Compiler Construction using Flex and Bison

Anthony A. Aaby
Walla Walla College
cs.wwc.edu
aabyan@wwc.edu

Version of February 25, 2004

Walla Walla College
204 S. College Ave.
College Place, WA 99324
E-mail: aabyan@wwc.edu
LaTeX sources available by request.

To Pamela Aaby

Contents

Chapter 1

Introduction

A language translator is a program which translates programs written in a 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 to some real or virtual machine. This chapter shows how context-free grammars are used in the construction of language translators. Since the translation is guided by 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 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.

  Lex and Flex are tools for generating scanners: programs which recognize lexical patterns in text. Flex is a faster version of Lex. In this chapter Lex/Flex refers to either of the tools. The appendix on Lex/Flex is a condensation of the manual page “flexdoc” by Vern Paxon.

- 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 parsers: programs which recognize the structure grammatical structure of programs. Bison is a faster version of Yacc. In this chapter, Yacc/Bison refers to either of these tools. The sections on Yacc/Bison are a condensation and extension of the document "BISON the Yacc-compatible Parser Generator" by Charles Donnelly and Richard Stallman.

- The *semantic analysis phase* analyzes 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 parser. 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.

- The *optimizer* applies semantics preserving transformations 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 code generator may be integrated with the parser.

- The *peep-hole* optimizer examines the object code, a few instructions at a time, and attempts to do machine dependent code improvements.

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.

For illustration purposes, we will construct a compiler for a simple imperative programming language called Simple. The context-free grammar for Simple is

```
program ::= LET [ declarations ] IN command_sequence END

declarations ::= INTEGER [ id_seq ] IDENTIFIER .

id_seq ::= id_seq... IDENTIFIER ,

command_sequence ::= command... command

command ::= SKIP ;
          | IDENTIFIER := expression ;
          | IF exp THEN command_sequence ELSE command_sequence FI ;
          | WHILE exp DO command_sequence END ;
          | READ IDENTIFIER ;
          | WRITE expression ;

expression ::= NUMBER | IDENTIFIER | '(' expression ')'
          | expression + expression | expression − expression
          | expression ∗ expression | expression / expression
          | expression ˆ expression
          | expression = expression
          | expression < expression
          | expression > expression
```

Figure 1.1: Simple

given in Figure 1.1 where the non-terminal symbols are given in all lower case, the terminal symbols are given in all caps or as literal symbols and, where the literal symbols conflict with the meta symbols of the EBNF, they are enclosed with single quotes. The start symbol is program. While the grammar uses upper-case to high-light terminal symbols, they are to be implemented in lower case.

There are two context sensitive requirements; variables must be declared before they are referenced and a variable may be declared only once.

Chapter 2

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.

Chapter 3

The Scanner

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. A input file for Lex is of the form:

Lex generates a file containing the function `yylex()` which returns an integer denoting the token recognized.

```
C and scanner declarations
%%
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 3.1 gives the format of some of the regular expressions that may be used to define the tokens. 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.

. 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

Figure 3.1: Lex/Flex Regular Expressions

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 Generator by M. E. Lesk and E. Schmidt.

Chapter 4

The Context

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 and

```
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,

```
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;
}
```

and getsym which returns a pointer to the symbol table entry corresponding to an identifier.

```
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^{t}h 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( $2 ); }
...
%%
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.

Chapter 5

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.

Chapter 6

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 [1] 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.

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
READ	0,N	Input integer in to location N: `S(N) := Input`
WRITE		Output top of stack: `Output := S(T); T:= T-1`
OPR		*Arithmetic and logical operations*
	0,0	process and function, return operation

[1] This is an adaptation of: Niklaus Wirth, *Algorithms + Data Structures = Programs* Prentice-Hall, Englewood Cliffs, N.J., 1976.

		T:= AR-1; AR:= S(T+2); P:= S(T+3)
ADD		ADD: S(T-1) := S(T-1) + S(T); T := T-1
SUB		SUBTRACT: S(T-1) := S(T-1) - S(T); T := T-1
MULT		MULTIPLY: S(T-1) := S(T-1) * S(T); T := T-1
DIV		DIVIDE: S(T-1) := S(T-1) / S(T); T := T-1
MOD		MOD: S(T-1) := S(T-1) mod S(T); T := T-1
EQ		TEST EQUAL: S(T-1) := if S(T-1) = S(T) then 1 else 0; T:= T-1
LT		TEST LESS THAN: S(T-1) := if S(T-1) < S(T) then 1 else 0; T:= T-1
GT		TEST GREATER THAN: S(T-1) := if S(T-1) > S(T) then 1 else 0; T:= T-1
LD_INT	0,N	LOAD literal value onto stack: T:= T+1; S(T):= N
LD_VAR	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
STORE	L,N	store value on top of stack into variable location at level offset L, base offset N in stack S(f(ld,AR)+os+3):= S(T); T:= T - 1

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}
		P:= cos; {Program Counter}
		T:= T+3 {Stack Top}

CAL	255,0	call procedure address in memory: POP address, PUSH return address, JUMP to address
DATA	0,NN	ADD NN to stack pointer T := T+NN
GOTO	0,N	JUMP: P := N
JMP_FALSE	C,N	JUMP: if S(T) = C then P:= N; T:= T-1

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 -> halt
                     & 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 `pc`, the instruction register `ir`, the activation record pointer `ar` (which points to be begining of the current activation record), and the pointer to the top of the stack `top`, are straight forward.

```
int                 pc   = 0;
struct 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++];
          /* 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);
 }
```

Chapter 7

Code Generation

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".

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:

```
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 *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
```

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, Figure 7.1 is a program in Simp and Figure 7.2.

```
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
```

Figure 7.1: A Simple program

```
 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
```

Figure 7.2: Stack code

Chapter 8

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
```

Chapter 9

Further Reading

For information on compiler construction using Lex and Yacc see[?]. Pratt [?] emphasizes virtual machines.

Chapter 10

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.

   (a) Declarations: Change the semantic processing of identifier references to require previous declaration.

   (b) 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.

   (c) Multiplication and division: Make appropriate changes to the semantic routines to handle code generation based on the new tokens.

   (d) **if** and **while** statements: Semantic routines must generate the proper tests and jumps.

   (e) 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:

(a) An interpreter for the code produced by the compiler

(b) 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.

   (a) Scanner extensions to handle the new tokens, use of parser generator to produce new tables(20 points).

   (b) Declarations of integer and real variables(10 points).

   (c) Integer literals, expressions involving integers, I/O for integers, and output for strings(10 points).

   (d) The **loop** and **exit** statements and addition of the **else** and **elsif** parts to the **if** statement(20 points).

   (e) Recursive procedures with parameters(8 points for simple procedures, 8 points for recursion, 12 points for parameters).

   (f) Record declarations and field references(8 points).

   (g) Array declarations and element references(12 points).

   (h) 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)

   (a) (100 points)

   i. Integer, Real, Boolean types (5 points)

   ii. Basic expressions involving Integer, Real and Boolean types $(+, -, *, /,$ **not, and or, abs, mod,** $**$, $<, <=, >, >=, =, / =)$ (30 points).

   iii. Input/Output

   A. Input of Integer, Real, Boolean scalars(5 points).

   B. Output of String literals and Integer, Real and Boolean expressions(excluding formatting)(5 points).

   iv. Block structure (including declaration of local variables and constants) (20 points).

   v. Assignment statement (10 points).

   vi. **if**, **loop**, and **exit** statements (10, 5, 10 points respectively)

(b) (30 points) Procedures and scalar-valued functions of no arguments (including nesting and non-local variables).

Optional Features (336 points possible)

(a) **loop** statements (15 points total)
   i. **in** and **in reverse** forms (10 points)
   ii. **while** form (5 points)

(b) Arrays (30 points total)
   i. One-dimensional, compile-time bounds, including First and Last attributes (10 points)
   ii. Multi-dimensional, compile-time bounds, including First and Last attributes (5-points)
   iii. Elaboration-time bounds (9 points)
   iv. Subscript checking (3 points)
   v. Record base type (3 points)

(c) Boolean short-circuit operators (**and then, or else**) (12 points)

(d) Strings (23 points total)
   i. Basic string operations (string variables, string assigns, all string operators (&, Substr, etc), I/O of strings) (10 points)
   ii. Unbounded-length strings (5 points)
   iii. Full garbage collection of unbounded-length strings (8 points)

(e) Records (15 points total)
   i. Basic features (10 points)
   ii. Fields that are compile-time bounded arrays (2 points)
   iii. Fields that are elaboration-time sized (both arrays and records) (3 points)

(f) Procedures and functions(53 points total)
   i. Scalar parameters (15 points)
   ii. Array arguments and array-valued functions (compiler-time bounds) (7 points)
   iii. Array arguments and array-valued functions (elaboration-time bounds) (5 points)
   iv. Record arguments and record-value functions (4 points)
   v. Conformant array parameters (i.e. array declarations of the form **type array (**T **range** <>**) of** T2) (8 points)
   vi. Array-valued functions (elaboration-sized bounds) (3 points)
   vii. Array-valued functions (conformant bounds) (4 points)
   viii. Forward definition of procedures and functions (3 points)
   ix. String arguments and string-valued functions (4 points)

(g) **case** statement (20 points total)
   i. Jump code (10 points)

ii. If-then-else code (4 points)

iii. Search-table code (6 points)

(h) Constrained **subtypes** (including First and Last attributes) (10 points total)

i. Run-time range checks (7 points)

ii. Compile-time range checks (3 points)

(i) Folding of scalar constant expressions (8 points)

(j) Initialized variables (10 points total).

i. Compile-time values, global (without run-time code) (3 points)

ii. Compile-time values, local (2 points)

iii. Elaboration-time values (2 points)

iv. Record fields (3 points)

(k) Formatted writes (3 points).

(l) Enumerations (18 points total).

i. Declaration of enumeration types; variables, assignment, and comparison operations (9 points)

ii. Input and Output of enumeration values (5 points)

iii. Succ, Pred, Char, and Val attributes (4 points)

(m) Arithmetic type conversion (3 points).

(n) Qualified names (from blocks and subprograms) (3 points).

(o) Pragmata (2 points).

(p) Overloading (25 points total)

i. Subprogram identifier (18 points)

ii. Operators (7 points)

(q) Packages (55 points total).

i. Combined packages (containing both declaration and body parts); qualified access to visible part (20 points)

ii. Split packages (with distinct declaration and body parts) (5 points)

iii. Private types (10 points)

iv. Separate compilation of package bodies (20 points)

(r) Use statements (11 points)

(s) 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.

Appendix A

Simple - The complete implementation

A.1 The parser: Simple.y

```
%{/*************************************************************************

                   Compiler for the Simple language

***************************************************************************/
/*=========================================================================
       C Libraries, Symbol Table, Code Generator & other C 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                                 */
/*-------------------------------------------------------------------------
               The following support backpatching
-------------------------------------------------------------------------*/
 struct  lbs              /* Labels for data, 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 identifier & check if previously defined.
-------------------------------------------------------------------------*/
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 );
   }
}
/*-------------------------------------------------------------------------
                 If identifier is defined, generate code
-------------------------------------------------------------------------*/
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 );
}
/*=========================================================================
                          SEMANTIC RECORDS
=========================================================================*/
%}
%union semrec             /* The Semantic Records                        */
{
 int     intval;          /* Integer values                              */
 char    *id;             /* Identifiers                                 */
 struct lbs *lbls;        /* For backpatching                            */
}
/*=========================================================================
                               TOKENS
=========================================================================*/
%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
/*=========================================================================
                          OPERATOR PRECEDENCE
=========================================================================*/
%left ’-’ ’+’
%left ’*’ ’/’
%right ’^’
/*=========================================================================
                   GRAMMAR RULES for the Simple language
=========================================================================*/
%%
program : LET
             declarations
          IN           { gen_code( DATA, data_location() - 1 );          }
             commands
          END          { gen_code( HALT, 0 ); YYACCEPT;                  }
;
declarations : /* empty */
   | INTEGER id_seq IDENTIFIER ’.’ { install( $3 );                      }
;
id_seq : /* empty */
   | id_seq IDENTIFIER ’,’  { install( $2 );                             }
;
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() );                       }
;
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 ’)’
;
%%
/*=========================================================================
                                  MAIN
=========================================================================*/
main( int argc, char *argv[] )
{ extern FILE *yyin;
  ++argv; --argc;
  yyin = fopen( argv[0], "r" );
  /*yydebug = 1;*/
  errors = 0;
  yyparse ();
  printf ( "Parse Completed\n" );
  if ( errors == 0 )
  { print_code ();
    fetch_execute_cycle();
  }
}
/*=========================================================================
                                 YYERROR
=========================================================================*/
yyerror ( char *s )  /* Called by yyparse on error */
{
  errors++;
  printf ("%s\n", s);
}
/**************************** End Grammar File ***************************/
```

A.2 Directions

```
Directions:  this file contains a sample terminal session.


> bison -d Simple.y
   or
> bison -dv Simple.y
Simple.y contains 39 shift/reduce conflicts.
> gcc -c Simple.tab.c
> flex Simple.lex
> gcc -c lex.yy.c
> gcc -o Simple Simple.tab.o lex.yy.o -lm
> Simple test_simple
Parse Completed
  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
Input: 6
Output: 10
Output: 625
```

A.3 The scanner: Simple.lex

```
/***************************************************************************

                   Scanner for the Simple language

***************************************************************************/
%{
/*=========================================================================
                      C-libraries and Token  definitions
=========================================================================*/
#include <string.h>     /* for strdup                       */
/*#include <stdlib.h> */    /* for atoi                         */
#include "Simple.tab.h" /* for token definitions and yylval */
%}
/*=========================================================================
                             TOKEN Definitions
=========================================================================*/
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
/*=========================================================================
      REGULAR EXPRESSIONS defining the tokens for the Simple language
=========================================================================*/
%%
":="     { return(ASSGNOP);  }
{DIGIT}+ { yylval.intval = atoi( yytext );
           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]);}
%%
```

```
int yywrap(void){}
/************************** End Scanner File *****************************/
```

A.4 The symbol table: ST.h

```
/***************************************************************************
                             Symbol Table Module
***************************************************************************/
/*=========================================================================
                             DECLARATIONS
=========================================================================*/
/*-------------------------------------------------------------------------
                         SYMBOL TABLE RECORD
-------------------------------------------------------------------------*/
struct symrec
{
  char *name;  /* name of symbol                     */
  int offset;  /* data offset                        */
  struct symrec *next;    /* link field              */
};
typedef struct symrec symrec;
/*-------------------------------------------------------------------------
                         SYMBOL TABLE ENTRY
-------------------------------------------------------------------------*/
symrec *identifier;
/*-------------------------------------------------------------------------
                              SYMBOL TABLE
                  Implementation: a chain of records.
------------------------------------------------------------------------*/
symrec *sym_table = (symrec *)0; /* The pointer to the Symbol Table */
/*========================================================================
                       Operations: Putsym, Getsym
========================================================================*/
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;
}
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;
}
/************************** End Symbol Table **************************/
```

A.5 The code generator: CG.h

```
/***************************************************************************
                             Code Generator
***************************************************************************/
/*-------------------------------------------------------------------------
                             Data Segment
-------------------------------------------------------------------------*/
int data_offset = 0;          /* Initial offset                            */
int data_location()           /* Reserves a data location                  */
{
   return data_offset++;
}
/*-------------------------------------------------------------------------
                             Code Segment
-------------------------------------------------------------------------*/
int code_offset = 0;          /* Initial offset                            */

int gen_label()               /* Returns current offset                    */
{
   return code_offset;
}

int reserve_loc()             /* Reserves a code location                  */
{
   return code_offset++;
}
                              /* Generates code at current location      */
void gen_code( enum code_ops operation, int arg )
```

```
{ code[code_offset].op    = operation;
  code[code_offset++].arg = arg;
}
                              /* Generates code at a reserved location   */
void back_patch( int addr,  enum code_ops operation, int arg  )
{
  code[addr].op  = operation;
  code[addr].arg = arg;
}
/*-------------------------------------------------------------------------
                           Print Code to stdio
-------------------------------------------------------------------------*/
void print_code()
{
   int i = 0;

   while (i < code_offset) {
      printf("%3ld: %-10s%4ld\n",i,op_name[(int) code[i].op], code[i].arg );
      i++;
   }
}
/************************** End Code Generator **************************/
```

A.6 The stack machine: SM.h

```
/***************************************************************************
                             Stack Machine
***************************************************************************/
/*=========================================================================
                              DECLARATIONS
=========================================================================*/

/* 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;
 };

/* CODE Array */

struct instruction code[999];

/* RUN-TIME Stack */

int stack[999];

/*-------------------------------------------------------------------------
                              Registers
-------------------------------------------------------------------------*/
int                 pc   = 0;
struct instruction  ir;
int                 ar   = 0;
int                 top  = 0;
char         ch;
/*=========================================================================
                             Fetch Execute Cycle
=========================================================================*/
void fetch_execute_cycle()
 {   do { /*printf( "PC = %3d IR.arg = %8d AR = %3d Top = %3d,%8d\n",
             pc, ir.arg, ar, top, stack[top]); */
          /* Fetch         */
          ir = code[pc++];
          /* 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;
                                                                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);
 }
/*************************** End Stack Machine **************************/
```

A.7 Sample program: test_simple

```
let
   integer n,x.
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
```

Appendix B

Lex/Flex

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.

B.1 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.

B.2 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

B.2.1 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.

B.2.2 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 may be skipped, too.

In the definitions and rules sections, any 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? 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 ‘*’)

\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”.

ˆ 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”.

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 “[ˆ 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., “[ˆ 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 [ˆ "]* 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`.

B.2.3 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.

B.3 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.

B.4 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;
```

Appendix C

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).

C.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.

C.2 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);
}
```

C.3 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.

C.3.1 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 <val> 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:

%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.

C.3.2 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.

C.3.3 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.

C.3.4 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.

C.4 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.

C.5 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:

```
YYACCEPT Return immediately with value 0 (to report success).

YYABORT Return immediately with value 1 (to report failure).
```

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.

C.6 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.

C.7 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.