A Compact Guide to Lex & Yacc

by Tom Niemann

epaperpress.com

Contents

Preface

This document explains how to construct a compiler using lex and yacc. Lex and yacc are tools used to generate lexical analyzers and parsers. I assume you can program in C, and understand data structures such as linked-lists and trees.

The introduction describes the basic building blocks of a compiler and explains the interaction between lex and yacc. The next two sections describe lex and yacc in more detail. With this background, we construct a sophisticated calculator. Conventional arithmetic operations and control statements, such as **if-else** and **while**, are implemented. With minor changes, we convert the calculator into a compiler for a stack-based machine. The remaining sections discuss issues that commonly arise in compiler writing. Source code for examples may be downloaded from the web site listed below.

Tom Niemann
Portland, Oregon
web site: epaperpress.com

Introduction

Until 1975, writing a compiler was a very time-consuming process. Then Lesk [1975] and Johnson [1975] published papers on lex and yacc. These utilities greatly simplify compiler writing. Implementation details for lex and yacc may be found in Aho [1986]. Lex and yacc are available from

- Mortice Kern Systems (MKS), at www.mks.com,
- GNU flex and bison, at www.gnu.org,
- Ming, at www.mingw.org,
- Cygwin, at www.cygwin.com

The version from MKS is a high-quality commercial product that retails for about $300US. GNU software is free. Output from flex may be used in a commercial product, and, as of version 1.24, the same is true for bison. Cygwin is a 32-bit Windows ports of the GNU software. In fact Cygwin is a port of the Unix operating system to Windows, complete with compilers gcc and g++. To install download and run the setup executable. Under devel install bison, flex, gcc-g++, and make. Under editors install vim. Lately I've been using flex and bison under the cygwin environment.

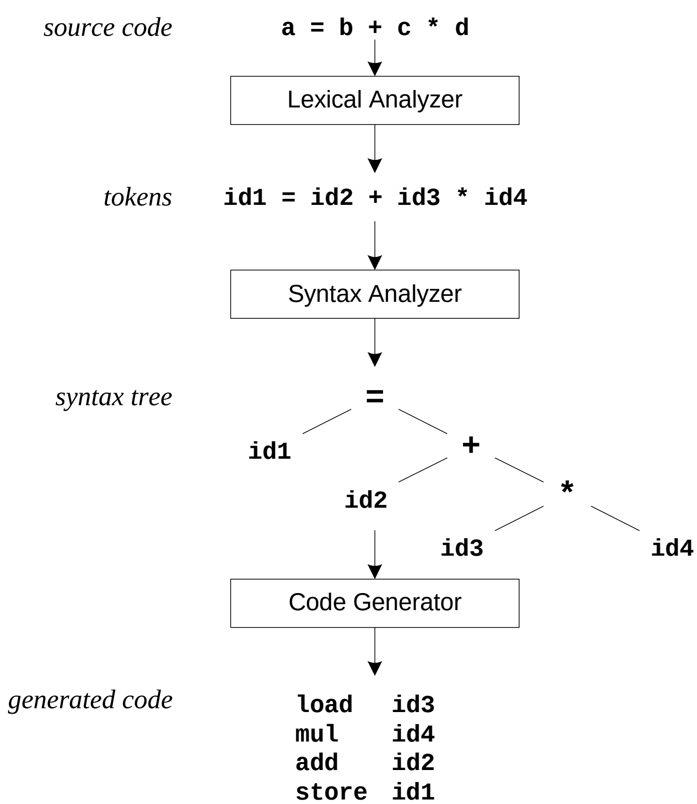

Figure 1: Compilation Sequence

Lex generates C code for a lexical analyzer, or scanner. It uses patterns that match strings in the input and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing. This is illustrated in Figure 1.

As lex finds identifiers in the input stream, it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of the variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

Yacc generates C code for a syntax analyzer, or parser. Yacc uses grammar rules that allow it to analyze tokens from lex and create a syntax tree. A syntax tree imposes a hierarchical structure on tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly.

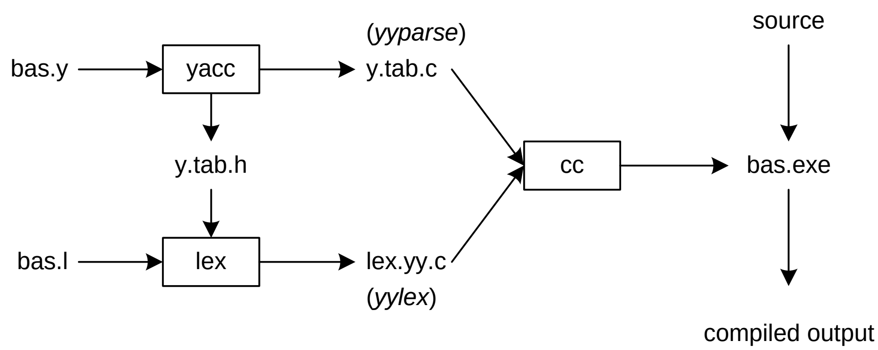

Figure 2: Building a Compiler with Lex/Yacc

Figure 2 illustrates the file naming conventions used by lex and yacc. We'll assume our goal is to write a BASIC compiler. First, we need to specify all pattern matching rules for lex (**bas.l**) and grammar rules for yacc (**bas.y**). Commands to create our compiler, **bas.exe**, are listed below:

```
yacc –d bas.y                    # create y.tab.h, y.tab.c
lex bas.l                        # create lex.yy.c
cc lex.yy.c y.tab.c –obas.exe    # compile/link
```

Yacc reads the grammar descriptions in **bas.y** and generates a parser, function **yyparse**, in file **y.tab.c**. Included in file **bas.y** are token declarations. The **–d** option causes yacc to generate definitions for tokens and place them in file **y.tab.h**. Lex reads the pattern descriptions in **bas.l**, includes file **y.tab.h**, and generates a lexical analyzer, function **yylex**, in file **lex.yy.c**.

Finally, the lexer and parser are compiled and linked together to form the executable, **bas.exe**. From **main**, we call **yyparse** to run the compiler. Function **yyparse** automatically calls **yylex** to obtain each token.

Lex

Theory

The first phase in a compiler reads the input source and converts strings in the source to tokens. Using regular expressions, we can specify patterns to lex that allow it to scan and match strings in the input. Each pattern in lex has an associated action. Typically an action returns a token, representing the matched string, for subsequent use by the parser. To begin with, however, we will simply print the matched string rather than return a token value. We may scan for identifiers using the regular expression

```
letter(letter|digit)*
```

This pattern matches a string of characters that begins with a single letter, and is followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "*****" operator
- alternation, expressed by the "**|**" operator
- concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state, and one or more final or accepting states.

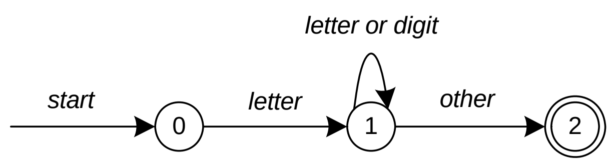

Figure 3: Finite State Automaton

In Figure 3, state 0 is the start state, and state 2 is the accepting state. As characters are read, we make a transition from one state to another. When the first letter is read, we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit, we transition to state 2, the accepting state. *Any* FSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:

```
start:  goto state0

state0: read c
        if c = letter goto state1
        goto state0

state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2

state2: accept string
```

This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next *input* character, and *current state*, the next state is easily determined by indexing into a computer-generated state table.

Now we can easily understand some of lex's limitations. For example, lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a "**(**", we push it on the stack. When a "**)**" is encountered, we match it with the top of the stack, and pop the stack. Lex, however, only has states and transitions between states. Since it has no stack, it is not well suited for parsing nested structures. Yacc augments an FSA with a stack, and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Yacc is appropriate for more challenging tasks.

Practice

<table>
<tr><th>Metacharacter</th><th>Matches</th></tr>
<tr><td>.</td><td>any character except newline</td></tr>
<tr><td>\n</td><td>newline</td></tr>
<tr><td>*</td><td>zero or more copies of the preceding expression</td></tr>
<tr><td>+</td><td>one or more copies of the preceding expression</td></tr>
<tr><td>?</td><td>zero or one copy of the preceding expression</td></tr>
<tr><td>^</td><td>beginning of line</td></tr>
<tr><td>$</td><td>end of line</td></tr>
<tr><td>a|b</td><td>a or b</td></tr>
<tr><td>(ab)+</td><td>one or more copies of ab (grouping)</td></tr>
<tr><td>"a+b"</td><td>literal "a+b" (C escapes still work)</td></tr>
<tr><td>[]</td><td>character class</td></tr>
</table>

Table 1: Pattern Matching Primitives

<table>
<tr><th>Expression</th><th>Matches</th></tr>
<tr><td>abc</td><td>abc</td></tr>
<tr><td>abc*</td><td>ab abc abcc abccc ...</td></tr>
<tr><td>abc+</td><td>abc abcc abccc ...</td></tr>
<tr><td>a(bc)+</td><td>abc abcbc abcbcbc ...</td></tr>
<tr><td>a(bc)?</td><td>a abc</td></tr>
<tr><td>[abc]</td><td>one of: a, b, c</td></tr>
<tr><td>[a-z]</td><td>any letter, a-z</td></tr>
<tr><td>[a\-z]</td><td>one of: a, -, z</td></tr>
<tr><td>[-az]</td><td>one of: -, a, z</td></tr>
<tr><td>[A-Za-z0-9]+</td><td>one or more alphanumeric characters</td></tr>
<tr><td>[\t\n]+</td><td>whitespace</td></tr>
<tr><td>[^ab]</td><td>anything except: a, b</td></tr>
<tr><td>[a^b]</td><td>one of: a, ^, b</td></tr>
<tr><td>[a|b]</td><td>one of: a, |, b</td></tr>
<tr><td>a|b</td><td>one of: a, b</td></tr>
</table>

Table 2: Pattern Matching Examples

Regular expressions in lex are composed of metacharacters (Table 1). Pattern matching examples are shown in Table 2. Within a character class, normal operators lose their meaning. Two operators allowed in a character class are the hyphen ("**-**") and circumflex ("**^**"). When used between two characters, the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Input to Lex is divided into three sections, with **%%** dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

```
%%
```

Input is copied to output, one character at a time. The first **%%** is always required, as there must always be a rules section. However, if we don't specify any rules, then the default action is to match everything and copy it to output. Defaults for input and output are **stdin** and **stdout**, respectively. Here is the same example, with defaults explicitly coded:

```
%%
    /* match everything except newline */
.   ECHO;
    /* match newline */
\n  ECHO;

%%

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline), and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, "**.**" and "**\n**", with an **ECHO** action associated for each pattern. Several macros and variables are predefined by lex. **ECHO** is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, **ECHO** is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable **yytext** is a pointer to the matched string (NULL-terminated), and **yyleng** is the length of the matched string. Variable **yyout** is the output file, and defaults to stdout. Function **yywrap** is called by lex when input is exhausted. Return 1 if you are done, or 0 if more processing is required. Every C program requires a **main** function. In this case, we simply call **yylex**, the main entry-point for lex. Some implementations of lex include copies of **main** and **yywrap** in a library, eliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly.

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

Table 3: Lex Predefined Variables

Here's a program that does nothing at all. All input is matched, but no action is associated with any pattern, so there will be no output.

```
%%
.
\n
```

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate **yylineno**. The input file for lex is **yyin**, and defaults to **stdin**.

```
%{
    int yylineno;
%}
%%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

The definitions section is composed of substitutions, code, and start states. Code in the definitions section is simply copied as-is to the top of the generated C file, and must be bracketed with "**%{**" and "**%}**" markers. Substitutions simplify pattern-matching rules. For example, we may define digits and letters:

```
digit    [0-9]
letter   [A-Za-z]
%{
    int count;
%}
%%
    /* match identifier */
{letter}({letter}|{digit})*        count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

Whitespace must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces (**{letter}**) to distinguish them from literals. When we have a match in the rules section, the associated C code is executed. Here is a scanner that counts the number of characters, words, and lines in a file (similar to Unix wc):

```
%{
    int nchar, nword, nline;
%}
%%
\n          { nline++; nchar++; }
[^ \t\n]+   { nword++, nchar += yyleng; }
.           { nchar++; }
%%
int main(void) {
    yylex();
    printf("%d\t%d\t%d\n", nchar, nword, nline);
    return 0;
}
```

Yacc

Theory

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique was pioneered by John Backus and Peter Naur, and used to describe ALGOL60. A BNF grammar can be used to express *context-free* languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

```
E -> E + E
E -> E * E
E -> id
```

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as **E** (expression) are nonterminals. Terms such as **id** (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

```
E -> E * E          (r2)
  -> E * z          (r3)
  -> E + E * z      (r1)
  -> E + y * z      (r3)
  -> x + y * z      (r3)
```

At each step we expanded a term, replacing the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression, we actually need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar, we need to reduce an expression to a single nonterminal. This is known as *bottom-up* or *shift-reduce* parsing, and uses a stack for storing terms. Here is the same derivation, but in reverse order:

```
 1    . x + y * z    shift
 2    x . + y * z    reduce(r3)
 3    E . + y * z    shift
 4    E + . y * z    shift
 5    E + y . * z    reduce(r3)
 6    E + E . * z    shift
 7    E + E * . z    shift
 8    E + E * z .    reduce(r3)
 9    E + E * E .    reduce(r2)       emit multiply
10    E + E .        reduce(r1)       emit add
11    E .            accept
```

Terms to the left of the dot are on the stack, while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production, we replace the matched tokens on the stack with the lhs of the production. Conceptually, the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a *handle*, and we are *reducing* the handle to the lhs of the production. This process continues until we have shifted all input to the stack, and only the starting nonterminal remains on the stack. In step 1 we shift the **x** to the stack. Step 2 applies rule r3 to the stack, changing **x** to **E**. We continue shifting and reducing, until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction. Similarly, the add instruction is emitted in step 10. Thus, multiply has a higher precedence than addition.

Consider, however, the shift at step 6. Instead of shifting, we could have reduced, applying rule r1. This would result in addition having a higher precedence than multiplication. This is known as a *shift-reduce* conflict. Our grammar is *ambiguous*, as there is more than one possible derivation that will yield the expression. In this case, operator precedence is affected. As another example, associativity in the rule

```
E -> E + E
```

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar, or supply yacc with directives that indicate which operator has precedence. The latter method is simpler, and will be demonstrated in the practice section.

The following grammar has a *reduce-reduce* conflict. With an **id** on the stack, we may reduce to **T**, or reduce to **E**.

```
E -> T
E -> id
T -> id
```

Yacc takes a default action when there is a conflict. For shift-reduce conflicts, yacc will shift. For reduce-reduce conflicts, it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

Practice, Part I

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Input to yacc is divided into three sections. The definitions section consists of token declarations, and C code bracketed by "**%{**" and "**%}**". The BNF grammar is placed in the rules section, and user subroutines are added in the subroutines section.

This is best illustrated by constructing a small calculator that can add and subtract numbers. We'll begin by examining the linkage between lex and yacc. Here is the definitions section for the yacc input file:

```
%token INTEGER
```

This definition declares an **INTEGER** token. When we run yacc, it generates a parser in file **y.tab.c**, and also creates an include file, **y.tab.h**:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Lex includes this file and utilizes the definitions for token values. To obtain tokens, yacc calls **yylex**. Function **yylex** has a return type of int, and returns the token. Values associated with the token are returned by lex in variable **yylval**. For example,

```
[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }
```

would store the value of the integer in **yylval**, and return token **INTEGER** to yacc. The type of **yylval** is determined by **YYSTYPE**. Since the default type is integer, this works well in this case. Token values 0-255 are reserved for character values. For example, if you had a rule such as

```
[-+]         return *yytext;        /* return operator */
```

the character value for minus or plus is returned. Note that we placed the minus sign first so that it wouldn't be mistaken for a range designator. Generated token values typically start around 258, as lex reserves several values for end-of-file and error processing. Here is the complete lex input specification for our calculator:

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}

%%

[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }

[-+\n]      return *yytext;

[ \t]       ; /* skip whitespace */

.           yyerror("invalid character");

%%

int yywrap(void) {
    return 1;
}
```

Internally, yacc maintains two stacks in memory; a parse stack and a value stack. The parse stack contains terminals and nonterminals, and represents the current parsing state. The value stack is an array of **YYSTYPE** elements, and associates a value with each element in the parse stack. For example, when lex returns an **INTEGER** token, yacc shifts this token to the parse stack. At the same time, the corresponding **yylval** is shifted to the value stack. The parse and value stacks are always synchronized, so finding a value related to a token on the stack is easily accomplished. Here is the yacc input specification for our calculator:

```
%{
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

program:
        program expr '\n'         { printf("%d\n", $2); }
        |
        ;

expr:
        INTEGER                   { $$ = $1; }
        | expr '+' expr           { $$ = $1 + $3; }
        | expr '-' expr           { $$ = $1 - $3; }
        ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}

int main(void) {
    yyparse();
    return 0;
}
```

The rules section resembles the BNF grammar discussed earlier. The left-hand side of a production, or nonterminal, is entered left-justified, followed by a colon. This is followed by the right-hand side of the production. Actions associated with a rule are entered in braces.

By utilizing left-recursion, we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected, we print the value of the expression. When we apply the rule

```
expr: expr '+' expr           { $$ = $1 + $3; }
```

we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case, we pop "**expr '+' expr**" and push "**expr**". We have reduced the stack by popping three terms off the stack, and pushing back one term. We may reference positions in the value stack in our C code by specifying "**$1**" for the first term on the right-hand side of the production, "**$2**" for the second, and so on. "**$$**" designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum. Thus, the parse and value stacks remain synchronized.

Numeric values are initially entered on the stack when we reduce from **INTEGER** to **expr**. After **INTEGER** is shifted to the stack, we apply the rule

```
expr: INTEGER          { $$ = $1; }
```

The **INTEGER** token is popped off the parse stack, followed by a push of **expr**. For the value stack, we pop the integer value off the stack, and then push it back on again. In other words, we do nothing. In fact, this is the default action, and need not be specified. Finally, when a newline is encountered, the value associated with **expr** is printed.

In the event of syntax errors, yacc calls the user-supplied function **yyerror**. If you need to modify the interface to **yyerror**, you can alter the canned file that yacc includes to fit your needs. The last function in our yacc specification is **main** … in case you were wondering where it was. This example still has an ambiguous grammar. Yacc will issue shift-reduce warnings, but will still process the grammar using shift as the default operation.

Practice, Part II

In this section we will extend the calculator from the previous section to incorporate some new functionality. New features include arithmetic operators multiply, and divide. Parentheses may be used to over-ride operator precedence, and single-character variables may be specified in assignment statements. The following illustrates sample input and calculator output:

```
user:  3 * (4 + 5)
calc:  27
user:  x = 3 * (4 + 5)
user:  y = 5
user:  x
calc:  27
user:  y
calc:  5
user:  x + 2*y
calc:  37
```

The lexical analyzer returns **VARIABLE** and **INTEGER** tokens. For variables, **yylval** specifies an index to **sym**, our symbol table. For this program, **sym** merely holds the value of the associated variable. When **INTEGER** tokens are returned, **yylval** contains the number scanned. Here is the input specification for lex:

```
%{
    #include <stdlib.h>
    void yyerror(char *);
    #include "y.tab.h"
%}

%%

    /* variables */
[a-z]       {
                yylval = *yytext - 'a';
                return VARIABLE;
            }

    /* integers */
[0-9]+      {
                yylval = atoi(yytext);
```

```
                return INTEGER;
            }

    /* operators */
[-+()=/*\n] { return *yytext; }

    /* skip whitespace */
[ \t]       ;

    /* anything else is an error */
.               yyerror("invalid character");

%%

int yywrap(void) {
    return 1;
}
```

The input specification for yacc follows. The tokens for **INTEGER** and **VARIABLE** are utilized by yacc to create **#defines** in **y.tab.h** for use in lex. This is followed by definitions for the arithmetic operators. We may specify **%left**, for left-associative, or **%right**, for right associative. The last definition listed has the highest precedence. Thus, multiplication and division have higher precedence than addition and subtraction. All four operators are left-associative. Using this simple technique, we are able to disambiguate our grammar.

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'

%{
    void yyerror(char *);
    int yylex(void);
    int sym[26];
%}

%%

program:
        program statement '\n'
        |
        ;

statement:
        expr                      { printf("%d\n", $1); }
        | VARIABLE '=' expr       { sym[$1] = $3; }
        ;

expr:
        INTEGER
        | VARIABLE                { $$ = sym[$1]; }
        | expr '+' expr           { $$ = $1 + $3; }
        | expr '-' expr           { $$ = $1 - $3; }
        | expr '*' expr           { $$ = $1 * $3; }
        | expr '/' expr           { $$ = $1 / $3; }
        | '(' expr ')'            { $$ = $2; }
        ;
```

```
%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}

int main(void) {
    yyparse();
    return 0;
}
```

Calculator

Description

This version of the calculator is substantially more complex than previous versions. Major changes include control constructs such as **if-else** and **while**. In addition, a syntax tree is constructed during parsing. After parsing, we walk the syntax tree to produce output. Two versions of the tree walk routine are supplied:

- an interpreter that executes statements during the tree walk, and
- a compiler that generates code for a hypothetical stack-based machine.

To make things more concrete, here is a sample program,

```
x = 0;
while (x < 3) {
    print x;
    x = x + 1;
}
```

with output for the interpretive version,

```
0
1
2
```

and output for the compiler version, and

```
    push    0
    pop     x
L000:
    push    x
    push    3
    compLT
    jz      L001
    push    x
    print
    push    x
    push    1
    add
    pop     x
    jmp     L000
L001:
```

a version that generates a syntax tree.

```
Graph 0:

    [=]
     |
   |----|
   |    |
 id(X) c(0)

Graph 1:

               while
                 |
     |----------------|
     |                |
    [<]              [;]
     |                |
   |----|     |----------|
   |    |     |          |
 id(X) c(3) print       [=]
              |          |
              |     |-------|
              |     |       |
            id(X) id(X)    [+]
                            |
                          |----|
                          |    |
                        id(X) c(1)
```

The include file contains declarations for the syntax tree and symbol table. The symbol table, **sym**, allows for single-character variable names. A node in the syntax tree may hold a constant (**conNodeType**), an identifier (**idNodeType**), or an internal node with an operator (**oprNodeType**). A union encapsulates all three variants, and **nodeType.type** is used to determine which structure we have.

The lex input file contains patterns for **VARIABLE** and **INTEGER** tokens. In addition, tokens are defined for 2-character operators such as **EQ** and **NE**. Single-character operators are simply returned as themselves.

The yacc input file defines **YYSTYPE**, the type of **yylval**, as

```
%union {
    int iValue;                 /* integer value */
    char sIndex;                /* symbol table index */
    nodeType *nPtr;             /* node pointer */
};
```

This causes the following to be generated in y.tab.h:

```
typedef union {
    int iValue;                 /* integer value */
    char sIndex;                /* symbol table index */
    nodeType *nPtr;             /* node pointer */
} YYSTYPE;
extern YYSTYPE yylval;
```

Constants, variables, and nodes can be represented by **yylval** in the parser's value stack. Notice the type definitions

```
%token <iValue> INTEGER
%type <nPtr> expr
```

This binds **expr** to **nPtr**, and **INTEGER** to **iValue** in the **YYSTYPE** union. This is required so that yacc can generate the correct code. For example, the rule

```
expr: INTEGER { $$ = con($1); }
```

should generate the following code. Note that **yyvsp[0]** addresses the top of the value stack, or the value associated with **INTEGER**.

```
yylval.nPtr = con(yyvsp[0].iValue);
```

The unary minus operator is given higher priority than binary operators as follows:

```
%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
```

The **%nonassoc** indicates no associativity is implied. It is frequently used in conjunction with **%prec** to specify precedence of a rule. Thus, we have

```
expr: '-' expr %prec UMINUS { $$ = node(UMINUS, 1, $2); }
```

indicating that the precedence of the rule is the same as the precedence of token **UMINUS**. And, as defined above, **UMINUS** has higher precedence than the other operators. A similar technique is used to remove ambiguity associated with the if-else statement (see If-Else Ambiguity).

The syntax tree is constructed bottom-up, allocating the leaf nodes when variables and integers are reduced. When operators are encountered, a node is allocated and pointers to previously allocated nodes are entered as operands.

After the tree is built, function **ex** is called to do a depth-first walk of the syntax tree. A depth-first walk visits nodes in the order that they were originally allocated. This results in operators being applied in the order that they were encountered during parsing. Three versions of **ex** are included: an interpretive version, a compiler version, and a version that generates a syntax tree.

Include File

```
typedef enum { typeCon, typeId, typeOpr } nodeEnum;

/* constants */
typedef struct {
    int value;                  /* value of constant */
} conNodeType;

/* identifiers */
typedef struct {
    int i;                      /* subscript to sym array */
} idNodeType;

/* operators */
typedef struct {
    int oper;                   /* operator */
    int nops;                   /* number of operands */
    struct nodeTypeTag *op[1];  /* operands (expandable) */
} oprNodeType;

typedef struct nodeTypeTag {
    nodeEnum type;              /* type of node */

    /* union must be last entry in nodeType */
    /* because operNodeType may dynamically increase */
    union {
        conNodeType con;        /* constants */
        idNodeType id;          /* identifiers */
        oprNodeType opr;        /* operators */
    };
} nodeType;

extern int sym[26];
```

Lex Input

```
%{
#include <stdlib.h>
#include "calc3.h"
#include "y.tab.h"
void yyerror(char *);
%}

%%

[a-z]       {
                yylval.sIndex = *yytext - 'a';
                return VARIABLE;
            }

[0-9]+      {
                yylval.iValue = atoi(yytext);
                return INTEGER;
            }

[-()<>=+*/;{}.] {
                return *yytext;
             }

">="            return GE;
"<="            return LE;
"=="            return EQ;
"!="            return NE;
"while"         return WHILE;
"if"            return IF;
"else"          return ELSE;
"print"         return PRINT;

[ \t\n]+        ;       /* ignore whitespace */

.               yyerror("Unknown character");
%%
int yywrap(void) {
    return 1;
}
```

Yacc Input

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "calc3.h"

/* prototypes */
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i);
nodeType *con(int value);
void freeNode(nodeType *p);
int ex(nodeType *p);
int yylex(void);

void yyerror(char *s);
int sym[26];                    /* symbol table */
%}

%union {
    int iValue;                 /* integer value */
    char sIndex;                /* symbol table index */
    nodeType *nPtr;             /* node pointer */
};

%token <iValue> INTEGER
%token <sIndex> VARIABLE
%token WHILE IF PRINT
%nonassoc IFX
%nonassoc ELSE

%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%type <nPtr> stmt expr stmt_list


%%

program:
  function                { exit(0); }
  ;

function:
    function stmt         { ex($2); freeNode($2); }
  | /* NULL */
  ;

stmt:
    ';'                     { $$ = opr(';', 2, NULL, NULL); }
  | expr ';'                { $$ = $1; }
  | PRINT expr ';'          { $$ = opr(PRINT, 1, $2); }
```

```
    | VARIABLE '=' expr ';'   { $$ = opr('=', 2, id($1), $3); }
    | WHILE '(' expr ')' stmt { $$ = opr(WHILE, 2, $3, $5); }
    | IF '(' expr ')' stmt %prec IFX { $$ = opr(IF, 2, $3, $5); }
    | IF '(' expr ')' stmt ELSE stmt
                            { $$ = opr(IF, 3, $3, $5, $7); }
    | '{' stmt_list '}'       { $$ = $2; }
      ;

stmt_list:
      stmt                  { $$ = $1; }
    | stmt_list stmt        { $$ = opr(';', 2, $1, $2); }
    ;

expr:
      INTEGER               { $$ = con($1); }
    | VARIABLE              { $$ = id($1); }
    | '-' expr %prec UMINUS { $$ = opr(UMINUS, 1, $2); }
    | expr '+' expr         { $$ = opr('+', 2, $1, $3); }
    | expr '-' expr         { $$ = opr('-', 2, $1, $3); }
    | expr '*' expr         { $$ = opr('*', 2, $1, $3); }
    | expr '/' expr         { $$ = opr('/', 2, $1, $3); }
    | expr '<' expr         { $$ = opr('<', 2, $1, $3); }
    | expr '>' expr         { $$ = opr('>', 2, $1, $3); }
    | expr GE expr          { $$ = opr(GE, 2, $1, $3); }
    | expr LE expr          { $$ = opr(LE, 2, $1, $3); }
    | expr NE expr          { $$ = opr(NE, 2, $1, $3); }
    | expr EQ expr          { $$ = opr(EQ, 2, $1, $3); }
    | '(' expr ')'          { $$ = $2; }
    ;


%%

#define SIZEOF_NODETYPE ((char *)&p->con - (char *)p)

nodeType *con(int value) {
    nodeType *p;
    size_t nodeSize;

    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(conNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeCon;
    p->con.value = value;

    return p;
}

nodeType *id(int i) {
    nodeType *p;
    size_t nodeSize;

    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(idNodeType);
```

```
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeId;
    p->id.i = i;

    return p;
}

nodeType *opr(int oper, int nops, ...) {
    va_list ap;
    nodeType *p;
    size_t nodeSize;
    int i;

    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(oprNodeType) +
        (nops - 1) * sizeof(nodeType*);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeOpr;
    p->opr.oper = oper;
    p->opr.nops = nops;
    va_start(ap, nops);
    for (i = 0; i < nops; i++)
        p->opr.op[i] = va_arg(ap, nodeType*);
    va_end(ap);
    return p;
}

void freeNode(nodeType *p) {
    int i;

    if (!p) return;
    if (p->type == typeOpr) {
        for (i = 0; i < p->opr.nops; i++)
            freeNode(p->opr.op[i]);
    }
    free (p);
}

void yyerror(char *s) {
    fprintf(stdout, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}
```

Interpreter

```
#include <stdio.h>
#include "calc3.h"
#include "y.tab.h"

int ex(nodeType *p) {
    if (!p) return 0;
    switch(p->type) {
    case typeCon:       return p->con.value;
    case typeId:        return sym[p->id.i];
    case typeOpr:
    switch(p->opr.oper) {
      case WHILE:   while(ex(p->opr.op[0]))
                        ex(p->opr.op[1]); return 0;
      case IF:      if (ex(p->opr.op[0]))
                        ex(p->opr.op[1]);
                    else if (p->opr.nops > 2)
                        ex(p->opr.op[2]);
                    return 0;
      case     PRINT:              printf("%d\n",     ex(p->opr.op[0]));
                    return 0;
      case ';':     ex(p->opr.op[0]);
                    return ex(p->opr.op[1]);
      case '=':     return sym[p->opr.op[0]->id.i] =
                        ex(p->opr.op[1]);
      case UMINUS: return -ex(p->opr.op[0]);
      case '+': return ex(p->opr.op[0]) + ex(p->opr.op[1]);
      case '-': return ex(p->opr.op[0]) - ex(p->opr.op[1]);
      case '*': return ex(p->opr.op[0]) * ex(p->opr.op[1]);
      case '/': return ex(p->opr.op[0]) / ex(p->opr.op[1]);
      case '<': return ex(p->opr.op[0]) < ex(p->opr.op[1]);
      case '>': return ex(p->opr.op[0]) > ex(p->opr.op[1]);
      case GE:  return ex(p->opr.op[0]) >= ex(p->opr.op[1]);
      case LE:  return ex(p->opr.op[0]) <= ex(p->opr.op[1]);
      case NE:  return ex(p->opr.op[0]) != ex(p->opr.op[1]);
      case EQ:  return ex(p->opr.op[0]) == ex(p->opr.op[1]);
    }
  }
  return 0;
}
```

Compiler

```
#include <stdio.h>
#include "calc3.h"
#include "y.tab.h"

static int lbl;

int ex(nodeType *p) {
    int lbl1, lbl2;

    if (!p) return 0;
    switch(p->type) {
    case typeCon:
        printf("\tpush\t%d\n", p->con.value);
        break;
    case typeId:
        printf("\tpush\t%c\n", p->id.i + 'a');
        break;
    case typeOpr:
        switch(p->opr.oper) {
        case WHILE:
            printf("L%03d:\n", lbl1 = lbl++);
            ex(p->opr.op[0]);
            printf("\tjz\tL%03d\n", lbl2 = lbl++);
            ex(p->opr.op[1]);
            printf("\tjmp\tL%03d\n", lbl1);
            printf("L%03d:\n", lbl2);
            break;
        case IF:
            ex(p->opr.op[0]);
            if (p->opr.nops > 2) {
                /* if else */
                printf("\tjz\tL%03d\n", lbl1 = lbl++);
                ex(p->opr.op[1]);
                printf("\tjmp\tL%03d\n", lbl2 = lbl++);
                printf("L%03d:\n", lbl1);
                ex(p->opr.op[2]);
                printf("L%03d:\n", lbl2);
            } else {
                /* if */
                printf("\tjz\tL%03d\n", lbl1 = lbl++);
                ex(p->opr.op[1]);
                printf("L%03d:\n", lbl1);
            }
            break;


        case PRINT:
            ex(p->opr.op[0]);
            printf("\tprint\n");
            break;
```

```
        case '=':
            ex(p->opr.op[1]);
            printf("\tpop\t%c\n", p->opr.op[0]->id.i + 'a');
            break;
        case UMINUS:
            ex(p->opr.op[0]);
            printf("\tneg\n");
            break;
        default:
            ex(p->opr.op[0]);
            ex(p->opr.op[1]);
            switch(p->opr.oper) {
            case '+':   printf("\tadd\n"); break;
            case '-':   printf("\tsub\n"); break;
            case '*':   printf("\tmul\n"); break;
            case '/':   printf("\tdiv\n"); break;
            case '<':   printf("\tcompLT\n"); break;
            case '>':   printf("\tcompGT\n"); break;
            case GE:    printf("\tcompGE\n"); break;
            case LE:    printf("\tcompLE\n"); break;
            case NE:    printf("\tcompNE\n"); break;
            case EQ:    printf("\tcompEQ\n"); break;
            }
        }
    }
    return 0;
}
```

Graph

```
/* source code courtesy of Frank Thomas Braun */

#include <stdio.h>
#include <string.h>

#include "calc3.h"
#include "y.tab.h"

int del = 1; /* distance of graph columns */
int eps = 3; /* distance of graph lines */

/* interface for drawing (can be replaced by "real" graphic using GD or
other) */
void graphInit (void);
void graphFinish();
void graphBox (char *s, int *w, int *h);
void graphDrawBox (char *s, int c, int l);
void graphDrawArrow (int c1, int l1, int c2, int l2);

/* recursive drawing of the syntax tree */
void exNode (nodeType *p, int c, int l, int *ce, int *cm);

/***********************************************************/

/* main entry point of the manipulation of the syntax tree */
int ex (nodeType *p) {
    int rte, rtm;

    graphInit ();
    exNode (p, 0, 0, &rte, &rtm);
    graphFinish();
    return 0;
}

/*c----cm---ce---->                       drawing of leaf-nodes
 l leaf-info
 */

/*c---------------cm--------------ce----> drawing of non-leaf-nodes
 l            node-info
 *                |
 *    -------------     ...----
 *    |       |               |
 *    v       v               v
 * child1  child2  ...     child-n
 *        che     che             che
 *cs      cs      cs              cs
 *
 */
```

```
void exNode
    (   nodeType *p,
        int c, int l,        /* start column and line of node */
        int *ce, int *cm     /* resulting end column and mid of node */
    )
{
    int w, h;           /* node width and height */
    char *s;            /* node text */
    int  cbar;          /*  "real"  start  column  of  node  (centred  above
subnodes) */
    int k;              /* child number */
    int che, chm;       /* end column and mid of children */
    int cs;             /* start column of children */
    char word[20];      /* extended node text */

    if (!p) return;

    strcpy (word, "???"); /* should never appear */
    s = word;
    switch(p->type) {
        case typeCon: sprintf (word, "c(%d)", p->con.value); break;
        case typeId:  sprintf (word, "id(%c)", p->id.i + 'A'); break;
        case typeOpr:
            switch(p->opr.oper){
                case WHILE:     s = "while"; break;
                case IF:        s = "if";    break;
                case PRINT:     s = "print"; break;
                case ';':       s = "[;]";     break;
                case '=':       s = "[=]";     break;
                case UMINUS:    s = "[_]";     break;
                case '+':       s = "[+]";     break;
                case '-':       s = "[-]";     break;
                case '*':       s = "[*]";     break;
                case '/':       s = "[/]";     break;
                case '<':       s = "[<]";     break;
                case '>':       s = "[>]";     break;
                case GE:        s = "[>=]";    break;
                case LE:        s = "[<=]";    break;
                case NE:        s = "[!=]";    break;
                case EQ:        s = "[==]";    break;
            }
            break;
    }

    /* construct node text box */
    graphBox (s, &w, &h);
    cbar = c;
    *ce = c + w;
    *cm = c + w / 2;

    /* node is leaf */
    if (p->type == typeCon || p->type == typeId || p->opr.nops == 0) {
        graphDrawBox (s, cbar, l);
        return;
    }
```

```
    /* node has children */
    cs = c;
    for (k = 0; k < p->opr.nops; k++) {
        exNode (p->opr.op[k], cs, l+h+eps, &che, &chm);
        cs = che;
    }

    /* total node width */
    if (w < che - c) {
        cbar += (che - c - w) / 2;
        *ce = che;
        *cm = (c + che) / 2;
    }

    /* draw node */
    graphDrawBox (s, cbar, l);

    /* draw arrows (not optimal: children are drawn a second time) */
    cs = c;
    for (k = 0; k < p->opr.nops; k++) {
        exNode (p->opr.op[k], cs, l+h+eps, &che, &chm);
        graphDrawArrow (*cm, l+h, chm, l+h+eps-1);
        cs = che;
    }
}

/* interface for drawing */

#define lmax 200
#define cmax 200

char graph[lmax][cmax]; /* array for ASCII-Graphic */
int graphNumber = 0;

void graphTest (int l, int c)
{   int ok;
    ok = 1;
    if (l < 0) ok = 0;
    if (l >= lmax) ok = 0;
    if (c < 0) ok = 0;
    if (c >= cmax) ok = 0;
    if (ok) return;
    printf ("\n+++error: l=%d, c=%d not in drawing rectangle 0, 0 ...
%d, %d",
        l, c, lmax, cmax);
    exit (1);
}

void graphInit (void) {
    int i, j;
    for (i = 0; i < lmax; i++) {
        for (j = 0; j < cmax; j++) {
            graph[i][j] = ' ';
        }
    }
}
```

```
void graphFinish() {
    int i, j;
    for (i = 0; i < lmax; i++) {
        for (j = cmax-1; j > 0 && graph[i][j] == ' '; j--);
        graph[i][cmax-1] = 0;
        if (j < cmax-1) graph[i][j+1] = 0;
        if (graph[i][j] == ' ') graph[i][j] = 0;
    }
    for (i = lmax-1; i > 0 && graph[i][0] == 0; i--);
    printf ("\n\nGraph %d:\n", graphNumber++);
    for (j = 0; j <= i; j++) printf ("\n%s", graph[j]);
    printf("\n");
}

void graphBox (char *s, int *w, int *h) {
    *w = strlen (s) + del;
    *h = 1;
}

void graphDrawBox (char *s, int c, int l) {
    int i;
    graphTest (l, c+strlen(s)-1+del);
    for (i = 0; i < strlen (s); i++) {
        graph[l][c+i+del] = s[i];
    }
}

void graphDrawArrow (int c1, int l1, int c2, int l2) {
    int m;
    graphTest (l1, c1);
    graphTest (l2, c2);
    m = (l1 + l2) / 2;
    while (l1 != m) {
        graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--;
    }
    while (c1 != c2) {
        graph[l1][c1] = '-'; if (c1 < c2) c1++; else c1--;
    }
    while (l1 != l2) {
        graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--;
    }
    graph[l1][c1] = '|';
}
```

More Lex

Strings

Quoted strings frequently appear in programming languages. Here is one way to match a string in lex:

```
%{
    char *yylval;
    #include <string.h>
%}
%%
\"[^"\n]*["\n] {
           yylval = strdup(yytext+1);
           if (yylval[yyleng-2] != '"')
               warning("improperly terminated string");
           else
               yylval[yyleng-2] = 0;
           printf("found '%s'\n", yylval);
       }
```

The above example ensures that strings don’t cross line boundaries, and removes enclosing quotes. If we wish to add escape sequences, such as **\n** or **\"**, start states simplify matters:

```
%{
char buf[100];
char *s;
%}
%x STRING

%%

\"              { BEGIN STRING; s = buf; }
<STRING>\\n     { *s++ = '\n'; }
<STRING>\\t     { *s++ = '\t'; }
<STRING>\\\"    { *s++ = '\"'; }
<STRING>\"      {
                  *s = 0;
                  BEGIN 0;
                  printf("found '%s'\n", buf);
                }
<STRING>\n      { printf("invalid string"); exit(1); }
<STRING>.       { *s++ = *yytext; }
```

Exclusive start state **STRING** is defined in the definition section. When the scanner detects a quote, the **BEGIN** macro shifts lex into the **STRING** state. Lex stays in the **STRING** state, recognizing only patterns that begin with **<STRING>**, until another **BEGIN** is executed. Thus, we have a mini-environment for scanning strings. When the trailing quote is recognized, we switch back to state 0, the initial state.

Reserved Words

If your program has a large collection of reserved words, it is more efficient to let lex simply match a string, and determine in your own code whether it is a variable or reserved word. For example, instead of coding

```
"if"            return IF;
"then"          return THEN;
"else"          return ELSE;

{letter}({letter}|{digit})*  {
         yylval.id = symLookup(yytext);
         return IDENTIFIER;
     }
```

where **symLookup** returns an index into the symbol table, it is better to detect reserved words and identifiers simultaneously, as follows:

```
{letter}({letter}|{digit})*  {
         int i;

         if ((i = resWord(yytext)) != 0)
             return (i);
         yylval.id = symLookup(yytext);
         return (IDENTIFIER);
     }
```

This technique significantly reduces the number of states required, and results in smaller scanner tables.

Debugging Lex

Lex has facilities that enable debugging. This feature may vary with different versions of lex, so you should consult documentation for details. The code generated by lex in file **lex.yy.c** includes debugging statements that are enabled by specifying command-line option "**-d**". Debug output in flex (a GNU version of lex) may be toggled on and off by setting **yy_flex_debug**. Output includes the rule applied and corresponding matched text. If you're running lex and yacc together, specify the following in your yacc input file:

```
extern int yy_flex_debug;
int main(void) {
    yy_flex_debug = 1;
    yyparse();
}
```

Alternatively, you may write your own debug code by defining functions that display information for the token value, and each variant of the **yylval** union. This is illustrated in the following example. When **DEBUG** is defined, the debug functions take effect, and a trace of tokens and associated values is displayed.

```
%union {
    int ivalue;
    ...
};

%{
#ifdef DEBUG
```

```
    int dbgToken(int tok, char *s) {
        printf("token %s\n", s);
        return tok;
    }
    int dbgTokenIvalue(int tok, char *s) {
        printf("token %s (%d)\n", s, yylval.ivalue);
        return tok;
    }
    #define RETURN(x) return dbgToken(x, #x)
    #define RETURN_ivalue(x) return dbgTokenIvalue(x, #x)
#else
    #define RETURN(x) return(x)
    #define RETURN_ivalue(x) return(x)
#endif
%}

%%

[0-9]+      {
                yylval.ivalue = atoi(yytext);
                RETURN_ivalue(INTEGER);
            }

"if"        RETURN(IF);
"else"      RETURN(ELSE);
```

More Yacc

Recursion

When specifying a list, we may do so using left recursion,

```
list:
      item
      | list ',' item
      ;
```

or right recursion:

```
list:
      item
      | item ',' list
```

If right recursion is used, all items on the list are pushed on the stack. After the last item is pushed, we start reducing. With left recursion, we never have more than three terms on the stack, since we reduce as we go along. For this reason, it is advantageous to use left recursion.

If-Else Ambiguity

A shift-reduce conflict that frequently occurs involves the **if-else** construct. Assume we have the following rules:

```
stmt:
   IF expr stmt
   | IF expr stmt ELSE stmt
```

```
    ...
```

and the following state:

```
IF expr stmt IF expr stmt . ELSE stmt
```

We need to decide if we should shift the **ELSE**, or reduce the **IF expr stmt** at the top of the stack. If we shift, then we have

```
IF expr stmt IF expr stmt . ELSE stmt
IF expr stmt IF expr stmt ELSE . stmt
IF expr stmt IF expr stmt ELSE stmt .
IF expr stmt stmt .
```

where the second **ELSE** is paired with the second **IF**. If we reduce, we have

```
IF expr stmt IF expr stmt . ELSE stmt
IF expr stmt stmt . ELSE stmt
IF expr stmt . ELSE stmt
IF expr stmt ELSE . stmt
IF expr stmt ELSE stmt .
```

where the second **ELSE** is paired with the first **IF**. Modern programming languages pair an **ELSE** with the most recent unpaired **IF**, so the former behavior is expected. This works well with yacc, since default behavior, when a shift-reduce conflict is encountered, is to shift.

Although yacc does the right thing, it also issues a shift-reduce warning message. To remove the message, give **IF-ELSE** a higher precedence than the simple **IF** statement:

```
%nonassoc IFX
%nonassoc ELSE

   stmt:
       IF expr stmt %prec IFX
       | IF expr stmt ELSE stmt
```

Error Messages

A nice compiler gives the user meaningful error messages. For example, not much information is conveyed by the following message:

```
syntax error
```

If we track the line number in lex, then we can at least give the user a line number:

```
void yyerror(char *s) {
    fprintf(stderr, "line %d: %s\n", yylineno, s);
}
```

When yacc discovers a parsing error, default action is to call **yyerror**, and then return from yylex with a return value of one. A more graceful action flushes the input stream to a statement delimiter, and continues to scan:

```
stmt:
         ';'
```

```
        | expr ';'
        | PRINT expr ';'
        | VARIABLE '=' expr ';
        | WHILE '(' expr ')' stmt
        | IF '(' expr ')' stmt %prec IFX
        | IF '(' expr ')' stmt ELSE stmt
        | '{' stmt_list '}'
        | error ';'
        | error '}'
        ;
```

The error token is a special feature of yacc that will match all input until the token following error is found. For this example, when yacc detects an error in a statement it will call **yyerror**, flush input up to the next semicolon or brace, and resume scanning.

Inherited Attributes

The examples so far have used synthesized attributes. At any point in a syntax tree we can determine the attributes of a node based on the attributes of its children. Consider the rule

```
expr: expr '+' expr           { $$ = $1 + $3; }
```

Since we are parsing bottom-up, the values of both operands are available, and we can determine the value associated with the left-hand side. An inherited attribute of a node depends on the value of a parent or sibling node. The following grammar defines a C variable declaration:

```
decl: type varlist
type: INT | FLOAT
varlist:
        VAR                   { setType($1, $0); }
        | varlist ',' VAR     { setType($3, $0); }
```

Here is a sample parse:

```
. INT VAR
INT . VAR
type . VAR
type VAR .
type varlist .
decl .
```

When we reduce **VAR** to **varlist**, we should annotate the symbol table with the type of the variable. However, the **type** is buried in the stack. This problem is resolved by indexing back into the stack. Recall that **$1** designates the first term on the right-hand side. We can index backwards, using **$0**, **$-1**, and so on. In this case, **$0** will do just fine. If you need to specify a token type, the syntax is **$<tokentype>0**, angle brackets included. In this particular example, care must be taken to ensure that type always precedes **varlist**.

Embedded Actions

Rules in yacc may contain embedded actions:

```
list: item1 { do_item1($1); } item2 { do_item2($3); } item3
```

Note that the actions take a slot in the stack, so **do_item2** must use **$3** to reference **item2**. Actually, this grammar is transformed by yacc into the following:

```
list: item1 _rule01 item2 _rule02 item3
_rule01: { do_item1($0); }
_rule02: { do_item2($0); }
```

Debugging Yacc

Yacc has facilities that enable debugging. This feature may vary with different versions of yacc, so you should consult documentation for details. The code generated by yacc in file **y.tab.c** includes debugging statements that are enabled by defining **YYDEBUG** and setting it to a non-zero value. This may also be done by specifying command-line option "**-t**". With **YYDEBUG** properly set, debug output may be toggled on and off by setting **yydebug**. Output includes tokens scanned and shift/reduce actions.

```
%{
#define YYDEBUG 1
%}
%%
...
%%
int main(void) {
    #if YYDEBUG
        yydebug = 1;
    #endif
    yylex();
}
```

In addition, you can dump the parse states by specifying command-line option "**-v**". States are dumped to file **y.output**, and are often useful when debugging a grammar. Alternatively, you can write your own debug code by defining a **TRACE** macro, as illustrated below. When **DEBUG** is defined, a trace of reductions, by line number, is displayed.

```
%{
#ifdef DEBUG
#define TRACE printf("reduce at line %d\n", __LINE__);
#else
#define TRACE
#endif
%}

%%

statement_list:
          statement
                { TRACE $$ = $1; }
        | statement_list statement
                { TRACE $$ = newNode(';', 2, $1, $2); }
        ;
```

Bibliography

Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman [1986]. Compilers, Prinicples, Techniques and Tools. Addison-Wesley, Reading, Massachusetts.

Gardner, Jim, Chris Retterath and Eric Gisin [1988]. MKS Lex & Yacc. Mortice Kern Systems Inc., Waterloo, Ontario, Canada.

Johnson, Stephen C. [1975]. Yacc: Yet Another Compiler Compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey. A PDF version is available at ePaperPress.

Lesk, M. E. and E. Schmidt [1975]. Lex – A Lexical Analyzer Generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey. A PDF version is available at ePaperPress.

Levine, John R., Tony Mason and Doug Brown [1992]. Lex & Yacc. O'Reilly & Associates, Inc. Sebastopol, California.