.89 and 3.5 teaching fte depending on teaching loads. Our current staffing level of 2.0 teaching fte. If an MBA-CIS concentration is a viable option, steps should be taken to begin it as soon as possible. If it is not a viable option, a refocused CIS program would have between 0.3 and 0.89 teaching fte available for reassignment either to other courses within the School of Business or to the IST program proposed below.

The CIS and CS programs should

- combine their database courses. The combined course should require discrete mathematics as a prerequisite. The should also
- explore the possibilities of common courses in
  - software engineering/systems analysis and design and in
  - user interface design.

6.2 BS-CS

The CS program should

- drop the hardware and software options,
- increase the science requirements,
- consider requiring three additional courses in application areas such as business, engineering, mathematics, or science,
- drop courses with limited appeal such as
    - CPTR 324 Scientific Computer Applications
    - CPTR 351 Computer I/O, and
    - possibly CPTR 374 Simulation and Modeling,
- drop numerical analysis as a cognate but retain it as an elective, and
- reduce current teaching load to a maximum of two preparations per term.

The CS and CIS programs should

- combine their database courses. The combined course should require discrete mathematics as a prerequisite. The should also
- explore the possibilities of common courses in
    - software engineering/systems analysis and design and in
    - user interface design.

6.3 BSE-CpE

The CpE program should

- should find ways to provide more flexibility in the program possibly by reducing the engineering core by 4-12 hours
- examine all EE courses for content and prerequisites with the goal of shortening the prerequisite sequence and dependence upon ODE.

If a reduced engineering core is possible, then consideration should be given to the creation of computer and electrical engineering concentration (CpEE) in addition to the EE and CpE concentrations.

The CS and CpE programs should

- continue close cooperation and
- the School of Engineering and the Department of Computer Science should determine if some sort of merger would be to their mutual benefit.

6.4 BS-SE or BSE-SE

A SE program housed either in the CS department or the School of Engineering should be started as soon as possible through the addition of one faculty member. The SE curriculum would be constructed in cooperation with the CS department.

6.5 IST

It is not hard to argue that a program in computing infrastructure should be housed in the department of technology. A concentration with the title "Telecommunications and Networking" would be appropriate. Other possible programs include:

- computer engineering technology
- telecommunications engineering technology

A new department of Information Science and Technology should be started with a minimum of two teaching fte. The one contract fte currently in the CS department should be part of this department. Any excess capacity in CIS, CS, or SE should be allocated to the department. The INFO prefix and courses should be assigned to this department. Appropriate CS, CIS, and GRPH courses could be cross listed (a collection of earlier ideas is available).

A curriculum for webmaster minor/majorcurriculum should be developed.

This program has the explosive growth potential.

Comments:

One concern I have with the idea of merging CS and Engr (which I have been thinking more positively about lately), is the affect this might have on our affiliation schools. They do not see Engr as competing with them very much, and so have supported affiliation. Since they have CS programs, though, they may find a combined department to be much more threatening, and start resisting our affiliation efforts. I'm not sure how this angle should be best explored, but I think it needs to be considered before we proceed too far.

-- Ralph

I think that it is important to consider affiliation angle as well and have added it to the document.

I suspect that WWC and Southern are the heavy weights in computing and other CS programs are going to be severely impacted by Southern's aggressive campaign.

A sort of counter argument is that Southern's computing program competes directly with WWC for students that would otherwise be CpE and possibly EE majors. I suspect that we have a major recruiting battle shaping up between WWC and Southern for that group of students.

thanks for your comments.

-- Anthony

I agree entirely about the threat Southern's campaign poses to our EE/CpE recruiting. When I saw Carlton's post of the quotes from their website, my immediate thought was that the CS/Engr merger ought to be thought about very soon. PUC, La Sierra, Union, etc., might take offense though.

If we can figure out a way to make our new strategy attractive to the affiliation schools, it might be even better, as they will be hurt by Southern's aggression also. It sounds (from the webpage) like Southern has decided to forget about the "cooperative" approach they were proposing, and just go for "king of the mountain" status.

-- Ralph

Ralph,

You have stated an important reason why we need to discuss this thing vigorously. I think it's urgent that we get further into this. If computer employment reaches 1.5 times traditional engineering by 2005, we have little time to loose. That means potentially 300 students on this campus.

Carlton

Collegedale, Tenn.¯ The president of Southern Adventist University has announced an 18-month strategic marketing initiative for its School of Computing that seeks to put it on the map. "We see an open window of opportunity," explained Dr. Gordon Bietz, Southern's president. "Our School of Computing has exceptionally well-qualified professors, five of whom have completed doctorates in the field," he said. "They have worked with the industry's leading high-tech companies and in the last five years they've been invited to speak at over 60 international conferences. In short, we have the product. Now prospective students here in Chattanooga and across the nation need to hear about it." The curriculum fills a recognized need, according to Dr. Timothy Korson, dean of the School of Computing. He adds that the U.S. Department of Labor has predicted the top three fastest-growing occupations to be computer scientist, computer engineer, and computer analyst. Southern Adventist University offers bachelor of science degrees in computer science, computer systems administration, and computer information systems. "And our new Master of Software Engineering (M.S.E.) degree competes with the best in the country," Dr. Korson says. "We're positioned for significant growth." Other advantages he describes are the Software Technology Center (STC) with its connections to high tech corporations and a highly successful intern program. The STC is the research center of the School of Computing and is sponsored in part by the Consortium for the Management of Emerging Software Technology (Comsoft). Comsoft is funded by major corporations such as AT&T, IBM, Spring, Allstate, and NBC. STC provides opportunities for students and faculty to work together researching emerging software technologies. It also offers employment for motivated students to work on advanced software development projects with major corporations. "Focusing on the computer science aspect of Southern's academic offerings really shines the spotlight on all of Southern's programs," says Dr. Bietz, and I'm pleased to see the School of Computing embarking on this marketing push to get the word out." The School of Computing may be reached at 423.238.2936 or www.cs.southern.edu .

Information Systems

IS 97, IS 2000

IS 1 Fundamentals of Information Systems
IS 2 Personal Productivity with IS Technology
IS 3 Information Systems: Theory and Practice
IS 4 Information Technology Hardware and Software
IS 5 Programming, Data Files and Object Structures
IS 6 Systems Analysis and Design
IS 7 Telecommunications & Networks
IS 8 Database Design & Implementation
IS 9 Database programming
IS 10 Project Management

Business Organization and Process
Principles of Accounting
Introduction to Business
Operations Management and Production
Management and Organizational Behavior
Organizational Change and Development
Economics
Marketing

Cognates
Survey of Calculus
Discrete Mathematics
Applied Statistics
IS.P0: Personal Computing

Communications
Speech Communications
College Writing, Research Writing
Business Communcation/Writing for the Professions

ISCC 99 - Information Systems-Centric Curriculum

ISCC 11 Information Systems 40-45 semester hours
ISCC 21 Information Systems Architecture I
ISCC 22 Computer Ethics I
ISCC 31 Information Systems Architecture II
ISCC 41 Information, Databases and Transaction Processing
ISCC 42 Human Computer Interaction Issues and Methods
ISCC 43 Telecommunications and Networking
ISCC 44 Dynamics of Change
ISCC 45 Applications of AI in Enterprise Systems
ISCC 51 Distributed Systems
ISCC 52 Computer Ethics II
ISCC 53 Comprehensive Enterprise Information Systems Engineering
ISCC 61 Comprehensive Collaborative Project

Technical Courses
Data warehousing & data mining
Automated decision making
Virtual reality in systems
Decision science
Organizational Behavior and Management
Microeconomics

Functional areas
accounting,
finance,
operations management,
marketing,
human resource management
Project management
Economics

Cognates
Psychology
Psychology of Groups
Probability and Statistics
Discrete Mathematics
English
Communications
Technical Writing
Social Science
Humanities
Business/Enterprise
A foreign language

MSIS Model curriculum and Guidelines for

Graduate Degree Programs in Information Systems

IS 1	Fundamentals of Information Systems
IS 4	Information Technology Hardware and Software
IS 5	Programming, Data Files and Object Structures
MSIS 1	Data Management
MSIS 2	Analysis, Modeling and Design
MSIS 3	Data Communications and Networking
MSIS 4	Project and Change Management
MSIS 5	IS Policy and Strategy
MSIS 6	Integration
MSIS 6.1	Integrating the Enterprise
MSIS 6.1	Integrating the IS Function
MSIS 6.1	Integrating IS Technologies
MSIS 6.1	Integrating the Enterprise, IS Function, and IS Technologies

CSAB - CSAC Criteria 2000

- abridged for local use only

CSAB - Computer science is a discipline that involves the understanding and design of computers and computational processes. In its most general form it is concerned with the understanding of information transfer and transformation. Particular interest is placed on making processes efficient and endowing them with some form of intelligence. The discipline ranges from theoretical studies of algorithms to practical problems of implementation in terms of computational hardware and software. A central focus is on processes for handling and manipulating information. Thus, the discipline spans both advancing the fundamental understanding of algorithms and information processes in general as well as the practical design of efficient reliable software and hardware to meet given specifications. ... As such it includes theoretical studies, experimental methods, and engineering design all in one discipline. This differs radically from most physical sciences that separate the understanding and advancement of the science from the applications of the science in fields of engineering design and implementation. In
computer science there is an inherent intermingling of the theoretical concepts of computability and algorithmic efficiency with the modern practical advancements in electronics that continue to stimulate advances in the discipline. It is this close interaction of the theoretical and design aspects of the field that binds them together into a single discipline.

Thus, a well educated computer scientist should be able to apply the fundamental concepts and techniques of computation, algorithms, and computer design to a specific design problem. The work includes detailing of specifications, analysis of the problem, and provides a design that functions as desired, has satisfactory performance, is reliable and maintainable, and meets desired cost criteria. Clearly, the computer scientist must not only have sufficient training in the computer science areas to be able to accomplish such tasks, but must also have a firm understanding in areas of mathematics and science, as well as a broad education in liberal studies to provide a basis for understanding the societal implications of the work being performed.

General

1. The curriculum must include at least 40 semester hours of ... computer science topics.
2. The curriculum must contain at least 30 semester hours of study in mathematics and science ....
3. The curriculum must include at least 30 semester hours of study in humanities, social sciences, arts and other disciplines....

Computer Science

1. All students must take a broad-based core of fundamental computer science material consisting of at least 16 semester hours.
2. The core materials must provide basic coverage of algorithms, data structures, software design, concepts of programming languages, and computer organization and architecture.
3. Theoretical foundations, problem analysis, and solution design must be stressed within the program's core materials.
4. Students must be exposed to a variety of programming languages and systems and must become proficient in at least one higher-level language.

   All students must take at least 16 semester hours of advanced course work in computer science that provides breadth and builds on the core to provide depth.

Mathematics and Science

1. The curriculum must include at least 15 semester hours of mathematics.Course work in mathematics must include discrete mathematics, differential and integral calculus, and probability and statistics.
2. The curriculum must include at least 12 semester hours of science.
3. Course work in science must include the equivalent of a two-semester sequence in a laboratory science for science or engineering majors.
4. Science course work additional to that specified in Standard IV-13 must be in science courses or courses that enhance the student's ability to apply the scientific method.

Additional Areas of Study

1. The oral communications skills of the student must be developed and applied in the program.
2. The written communications skills of the student must be developed and applied in the program.
3. There must be sufficient coverage of social and ethical implications of computing to give students an understanding of a broad range of issues in this area.

Last Modified - Fri Feb 18 11:32:22 2000.

Engineering Criteria 2000

PROGRAM CRITERIA FOR ELECTRICAL, **COMPUTER**, AND SIMILARLY NAMED **ENGINEERING** PROGRAMS

Submitted by The Institute of Electrical and Electronics Engineers, Inc.

These program criteria apply to engineering programs which include electrical, electronic, computer, or similar modifiers in their titles.

1. Curriculum

The structure of the curriculum must provide both breath and depth across the range of engineering topics implied by the title of the program.

The program must demonstrate that graduates have: knowledge of probability and statistics, including applications appropriate to the program name and objectives; knowledge of mathematics through differential and integral calculus, basic sciences, and engineering sciences necessary to analyze and design complex electrical and electronic devices, software, and systems containing hardware and software components, as appropriate to program objectives.

Programs containing the modifier "electrical" in the title must also demonstrate that graduates have a knowledge of advanced mathematics, typically including differential equations, linear algebra, complex variables, and discrete mathematics.

Programs containing the modifier "computer" in the title must have a knowledge of discrete mathematics.

PROPOSED PROGRAM CRITERIA FOR **SOFTWARE** AND SIMILARLY NAMED **ENGINEERING** PROGRAMS

Submitted by The Institute of Electrical and Electronics Engineers, Inc.

These program criteria apply to engineering programs which include software or similar modifiers in their titles.

1.Curriculum

The curriculum must provide both breadth and depth across the range of engineering and computer science topics implied by the title and objectives of the program.

The program must demonstrate that graduates have: the ability to analyze, design, verify, validate, implement, apply, and maintain software systems; the ability to appropriately apply discrete mathematics, probability and statistics, and relevant topics in computer and management sciences to complex software systems.

Last Modified - .

Proposal: Webmaster Curriculum

DRAFT COPY

Proposal: Create a interdisciplinary Webmaster program at WWC. The program could be housed in a new department -- Department of Information Science and Technology. The program could be offered as a minor, major, and/or an emphasis in the MBA program.

Details

Working with the World Organization of Webmasters (WOW), Prentice Hall PTR has developed two book series (interactive workbooks) that are designed to train Webmasters. This proposal is based on these books and the program at Merrimack College.

The following table lists the titles of the books in the series, appropriate departments to teach the corresponding course, an estimate of the credits, and the identification of a near equivalent course already available.

Foundations of Web Site Architecture			
Dept	**Topic**	**Credits**	WWC Equivalent
Graphics	Understanding Web Development	1-2	GRPH Web page design
CS/CIS	Administrating Web Servers, Security & Management	2-3	CPTR module CPTR module
Marketing Management	Exploring Web Marketing & Project Management	2-3	
Graphics COMM	Creating Web Graphics, Audio & Video		GPRH
Advanced Web Site Architecture			
Graphics	Designing Web Interfaces, Hypertext and Multimedia		GPRH
CS/CIS	Supporting Web Servers, Networking, Programming, & Emerging Technologies		CPTR module CPTR module
Business	Exploring Electronic Commerce, Site Management, & Internet Law		

The first three books in the series are available in my office - Anthony.

Laboratory experience

In addition to the courses, practical experience should be required. This may be achieved by the student's participation in the development and management of the various WWC web sites. This would require the participation of the WWC IS Department.

Additional considerations

The following table lists some additional courses that should be considered.

Additional Courses		
Dept	**Topic**	**Credits**
CS/CIS	Database: MS Access, MS SQL Server, MySQL, Oracle	1-2
CS/CIS/GRPH	Scripting languages MS-VB, JavaScript, Perl, PHP, Python	2-3
CS/CIS	MS Windows 2000 Intro to Unix	1-2
CS/CIS	MS Windows 2000 Administration Unix Administration	2-3

Additional work should be available in areas such as

- business
- communications
- graphics
- marketing
- organization and management

Several of the courses may be adapted from courses or modules in courses currently available. Initially no additional staff would be required.

Comments:

yes, this is exactly what I have been working around in my head, too. With a bit further study, it looks like we may already have in place that which would enable us to start this type of program. Is there a group of us who could meet at some point to get this thing off the ground?

-- Linda

Information Sciences and Technology

The future of IT at WWC

Masters in Information Technology (MIT)

- Masters Degree in Information Technologies
- Q&A re MS in ITs - presented to Grad Council 98.10.16

The Future: Information Technology Group

Abstract: *Based on developments at other academic institutions and the rapid penetration of computers into virtually all occupations, a proposal is advanced for the creation of an Information Technology Group to spearhead the creation of courses, shared laboratories, new majors and interdisciplinary programs.*

Curriculum

Course Selection Guide

Shared Labs

Questionnaire

- Program proposals
  - Information Science and Technology Minor
  - IST minor
  - IST minor memo
  - International Studies Major & Minor. Based on the general studies global perspective proposal below, a major of 48 hours and a minor of 27 hours. No more than 20 hours of foreign language.

Course Proposals

- Creative Problem Solving
- Project Management
- Human Factors Engineering/Human Computer Interaction

General Studies Proposals

- Global perspective
- Math

Last Modified - .

Send comments to webmaster@cs.wwc.edu

Software Engineering

Rationale

- Merger of ABET & CSAB
- Texas Board of Professional Engineers
- IEEE/ACM
- ABET - Criteria 2000 for SE
- Job market
- WWC Leadership

U.S. Bureau of Labor Statistics

In 1998, engineers held **1.5 million jobs**. The following tabulation shows the distribution of employment by engineering specialty.

Specialty	*Employment*	*Percent*
Total all engineers	1,462,000	100.0
Electrical and electronics	357,000	24
Mechanical	220,000	15
Civil	195,000	13
Industrial	126,000	9
Aerospace	53,000	4
Chemical	48,000	3
Materials	20,000	1
Petroleum	12,000	<1
Nuclear	12,000	<1
Mining	4,000	<1
All other engineers	415,000	28

Computer systems analysts, engineers, and scientists held about **1.5 million jobs** in 1998, including about 114,000 who were self-employed. Their employment was distributed among the following detailed occupations:

Computer systems analysts	617,000
Computer support specialists	429,000
Computer engineers	299,000
Database administrators	87,000
All other computer scientists	97,000

Computer Science and Related Degree Programs

Engineering	**Computer Science**	**Business**
BSE concentration in CpE	BS	BBA - CIS & BS - CIS
Engineering core	**Common core**	**Business core:** 67 cr. hrs
Intro. to Programming Data Structures and Algorithms Circuit Analysis Linear Network Analysis Physical Electronics & other courses	Introduction to Programming Data Structures and Algorithms Assembly Language Programming Programming Languages Design and Analysis of Algorithms Colloquium Seminar	**CIS core** Intro. Busn. App. Programming Interm. Busn. App. Programming Systems Analysis and Design Telecommunications

CpE Concentration	**Hardware option**	**Standard option**	**Software option**	**Systems Development**	**System Support**
Assembly Language Programming Prgramming Languages Computer Architecture Operating System Design Software Engineering Digital Logic Microprocessor Systems Design Engineering Electronics Digital Design and 12 cr. hrs. of electives	Computer Architecture Computer I/O Operating System Design Digital Logic Microprocessor Sys. Design Electives	Theory of Computation Digital Logic Computer Architecture Computer I/O Operating System Design Electives	Operating System Design or Parallel and Distributed Computation Systems Analysis and Design Database Management Systems Electives Application Domain (30 cr. hrs.)	Adv. Busn. App. Progmng Database Management Sys. Database Management App. Electives	Intro. Network Adm. Interm. Network Adm. Adv. Network Adm. Electives

Cognates	**Cognates**	**Cognates**	**Cognates**	**Cognates**
	Circuit Analysis Digital Design			Personal Computing
Calculus I-IV Discrete Mathematics Linear Algebra Probability and Statistics Ordinary Differential Equations General Chemistry Principles of Physics	Calculus I-IV Discrete Mathematics Linear Algebra Probability and Statistics Numerical Analysis Principles of Physics	Calculus I-IV Discrete Mathematics Linear Algebra Probability and Statistics Numerical Analysis Principles of Physics	Survey of Calculus Discrete Mathematics Linear Algebra Applied Statistics Fundamentals of Electronics	Survey of Calculus Business Statistics (part of the Business core) Psychology Speech Communications
Engineering General Studies	**General Studies - same for the three options**			**General Studies**

WWC CS-Dept. Last Modified - .

Send comments to webmaster@cs.wwc.edu

SE Definition

Encompasses

- theory,
- technology,
- practice and application of software in computer-based systems

Central theme

- to engender an engineering discipline in students,
- enabling them to define and use
  - processes,
  - models and
  - metrics in software and system development.

-- IEEE/ACM

SE Curriculum

Curriculum (IEEE/ACM) - approximately equal segments

- in software engineering,
- in computer science and engineering,
- in appropriate supporting areas, and
- in advanced materials.

Curriculum (ABET - Criteria 2000)

- breadth and depth across the range of engineering and computer science topics
- must demonstrate that graduates have: the ability to
    1. analyze,
    2. design,
    3. verify,
    4. validate,
    5. implement,
    6. apply, and
    7. maintain software systems;
- have the ability to appropriately apply
    1. discrete mathematics,
    2. probability and statistics, and
    3. relevant topics in computer and management sciences to complex software systems.

BSE - SE Concentration

Engineering Core Requirements

Identical with computer engineering except that **8 rather than 12 hours of chemistry** are required.

CONCENTRATION: Software Engineering(56 credits)

CPTR 143	Data Structures and Algorithms	4
CPTR 215	Assembly Language Programming	3
CPTR 316	Programming Paradigms	3-4
CPTR 350	Computer Architecture	4
CPTR 352	Operating System Design	4
CPTR 435	Software Engineering	4
CPTR 454	Design and Analysis of Algorithms	4
ENGR 354	Digital Logic	3
	Electives	26-27
CPTR 235	System Software and Programming	4
CPTR 245	Object-Oriented System Design	4
CPTR 345	Theory of Computation	4
CPTR 355	Computer Graphics	4
CPTR 374	Simulation and Modeling	3
CPTR 415	Introduction to Databases	4
CPTR 425	Introduction to Networking	4
CPTR 445	Intro to Artificial Intelligence	4
CPTR 460	Parallel and Distributed Computation	4
CPTR 464	Compiler Design	4
ENGR 355	Embedded System Design	3

BS - SE

Software engineering encompasses theory, technology, practice and application of software in computer-based systems. A central theme of the curriculum is to engender an engineering discipline in students, enabling them to define and use processes, models and metrics in software and system development. Combined with appropriate knowledge of an application domain, it is used to provide computer based solutions. For example, combined with business, it prepares you for a career in computer information systems. *Employment opportunities* are found throughout business, government, industry and research.

Computer Science and Engineering - 40 hours +

Include the areas of algorithms and data structures, computer architecture, databases, programming languages, operating systems, and networking. The computer science principles in these areas should be integrated and applied in advanced software engineering courses and projects.

CPTR 141	Introduction to Programming	4
CPTR 142, 143	Data Structures and Algorithms	4,4
CPTR 215	Assembly Language Programming	3
CPTR 221, 222	Programming Languages	3,3
ENGR 354	Digital Logic	4
CPTR 350	Computer Architecture	3
CPTR 351	Computer I/O	4
CPTR 352	Operating System Design	4
CPTR	Networking	4
CIS 440	Database Management Systems	4

Software Engineering - 40 hours +

Covers processes and techniques for developing and maintaining large systems. Courses should address the areas of requirements analysis, software architecture and design, testing and quality assurance, software management, selection and use of software tools and components, computer and human interaction, maintenance and documentation. Substantial design work must be included and the students must be exposed to a variety of languages and systems.

Engineering responsibility and practice must be stressed, which includes conveying ethical, social, legal, economic and safety issues. These concerns must be reinforced in advanced work, as must the appropriate use of software engineering standards. Students should also learn methods for technical and economic decision making, such as project planning and resource management. Additionally, students must achieve an understanding of the need for and an ability to engage in life-long learning.

CPTR 435	Software Engineering	4
CPTR 454	Design and Analysis of Algorithms	4

Advanced Areas

Providing depth in one or more areas. This part of the program may incorporate further study in the software engineering and computer science topics indicated above, may involve work in additional areas of theory or technology, and should include work in one or more significant application domains. Particular domains may require additional work in supporting areas such as mathematics and science.

CPTR 324	Scientific Computer Applications	4
CPTR 345	Theory of Computation	4
CPTR 355	Computer Graphics	4
CPTR 374	Simulation & Modeling	4
CPTR 445	Introduction to Artificial Intelligence	4
CPTR 460	Parallel & Distributed Computation	4
CPTR 464	Compiler Design	4
MATH 341	Numerical Analysis	4
MATH 351	Operations Research	4
MATH 442	Advanced Numerical Analysis	4

Supporting Areas - 40 hours +

Included are communications (oral, written, listening), including the abilities to work in teams, and mathematics focusing primarily on discrete mathematics and probability and statistics.

ENGL 121, 122, 223	Writing, Research Writing	9
SPCH 101	Fundamentals of Speech Communications	4
MATH 181, 281, 282	Calculus I, II, III	12
MATH 250	Discrete Mathematics	4
MATH 315	Probability and Statistics	4

"The practice of software engineering will mean a service or creative work such as analysis, design, or implementation of software systems, the adequate performance of which requires appropriate education, training or experience. Such education, training or experience shall include an acceptable combination of: computer sciences such as computer organization, algorithm analysis and design, data structures, concepts of programming languages, operating systems, and computer architecture; software design and architecture; discrete mathematics; embedded and real-time systems; or other engineering education. Such creative work will demonstrate the application of mathematical, engineering, physical or computer sciences to activities such as real-time and embedded systems; information or financial systems, user interfaces, and networks." Texas Board of Professional Engineers 2/4/1999

- IEEE-CS/ACM Education Task Force Accreditation Guidelines
- Software Engineering body of knowledge SWEBOK
- Working group on software engineering education and training
- Texas Board

Contact the *Computer Science Department* for more details.

Last update: a.aaby

Send comments to: webmaster@cs.wwc.edu

What is Programming?

- Art
- Craft
- Engineering discipline
- Science
- Mathematics

BS, BA, AS degrees - Computer Science Major

A proposal

The CS Body of Knowledge (topics) IEEE/ACM CC2001	Related WWC courses (recommended courses are in bold)			Representative texts & references
PF. Programming Fundamentals (65 core hours) PF1. Algorithms and problem-solving (8) PF2. Fundamental programming constructs (10) PF3. Basic data structures (12) PF4. Recursion (6) PF5. Abstract data types (9) PF6. Object-oriented programming (10) PF7. Event-driven and concurrent programming (4) PF8. Using modern APIs (6)	**CPTR 141** **CPTR 142** **CPTR 143** CIS 130 CIS 230 CIS 330 CPTR 245	**Intro to programming** **Data structures & algorithms** **Data structures & algorithms** Intro. business app. programming Interm. business app. programming Adv. business app. programming Object-oriented system design	**4** **4** **4** 4 4 4 4	APIs - MFC/STL/OpenGL/ODBC/CORBA
AL. Algorithms and Complexity (31 core hours) AL1. Basic algorithmic analysis (4) AL2. Algorithmic strategies (6) AL3. Fundamental computing algorithms (12) AL4. Distributed algorithms (3) AL5. Basic computability theory (6) AL6. The complexity classes P and NP AL7. Automata theory AL8. Advanced algorithmic analysis AL9. Cryptographic algorithms AL10. Geometric algorithms	**CPTR 454** CPTR 345	**Design & analysis of algorithms** Theory of Computation	**4** 4	

PL. Programming Languages (5 core hours) PL1. History and overview of programming languages (2) PL2. Virtual machines (1) PL3. Introduction to language translation (2) PL4. Language translation systems PL5. Type systems PL6. Models of execution control PL7. Declaration, modularity, and storage management PL8. Programming language semantics PL9. Functional programming paradigms PL10. Object-oriented programming paradigms PL11. Language-based constructs for parallelism	CPTR 316 CPTR 464	Programming Paradigms Compiler Design	4 4	
AR. Architecture (33 core hours) AR1. Digital logic and digital systems (3) AR2. Machine level representation of data (3) AR3. Assembly level machine organization (9) AR4. Memory system organization (5) AR5. I/O and communication (3) AR6. CPU implementation (10)	**CPTR 215** CPTR 350 ENGR 354 ENGR 355 ENGR 433 ENGR 434	**Assembly language programming** Computer architecture Digital logic Embedded system design Digital design VLSI design	**3** 4 **3** 3 4 4	
OS. Operating Systems (22 core hours) OS1. Operating system principles (2) OS2. Concurrency (6) OS3. Scheduling and dispatch (3) OS4. Virtual memory (3) OS5. Device management (2) OS6. Security and protection (3) OS7. File systems and naming (3) OS8. Real-time systems	CPTR 235 **CPTR 352**	System software & programming **Operating system design**	4 **4**	
HC. Human-Computer Interaction (3 core hours) HC1. Principles of HCI (3) HC2. Modeling the user HC3. Interaction HC4. Window management system design HC5. Help systems HC6. Evaluation techniques HC7. Computer-supported collaborative work	CPTR 235 CPTR 245 GRPH 263 INFO 250	System software & programming Object-oriented system design Webpage design & construction MFC programming	4 4 3 1	Fleming, Jenifer *Web Navigation: Designing the User Experience* O'Reilly 1998 Schneiderman, Ben *Designing the User Interface: Strategies for Effective Human-Computer Interaction* 3rd ed Addison-Wesley 1997 Norman, Donald *The Design of Everyday Things* Doubleday Books 1990

GR. Graphics (no core hours) GR1. Graphic systems GR2. Fundamental techniques in graphics GR3. Basic rendering GR4. Basic geometric modeling GR5. Visualization GR6. Virtual reality GR7. Computer animation GR8. Advanced rendering GR9. Advanced geometric modeling GR10. Multimedia data technologies GR11. Compression and decompression GR12. Multimedia applications and content authoring GR13. Multimedia servers and filesystems GR14. Networked and distributed multimedia systems	CPTR 355	Computer graphics	4
IS. Intelligent Systems (10 core hours) IS1. Fundamental issues in intelligent systems (2) IS2. Search and optimization methods (4) IS3. Knowledge representation and reasoning (4) IS4. Learning IS5. Agents IS6. Computer vision IS7. Natural language processing IS8. Pattern recognition IS9. Advanced machine learning IS10. Robotics IS11. Knowledge-based systems IS12. Neural networks IS13. Genetic algorithms	CPTR 445	Introduction to artificial intelligence	4
IM. Information Management (10 core hours) IM1. Database systems (2) IM2. Data modeling and the relational model (8) IM3. Database query languages IM4. Relational database design IM5. Transaction processing IM6. Distributed databases IM7. Advanced relational database design IM8. Physical database design	INFO 150 CIS 240 CIS 440 **CPTR 235** CPTR 415	MS Access Intermediate business applications Database management systems **System software & programming** Introduction to databases	1 4 4 **4** 4

NC. Net-Centric Computing (15 core hours) NC1. Introduction to net-centric computing (9) NC2. The web as an example of client-server computing (6) NC3. Building web applications NC4. Communication and networking NC5. Distributed object systems NC6. Collaboration technology and groupware NC7. Distributed operating systems NC8. Distributed systems	CIS 290 CIS 350 CIS 390 CIS 490 CIS 330 CIS 489 **CPTR 235** CPTR 425 CPTR 460	Introduction to network administration Telecommunications Intermediate network administration Advanced network administration Adv. business app. programming Integrated systems development **System software & programming** Introduction to networking Parallel & distributed computation	4 4 4 4 4 **4** 4 4	
SE. Software Engineering (30 core hours) SE1. Software processes and metrics (6) SE2. Software requirements and specifications (6) SE3. Software design and implementation (6) SE4. Verification and validation (6) SE5. Software tools and environments (3) SE6. Software project methodologies (3)	**CPTR 435** CPTR 245 CIS 315 CIS 489	**Software engineering** Object-oriented system design Systems analysis & design Integrated systems development	**4** 4 4 4	Sommerville, Ian *Software Engineering, 6e* Addison-Wesley 2000 Beck, Kent *Extreme Programming Explained* Addison-Wesley 2000
CN. Computational Science (no core hours) CN1. Numerical analysis CN2. Scientific visualization CN3. Architecture for scientific computing CN4. Programming for parallel architectures CN5. Applications	CPTR 324 CPTR 435 CPTR 460 ENGR 468 MATH 341 MATH 351 MATH 442	Scientific computer applications Simulation & modeling Parallel & distributed computation Engineering finite element methods Numerical analysis Operations research Advanced numerical anlysis	4 3 4 4 4 4 4	visualization languages GRI
SP. Social and Professional Issues (16 core hours) SP1. History of computing (1) SP2. Social context of computing (2) SP3. Methods and tools of analysis (2) SP4. Professional and ethical responsibilities (2) SP5. Risks and liabilities of safety-critical systems (2) SP6. Intellectual property (3) SP7. Privacy and civil liberties (2) SP8. Social implications of the Internet (2)	**CPTR 435** CIS 301 CPTR 495	**Software engineering** Management information systems Colloquium (4 quarters)	**4** 4 0	John L Nesheim *High Tech Startup: The Complete Handbook for Creating Successful New High Tech Companies* Free Press 2000 W. Keith Schilit *The Entrepreneur's Guide to Preparing a Winning Business Plan and Raising Venture Capita*l Pearson 1990 Constance E. Bagley, Craig E. Dauchy *The Entrepreneur's Guide to Business Law* International Thomson Publishing 1997

SP9. Computer crime SP10. Economic issues in computing SP11. Philosophical foundations of ethics			
Capstone experience			
Senior Project	**CPTR 496** **CPTR 497** **CPTR 498**	**Seminar** **Seminar** **Seminar**	**1** **1** **1**
Mathematics - 12 quarter hours			
DS. Discrete Structures (37 core hours) DS1. Functions, relations, and sets (6) DS2. Basic logic (10) DS3. Proof techniques (12) DS4. Basics of counting (5) DS5. Graphs and trees (4)	**MATH 250**	**Discrete mathematics**	**4**
Probability & statistics - basic statistical techniques, focusing primarily on discrete probability with some coverage of mathematical expression and standard statistical measures (normal and Poisson), with an emphasis on the practical application of these techniques to problems that arise in the computing discipline	**MATH 206** MATH 315 MATH 316	**Applied statistics** Probability & statistices Statistics	**4** 4 4
At least one additional course to develop mathematical sophistication, which might be in any of a number of areas including calculus, linear algebra, number theory, or symbolic logic. Coding theory, cryptography	MATH 123 MATH 181 MATH 281 MATH 282 MATH 283 MATH 289	Survey of calculus Analytic geometry & calculus I Analytic geometry & calculus II Analytic geometry & calculus III Analytic geometry & calculus IV Linear algebra & its applications	4 4 4 4 4 3
Science - 12 quarter hours			
For science, there is a need for a genuine exposure to the scientific method. We believe that any science requirement should allow substantial flexibility in terms of subject matter, but should include a lab component to provide actual experience with the scientific method.		Scientific method with lab	12
Other Cognates			

Business basics	MGMT MKTG	Management course Marketing course Other from ACCT, ECON, FINA, or GBUS	12

The Computer Science Major Requirements

- BS degree requirements (192 hours total)
    1. 61 hours of CIS, CPTR, INFO and selected ENGR and MATH courses including
        1. the core hour and distribution requirement
        2. Four quarters of CPTR495 Colloquium
        3. CPTR 496-498 Seminar
    2. The mathematics requirement (12 hours)
    3. The science requirement (12 hours)
    4. MFAT exam
- The BA degree requirements (192 hours total)
    1. 48 hours of CIS, CPTR, INFO and selected ENGR and MATH courses including
        1. the core hour and distribution requirement
        2. Four quarters of CPTR495 Colloquium
        3. CPTR 496-498 Seminar
    2. The mathematics requirement (12 hours)
    3. The science requirement (12 hours)
    4. MFAT exam
- The AS degree requirements (96 hours total - 32 hours general studies)
    1. 53 hours of selected ACCT, CIS, CPTR, FINA, ENGR, GBUS, INFO, MATH, or MGMT courses including
        1. the core hour and distribution requirement
    2. The mathematics requirement (12 hours)
    3. The science requirement (12 hours)
    4. MFAT exam

Questions

- Can we package the CS core in an minor?
- Should we provide options such as:
    - hardware,
    - software,
    - graduate school preparation,
    - e-commerce,
    - CIS,
    - etc?
- What cognates should we require (See SE proposal for comparison)?
    - Math, Science
    - Social science
    - Philosophy
    - Communications

- Business

Notes

- The CS core consists of a minimum of 240 lectures hours representing a minimum of 24 quarter hours of credit (core hours are clock hours of lecture).
- CPTR 494 Cooperative education 0-2; INFO 150 Application software 1; INFO 250 System Software 1

DRAFT - Last update:

BS - Software Engineering

A curriculum proposal based on the
IEEE-CS/ACM Education Task Force
Accreditation Guidelines

STATUS
- Added an internship requirement 5/5/2000
- Circulated for comment to EE, CS, BUS, Tech 4/21/2000
- Elaborated math and science requirements 11/30/2000
- Approved by CS faculty -
- Approved by EE faculty -

Proposed curriculum

Senior students are required to take the MFAT exam in Computer Science.

SE major - BS degree 192 hours		CrHr
Computer Science & Engineering - 37 hours		
CPTR 141(1)	Introduction to Programming	4
CPTR 142, 143	Data Structures and Algorithms	4,4
CPTR 215	Assembly Language Programming	3
CPTR 316	Programming Paradigms	4
CPTR 352	Design and Analysis of Algorithms	4
CPTR 425(2)	Introduction to Networking	4
CPTR 454	Operating System Design	4
ENGR 121-123	Introduction to Engineering	6
Software engineering - 34 hours		

CPTR 235	System Software & Programming	4
CPTR 245	Object-Oriented System Design	4
CPTR 415(3)	Introduction to Databases	4
CPTR 435(4)	Software Engineering	4
	Software engineering electives	10
ENGR 326(5)	Engineering Economy	3
ENGR 345(6)	Contracts and Specifications	2
ENGR 396	Seminar	0
ENGR 496-498	Seminar	3
ENGR 495	Colloquium	0

Applications and Advanced materials - 36 hours

	Math & science electives	8
	Zero or more hours	0-12
	CIS, CPTR, ENGR, INFO electives	
	One or more area (of 12+ hours each)	12-24
	Business (beyond requirement)	
	Engineering (beyond requirement)	
	Graphics	
	Mathematics (beyond requirement)	
	Science (beyond requirement)	

Supporting Areas - 39 hours

ENGL 121-2	College Writing	6
ENGL 323	Writing for Engineers	3
SPCH 101	Fund. of Speech Communications	4
SPCH 207(7)	Small Group Communications	3
MATH 206(8)	Applied Statistics	4
MATH 250	Discrete Mathematics	4
MATH 181(9)	Analytic Geom & Calc I, II	8
MATH 289	Linear Algebra and Applications	3
PHIL 206	Intro to Logic	4

General studies - 50 hours

PSYC 130	H&PE electives	2
	History electives	8
	General Psychology	4
PHYS	Humanities electives	8
	Religion electives	16
	General or Prin of Physics	12
		192

Courses may not be used to satisfy multiple requirements.

Footnotes (1-9) To meet the needs of a wider range of interests and aptitudes, the following substitutions are permitted.

1	CIS 130 CIS 230 CIS 330	Intro to Business App Programming Interm Business App Programming Adv Business App Programming	4 4 4
2	CIS 350 CIS 290	Telecommunications and Intro to Network Administration	4 4
3	CIS 440	Database Management Systems	4
4	CIS 315	Systems Analysis and Design	4
5	GBUS 366 MGMT 371	Operations Management and Production Management & Organizational Behavior	4 4
6	GBUS 361	Business Law	4
7	ENGL 325 GBUS 270 GBUS 370 PSYC 360 MGMT 476 SPCH 310 SPCH 410	Writing for the Professions *or* Business Communications *or* Advanced Business Communications *or* Small Group Procedures *or* Motivation and Leadership *or* Interpersonal & Nonverbal Communications *or* Introduction to General Semantics	3 3 3 3 4 3 2
8	BIOL GBUS MATH 315	Biostatistics *or* Business Statistics *or* Probability & Statistics	4
9	MATH 123	Survey of Calculus	4

Math-science requirement

ABET requires one year of mathematics and science i.e., 48 quarter hours. The proposed implementation is as follows:

Area	Classes	Rationale
Mathematics	Discrete, Applied Statistics	ABET
(23 hours)	Calculus I, II, Linear Algebra, Logic	Curricular support
Science	12 hours of General or Principles of Physics	Traditional bias
(16 hours)	4 hours of General Psychology	To support HCI
Electives (12 hours)	Science electives: Astronomy, Biology, Chemistry, Physics, Psychology	
	Math electives: any college level mathematics course	ABET

New courses

The proposed SE curricula are based on courses currently available. Comparison with ABET's curricula requirements, the recommendations of the SWECC, and other programs suggests that consideration be given to the addition of three to five new courses which would replace some of the suggested courses.

Additional discrete mathematics courses that emphsize skill in proof techniques and in symbolic manipulation such as

- Combinatorics
- Cryptography
- Number theory
- Set theory
- Symbolic logic

would be welcome.

Differences with current programs

1. BS-SE, BS-CIS differences
    - BS-SE requires a maximum of 24 hours of non CIS business courses and permits upto another 12 hours for a total of 36 hours while BS-CIS requires 59 hours. Note that event the BA-BA requires 59 hours of non CIS business courses.
    - BS-SE requires at least 64 hours of computing course work while BS-CIS requires 48 hours.
    - BS-SE degree requires 48 hours of math and science while the BS-CIS requires 20 hours.
    - Informal data collected over nine years suggests that students migrate from the BS-CS (software option) toward CIS and not vice versa.
2. BS-SE, BS-CS differences
    - BS-SE has no free electives. BS-CS has 33 hours of free electives and there is more elective choice in general studies.
3. BS-SE, BSE-CpE differences
    - BS-SE requires 55 hours of general studies hours of general studies while the BSE-CpE requires 44.
    - BSE-CpE requires 29 hours of engineering courses not required in the BS-SE.
    - BS-SE requires 48 hours of math and science while the BSE-CpE requires 55.

ABET and IEEE-CS/ACM Guidelines

Definition

> Software engineering encompasses theory, technology, practice and application of software in computer-based systems. A central theme of the curriculum is to engender an engineering discipline in students, enabling them to define and use processes, models and metrics in software and system development. The curriculum integrates technical requirements with general education requirements

and electives to prepare the student for a professional career in the field, for further study, and for functioning in modern society. The program must include approximately equal segments in software engineering, in computer science and engineering, in appropriate supporting areas, and in advanced materials. This material should cover about three-quarters of the overall academic program, with the remainder to include institutional requirements and electives.
-- from *Accreditation Criteria for Software Engineering -- IEEE Computer Society and ACM Software Engineering Coordinating Committee*

ABET Engineering Criteria 2000

... Students must be prepared for engineering practice through the curriculum culminating in a major design experience based on the knowledge and skills acquired in earlier course work and incorporating engineering standards and realistic constraints that include most of the following considerations: economic; environmental; sustainability; manufacturability; ethical; health and safety; social; and political. The professional component must include

a. one year of a combination of college level mathematics and basic sciences (some with experimental experience) appropriate to the discipline
b. one and one-half years of engineering topics, consisting of engineering sciences and engineering design appropriate to the student's field of study
c. a general education component that complements the technical content of the curriculum and is consistent with the program and institutional objectives.

...

The curriculum must provide both breadth and depth across the range of engineering and computer science topics implied by the title and objectives of the program.

The program must demonstrate that graduates have: the ability to analyze, design, verify, validate, implement, apply, and maintain software systems; the ability to appropriately apply discrete mathematics, probability and statistics, and relevant topics in computer and management sciences to complex software systems.

IEEE-CS/ACM Education Task Force Accreditation Guidelines

Graduates of the program must demonstrate the ability to analyze, design, verify, validate, implement, and maintain software systems, using appropriate quality assurance techniques/methods in all of these. Graduates must understand and use appropriate processes, models and metrics in software development. They must possess the necessary team and communication skills to function in a typical software development environment.

Software engineering encompasses theory, technology, practice and application of software in computer-based systems. A central theme of the curriculum is to engender an engineering discipline in students, enabling them to define and use processes, models and metrics in software and system development.

The program includes approximately equal segments in software engineering, in computer science and engineering, in appropriate supporting areas, and in advanced materials **(~36 hours each)**. This material covers about three-quarters **(144 hours)** of the overall academic program **(192 hours)**.

The program addresses all aspects of software development and maintenance, and provides experience in a realistic team environment. These notions are integrated throughout the curriculum, and are incorporated in a meaningful major project that integrates many aspects of the curriculum.

Computer Science and Engineering: The areas of algorithms and data structures, computer architecture, databases, programming languages, operating systems, and networking, integrated and applied in advanced software engineering courses and projects.

Software Engineering: Covers processes and techniques for developing and maintaining large systems. Courses should address the areas of requirements analysis, software architecture and design, testing and quality assurance, software management, selection and use of software tools and components, computer and human interaction, maintenance and documentation. Substantial design work must be included and the students must be exposed to a variety of languages and systems.

Engineering responsibility and practice must be stressed, which includes conveying ethical, social, legal, economic and safety issues. These concerns must be reinforced in advanced work, as must the appropriate use of software engineering standards. Students should also learn methods for technical and economic decision making, such as project planning and resource management. Additionally, students must achieve an understanding of the need for and an ability to engage in life-long learning.

Advanced Materials: Providing depth in one or more areas. This part of the program may incorporate further study in the software engineering and computer science topics indicated above, may involve work in additional areas of theory or technology, and should include work in one or more significant application domains. Particular domains may require additional work in supporting areas such as mathematics and science.

Supporting Areas: Communications (oral, written, listening), the abilities to work in teams, and *mathematics focusing primarily on discrete mathematics and probability and statistics.*

REFERENCES

- ABET Engineering Criteria 2000
- IEEE-CS/ACM Curriculum 2001
- Software Engineering body of knowledge SWEBOK
- Software Engineering Coordinating Committee (SWECC)

- Software Engineering Code of Ethics and Professional Practice
- Sample BS SwE Programs
  - Auburn University
  - Capitol-College
  - Gannon University
  - Milwaukee School of Engineering
  - Rochester Institute of Technology

Send comments to aabyan@wwc.edu