Chapter 1. Lex and Yacc

Lex and yacc help you write programs that transform structured input. This includes an enormous range of applications—anything from a simple text search program that looks for patterns in its input file to a C compiler that transforms a source program into optimized object code.

In programs with structured input, two tasks that occur over and over are dividing the input into meaningful units, and then discovering the relationship among the units. For a text search program, the units would probably be lines of text, with a distinction between lines that contain a match of the target string and lines that don't. For a C program, the units are variable names, constants, strings, operators, punctuation, and so forth. This division into units (which are usually called *tokens)* is known as *lexical analysis*, or *lexing* for short. Lex helps you by taking a set of descriptions of possible tokens and producing a C routine, which we call a *lexical analyzer*, or a *lexer*, or a *scanner* for short, that can identify those tokens. The set of descriptions you give to lex is called a *lex specification*.

The token descriptions that lex uses are known as *regular expressions*, extended versions of the familiar patterns used by the *grep* and *egrep* commands. Lex turns these regular expressions into a form that the lexer can use to scan the input text extremely fast, independent of the number of expressions that it is trying to match. A lex lexer is almost always faster than a lexer that you might write in C by hand.

As the input is divided into tokens, a program often needs to establish the relationship among the tokens. A C compiler needs to find the expressions, statements, declarations, blocks, and procedures in the program. This task is known as *parsing* and the list of rules that define the relationships that the program understands is a *grammar*. Yacc takes a concise description of a grammar and produces a C routine that can parse that grammar, a *parser*. The yacc parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar and also detects a syntax error whenever its input doesn't match any of the rules. A yacc parser is generally not as fast as a parser you could write by hand, but the ease in writing and modifying the parser is invariably worth any speed loss. The amount of time a program spends in a parser is rarely enough to be an issue anyway.

When a task involves dividing the input into units and establishing some relationship among those units, you should think of lex and yacc. (A search program is so simple that it doesn't need to do any parsing so it uses lex but

doesn't need yacc. We'll see this again in Chapter 2, where we build several applications using lex but not yacc.)

By now, we hope we've whetted your appetite for more details. We do not intend for this chapter to be a complete tutorial on lex and yacc, but rather a gentle introduction to their use.

The Simplest Lex Program

This lex program copies its standard input to its standard output:

```
%%
.|\n      ECHO;
%%
```

It acts very much like the UNIX *cat* command run with no arguments.

Lex automatically generates the actual C program code needed to handle reading the input file and sometimes, as in this case, writing the output as well.

Whether you use lex and yacc to build parts of your program or to build tools to aid you in programming, once you master them they will prove their worth many times over by simplifying difficult input handling problems, providing more easily maintainable code base, and allowing for easier "tinkering" to get the right semantics for your program.

Recognizing Words with Lex

Let's build a simple program that recognizes different types of English words. We start by identifying parts of speech (noun, verb, etc.) and will later extend it to handle multiword sentences that conform to a simple English grammar.

We start by listing a set of verbs to recognize:

is	am	are	were
was	be	being	been
do	does	did	will
would	should	can	could

has have had go

Example 1-1 shows a simple lex specification to recognize these verbs.

Example 1-1. Word recognizer ch1-02.l

```
%{
/*
 * this sample demonstrates (very) simple recognition:
 * a verb/not a verb.
 */

%}
%%

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

is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did  |
will |
would |
should |
can  |
could |
has  |
have |
had |
go         { printf("%s: is a verb\n", yytext); }
[a-zA-Z]+ { printf("%s: is not a verb\n", yytext); }

.|\n       { ECHO; /* normal default anyway */ }
%%

main()
{
      yylex() ;
}
```

Here's what happens when we compile and run this program. What we type is in **bold**.

```
% example1
did I have fun?
did: is a verb
I: is not a verb
have: is a verb
fun: is not a verb
?
^D
%
```

To explain what's going on, let's start with the first section:

```
%{
/*
 * This sample demonstrates very simple recognition:
 * a verb/not a verb.
 */


%}
```

This first section, the *definition section*, introduces any initial C program code we want copied into the final program. This is especially important if, for example, we have header files that must be included for code later in the file to work. We surround the C code with the special delimiters "%{" and "%}." Lex copies the material between "%{" and "%}" directly to the generated C file, so you may write any valid C code here.

In this example, the only thing in the definition section is some C comments. You might wonder whether we could have included the comments without the delimiters. Outside of "%{" and "%}", comments must be indented with whitespace for lex to recognize them correctly. We've seen some amazing bugs when people forgot to indent their comments and lex interpreted them as something else.

The %% marks the end of this section.

The next section is the *rules section*. Each rule is made up of two parts: a *pattern* and an *action*, separated by whitespace. The lexer that lex generates will execute the action when it recognizes the pattern. These patterns are UNIX-style regular expressions, a slightly extended version of the same expressions used by tools such as *grep*, *sed*, and *ed*. Chapter 6 describes all the rules for regular expressions. The first rule in our example is the following:

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

The square brackets, “[]”, indicate that any one of the characters within the brackets matches the pattern. For our example, we accept either “\t” (a tab character) or " " (a space). The “+” means that the pattern matches one or more consecutive copies of the subpattern that precedes the plus. Thus, this pattern describes whitespace (any combination of tabs and spaces.) The second part of the rule, the *action*, is simply a semicolon, a do-nothing C statement. Its effect is to ignore the input.

The next set of rules uses the “|” (vertical bar) action. This is a special action that means to use the same action as the next pattern, so all of the verbs use the action specified for the last one.[1]

Our first set of patterns is:

```
is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
should |
can |
could |
has |
have |
had |
go          { printf("%s: is a verb\n", yytext); }
```

Our patterns match any of the verbs in the list. Once we recognize a verb, we

execute the action, a C **printf** statement. The array **yytext** contains the text that matched the pattern. This action will print the recognized verb followed by the string “: is a verb\n”.

The last two rules are:

```
[a-zA-Z]+  { printf("%s:  is not a verb\n", yytext);    }

.|\n       { ECHO;  /* normal default anyway */ }
```

The pattern "[a-zA-Z]+" is a common one: it indicates any alphabetic string with at least one character. The "-" character has a special meaning when used inside square brackets: it denotes a range of characters beginning with the character to the left of the "-" and ending with the character to its right. Our action when we see one of these patterns is to print the matched token and the string ": is not a verb\n".

It doesn't take long to realize that any word that matches any of the verbs listed in the earlier rules will match this rule as well. You might then wonder why it won't execute both actions when it sees a verb in the list. And would both actions be executed when it sees the word "island," since "island" starts with "is"? The answer is that lex has a set of simple disambiguating rules. The two that make our lexer work are:

1. Lex patterns only match a given input character or string once.
2. Lex executes the action for the longest possible match for the current input. Thus, lex would see "island" as matching our all-inclusive rule because that was a longer match than "is."

If you think about how a lex lexer matches patterns, you should be able to see how our example matches only the verbs listed.

The last line is the default case. The special character "." (period) matches any single character other than a newline, and "\n" matches a newline character. The special action ECHO prints the matched pattern on the output, copying any punctuation or other characters. We explicitly list this case although it is the default behavior. We have seen some complex lexers that worked incorrectly because of this very feature, producing occasional strange output when the default pattern matched unanticipated input characters. (Even though there is a default action for unmatched input characters, well-written lexers invariably have explicit rules to match all possible input.)

The end of the rules section is delimited by another %%.

The final section is the *user subroutines section*, which can consist of any legal C code. Lex copies it to the C file after the end of the lex generated code. We have included a **main()** program.

```
%%


main()
{
        yylex();
}
```

The lexer produced by lex is a C routine called **yylex()**, so we call it.[2] Unless the actions contain explicit **return** statements, **yylex()** won't return until it has processed the entire input.

We placed our original example in a file called *ch1-02.l* since it is our second example. To create an executable program on our UNIX system we enter these commands:

```
% lex  ch1-02.l
% cc  lex.yy.c -o first -ll
```

Lex translates the lex specification into a C source file called *lex.yy.c* which we compiled and linked with the lex library *-ll*. We then execute the resulting program to check that it works as we expect, as we saw earlier in this section. Try it to convince yourself that this simple description really does recognize exactly the verbs we decided to recognize.

Now that we've tackled our first example, let's "spruce it up." Our second example, Example 1-2, extends the lexer to recognize different parts of speech.

Example 1-2. Lex example with multiple parts of speech ch1-03.l

```
%{
/*
 * We expand upon the first example by adding recognition of some
other
 * parts of speech.
 */

%}
%%

[\t ]+                    /* ignore whitespace */ ;
is |
am |
are |
were |
was |
be |
```

```
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go       { printf("%s: is a verb\n", yytext); }

very |
simply |
gently |
quietly |
calmly |
angrily   { printf("%s: is an adverb\n", yytext); }

to |
from |
behind |
above |
below |
between
below     { printf("%s: is a preposition\n", yytext); }

if |
then |
and |
but |
or       { printf("%s: is a conjunction\n", yytext); }

their |
my |
your |
his |
her |
its      { printf("%s: is a adjective\n", yytext); }

I |
you |
he |
she  |
we  |
they       { printf("%s: is a pronoun\n", yytext); }


[a-zA-Z]+ {
       printf("%s:  don't recognize, might be a noun\n", yytext);
```

```
        }
.|\n        { ECHO;/* normal default anyway */ }

%%

main()
{
     yylex();
}
```

Symbol Tables

Our second example isn't really very different. We list more words than we did before, and in principle we could extend this example to as many words as we want. It would be more convenient, though, if we could build a table of words as the lexer is running, so we can add new words without modifying and recompiling the lex program. In our next example, we do just that—allow for the dynamic declaration of parts of speech as the lexer is running, reading the words to declare from the input file. Declaration lines start with the name of a part of speech followed by the words to declare. These lines, for example, declare four nouns and three verbs:

```
noun dog cat horse cow

verb chew eat lick
```

The table of words is a simple *symbol table*, a common structure in lex and yacc applications. A C compiler, for example, stores the variable and structure names, labels, enumeration tags, and all other names used in the program in its symbol table. Each name is stored along with information describing the name. In a C compiler the information is the type of symbol, declaration scope, variable type, etc. In our current example, the information is the part of speech.

Adding a symbol table changes the lexer quite substantially. Rather than putting separate patterns in the lexer for each word to match, we have a single pattern that matches any word and we consult the symbol table to decide which part of speech we've found. The names of parts of speech (noun, verb, etc.) are now "reserved words" since they introduce a declaration line. We still have a separate lex pattern for each reserved word. We also have to add symbol table maintenance routines, in this case **add_word()**, which puts a new word into the symbol table, and **lookup_word( )**, which looks up a word which should already be entered.

In the program's code, we declare a variable **state** that keeps track of whether we're looking up words, state LOOKUP, or declaring them, in which case **state** remembers what kind of words we're declaring. Whenever we see a

line starting with the name of a part of speech, we set the state to declare that kind of word; each time we see a \n we switch back to the normal lookup state.

Example 1-3 shows the definition section.

Example 1-3. Lexer with symbol table (part 1 of 3) ch1-04.l

```
%{
/*
 * Word recognizer with a symbol table.
 */

enum {
      LOOKUP =0, /* default - looking rather than defining. */
      VERB,
      ADJ,
      ADV,
      NOUN,
      REP,
      PRON,
      CONJ
};

int state;

int add_word(int type, char *word);
int lookup_word(char *word);
%}
```

We define an *enum* in order to use in our table to record the types of individual words, and to declare a variable **state**. We use this enumerated type both in the state variable to track what we're defining and in the symbol table to record what type each defined word is. We also declare our symbol table routines.

Example 1-4 shows the rules section.

Example 1-4. Lexer with symbol table (part 2 of 3) ch1-04.l

```
%%
\n    { state = LOOKUP; }   /* end of line, return to default
state */

      /* whenever a line starts with a reserved part of speech
name */
      /* start defining words of that type */
^verb { state = VERB; }
^adj  { state = ADJ; }
^adv  { state = ADV; }
^noun { state = NOUN; }
^prep { state = PREP; }
```

```
^pron { state = PRON; }
^conj { state = CONJ; }


[a-zA-Z]+ {
                /* a normal word, define it or look it up */
           if(state != LOOKUP) {
                /* define the current word */
                add_word(state, yytext);
              } else {
                 switch(lookup_word(yytext)) {
                 case VERB: printf("%s: verb\n", yytext); break;
                 case ADJ: printf("%s: adjective\n", yytext);
break;
                 case ADV: printf("%s: adverb\n", yytext); break;
                 case NOUN: printf("%s: noun\n", yytext); break;
                 case PREP: printf("%s: preposition\n", yytext);
break;
                 case PRON: printf("%s: pronoun\n", yytext);
break;
                 case CONJ: printf("%s: conjunction\n", yytext);
break;
                 default:
                         printf("%s: don't recognize\n", yytext);
                         break;
                 }
          }
         }

.    /* ignore anything else */ ;

%%
```

For declaring words, the first group of rules sets the state to the type corresponding to the part of speech being declared. (The caret, "^", at the beginning of the pattern makes the pattern match only at the beginning of an input line.) We reset the state to **LOOKUP** at the beginning of each line so that after we add new words interactively we can test our table of words to determine if it is working correctly. If the state is **LOOKUP** when the pattern "[a-zA-Z]+" matches, we look up the word, using **lookup_word()**, and if found print out its type. If we're in any other state, we define the word with **add_word()**.

The user subroutines section in Example 1-5 contains the same skeletal **main()** routine and our two supporting functions.

Example 1-5. Lexer with symbol table (part 3 of 3) ch1-04.l

```
main()
{
```

```
	yylex();
}

/* define a linked list of words and types */
struct word {
	char *word_name;
	int word_type;
	struct word *next;
};

struct word *word_list; /* first element in word list */

extern void *malloc() ;

int
add_word(int type, char *word)
{
	struct word *wp;

	if(lookup_word(word) != LOOKUP) {
		printf("!!! warning: word %s already defined \n",
word);
		return 0;
	}

	/* word not there, allocate a new entry and link it on the
list */

	wp = (struct word *) malloc(sizeof(struct word));

	wp->next = word_list;

	/* have to copy the word itself as well */

	wp->word_name = (char *) malloc(strlen(word)+1);
	strcpy(wp->word_name, word);
	wp->word_type = type;
	word_list = wp;
	return 1;  /* it worked */
}

int
lookup_word(char *word)
{
	struct word *wp = word_list;

	/* search down the list looking for the word */
	for(; wp; wp = wp->next) {
	if(strcmp(wp->word_name, word) == 0)
		return wp->word_type;
	}

	return LOOKUP;	/* not found */
```

```
}
```

These last two functions create and search a linked list of words. If there are a lot of words, the functions will be slow since, for each word, they might have to search through the entire list. In a production environment we would use a faster but more complex scheme, probably using a hash table. Our simple example does the job, albeit slowly.

Here is an example of a session we had with our last example:

```
            verb is am are was were be being been do
is
is: verb
noun dog cat horse cow
verb chew eat lick
verb run stand sleep
dog run
dog: noun
run: verb
chew eat sleep cow horse
chew: verb
eat: verb
sleep: verb
cow: noun
horse: noun
verb talk
talk
talk: verb
```

We strongly encourage you to play with this example until you are satisfied you understand it.

Grammars

For some applications, the simple kind of word recognition we've already done may be more than adequate; others need to recognize specific sequences of tokens and perform appropriate actions. Traditionally, a description of such a set of actions is known as a *grammar*. It seems especially appropriate for our example. Suppose that we wished to recognize common sentences. Here is a list of simple sentence types:

noun verb.

noun verb noun.

At this point, it seems convenient to introduce some notation for describing grammars. We use the right facing arrow, "→", to mean that a particular set of tokens can be replaced by a new symbol.[3] For instance:

subject → *noun* | *pronoun*

would indicate that the new symbol *subject* is either a noun or a pronoun. We

haven't changed the meaning of the underlying symbols; rather we have built our new symbol from the more fundamental symbols we've already defined. As an added example we could define an object as follows:

object → *noun*

While not strictly correct as English grammar, we can now define a sentence:

sentence → *subject verb object*

Indeed, we could expand this definition of sentence to fit a much wider variety of sentences. However, at this stage we would like to build a yacc grammar so we can test our ideas out interactively. Before we introduce our yacc grammar, we must modify our lexical analyzer in order to return values useful to our new parser.

Parser-Lexer Communication

When you use a lex scanner and a yacc parser together, the parser is the higher level routine. It calls the lexer **yylex()** whenever it needs a token from the input. The lexer then scans through the input recognizing tokens. As soon as it finds a token of interest to the parser, it returns to the parser, returning the token's code as the value of **yylex()**.

Not all tokens are of interest to the parser—in most programming languages the parser doesn't want to hear about comments and whitespace, for example. For these ignored tokens, the lexer doesn't return so that it can continue on to the next token without bothering the parser.

The lexer and the parser have to agree what the token codes are. We solve this problem by letting yacc define the token codes. The tokens in our grammar are the parts of
speech: **NOUN**, **PRONOUN**, **VERB**, **ADVERB**, **ADJECTIVE**, **PREPOSITI**

ON, and **CONJUNCTION**. Yacc defines each of these as a small integer using a preprocessor *#define*. Here are the definitions it used in this example:

```
# define NOUN 257
# define PRONOUN 258
# define VERB 259
# define ADVERB 260
# define ADJECTIVE 261
# define PREPOSITION 262
# define CONJUNCTION 263
```

Token code zero is always returned for the logical end of the input. Yacc doesn't define a symbol for it, but you can yourself if you want.

Yacc can optionally write a C header file containing all of the token definitions. You include this file, called *y.tab.h* on UNIX systems and *ytab.h* or *yytab.h* on MS-DOS, in the lexer and use the preprocessor symbols in your lexer action code.

The Parts of Speech Lexer

Example 1-6 shows the declarations and rules sections of the new lexer.

Example 1-6. Lexer to be called from the parser ch1-05.l

```
%{
/*
 * We now build a lexical analyzer to be used by a higher-level
parser.
 */

#include "y.tab.h"    /* token codes from the parser */

#define   LOOKUP 0   /* default - not a defined word type. */

int state;

%}

%%

\n    { state = LOOKUP; }

\.\n  {     state = LOOKUP;
            return 0;  /* end of sentence */
      }
```

```
^verb { state = VERB; }
^adj  { state = ADJECTIVE; }
^adv  { state = ADVERB; }
^noun { state = NOUN; }
^prep { state = PREPOSITION; }
^pron { state = PRONOUN; }
^conj { state = CONJUNCTION; }

[a-zA-Z]+ {
           if(state != LOOKUP) {
            add_word(state, yytext);
           } else {
            switch(lookup_word(yytext)) {
            case VERB:
              return(VERB);
            case ADJECTIVE:
              return(ADJECTIVE);
            case ADVERB:
              return(ADVERB);
            case NOUN:
              return(NOUN);
            case PREPOSITION:
              return(PREPOSITION);
            case PRONOUN:
              return(PRONOUN);
            case CONJUNCTION:
              return(CONJUNCTION);
            default:
              printf("%s: don't recognize\n", yytext);
              /* don't return, just ignore it */
            }
            }
          }
.    ;

%%
```

... same add_word() and lookup_word() as before ...

There are several important differences here. We've changed the part of speech names used in the lexer to agree with the token names in the parser. We have also added **return** statements to pass to the parser the token codes for the words that it recognizes. There aren't any **return** statements for the tokens that define new words to the lexer, since the parser doesn't care about them.

These return statements show that **yylex()** acts like a coroutine. Each time the parser calls it, it takes up processing at the exact point it left off. This allows us to examine and operate upon the input stream incrementally. Our first programs

didn’t need to take advantage of this, but it becomes more useful as we use the lexer as part of a larger program.

We added a rule to mark the end of a sentence:

```
\.\n  {       state = LOOKUP;
                return 0;  /* end of sentence */
      }
```

The backslash in front of the period quotes the period, so this rule matches a period followed by a newline. The other change we made to our lexical analyzer was to omit the **main()** routine as it will now be provided within the parser.

A Yacc Parser

Finally, Example 1-7 introduces our first cut at the yacc grammar.

Example 1-7. Simple yacc sentence parser ch1-05.y

```
%{
/*
 * A lexer for the basic grammar to use for recognizing English
sentences.
 */
#include <stdio.h>
%}

%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION

%%
sentence: subject VERB object{ printf("Sentence is valid.\n"); }
      ;

subject:    NOUN
      |     PRONOUN
      ;

object:           NOUN

      ;
%%

extern FILE *yyin;

main()
{
     do
       {
           yyparse();
```

```
      }
         while (!feof(yyin));
}

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

The structure of a yacc parser is, not by accident, similar to that of a lex lexer. Our first section, the definition section, has a literal code block, enclosed in "%{" and "%}". We use it here for a C comment (as with lex, C comments belong inside C code blocks, at least within the definition section) and a single include file.

Then come definitions of all the tokens we expect to receive from the lexical analyzer. In this example, they correspond to the eight parts of speech. The name of a token does not have any intrinsic meaning to yacc, although well-chosen token names tell the reader what they represent. Although yacc lets you use any valid C identifier name for a yacc symbol, universal custom dictates that token names be all uppercase and other names in the parser mostly or entirely lowercase.

The first %% indicates the beginning of the rules section. The second %% indicates the end of the rules and the beginning of the user subroutines section. The most important subroutine is **main()** which repeatedly calls **yyparse()** until the lexer's input file runs out. The routine **yyparse()** is the parser generated by yacc, so our main program repeatedly tries to parse sentences until the input runs out. (The lexer returns a zero token whenever it sees a period at the end of a line; that's the signal to the parser that the input for the current parse is complete.)

The Rules Section

The rules section describes the actual grammar as a set of *production rules* or simply *rules*. (Some people also call them *productions*.) Each rule consists of a single name on the left-hand side of the ":" operator, a list of symbols and action code on the right-hand side, and a semicolon indicating the end of the rule. By default, the first rule is the highest-level rule. That is, the parser attempts to find a list of tokens which match this initial rule, or more commonly, rules found from the initial rule. The expression on the right-hand side of the rule is a list of zero or more names. A typical simple rule has a single symbol on the right-hand side as in the **object** rule which is defined to be

a **NOUN**. The symbol on the left-hand side of the rule can then be used like a token in other rules. From this, we build complex grammars.

In our grammar we use the special character "|", which introduces a rule with the same left-hand side as the previous one. It is usually read as "or," e.g., in our grammar a subject can be either a **NOUN** or a **PRONOUN**. The *action* part of a rule consists of a C block, beginning with "{" and ending with "}". The parser executes an action at the end of a rule as soon as the rule matches. In our **sentence** rule, the action reports that we've successfully parsed a sentence. Since **sentence** is the top-level symbol, the entire input must match a **sentence**. The parser returns to its caller, in this case the main program, when the lexer reports the end of the input. Subsequent calls to **yyparse()** reset the state and begin processing again. Our example prints a message if it sees a "subject VERB object" list of input tokens. What happens if it sees "subject subject" or some other invalid list of tokens? The parser calls **yyerror()**, which we provide in the user subroutines section, and then recognizes the special rule **error**. You can provide error recovery code that tries to get the parser back into a state where it can continue parsing. If error recovery fails or, as is the case here, there is no error recovery code, **yyparse()** returns to the caller after it finds an error.

The third and final section, the user subroutines section, begins after the second %%. This section can contain any C code and is copied, verbatim, into the resulting parser. In our example, we have provided the minimal set of functions necessary for a yacc-generated parser using a lex-generated lexer to compile: **main()** and **yyerror()**. The main routine keeps calling the parser until it reaches the end-of-file on **yyin**, the lex input file. The only other necessary routine is **yylex()** which is provided by our lexer.

In our final example of this chapter, Example 1-8, we expand our earlier grammar to recognize a richer, although by no means complete, set of sentences. We invite you to experiment further with this example—you will see how difficult English is to describe in an unambiguous way.

Example 1-8. Extended English parser ch1-06.y

```
%{
#include <stdio.h>
%}

%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION

%%

sentence: simple_sentence  { printf("Parsed a simple
sentence.\n"); }
```

```
      | compound_sentence { printf("Parsed a compound
sentence.\n"); }
      ;

simple_sentence: subject verb object
      |     subject verb object prep_phrase
      ;

compound_sentence: simple_sentence CONJUNCTION simple_sentence
      |     compound_sentence CONJUNCTION simple_sentence
      ;

subject:    NOUN
      |     PRONOUN
      |     ADJECTIVE subject
      ;

verb:       VERB
      |     ADVERB VERB
      |     verb VERB
      ;

object:           NOUN
      |     ADJECTIVE object
      ;

prep_phrase:     PREPOSITION NOUN
      ;

%%

extern FILE *yyin;

main()
{
      do
        {
            yyparse();
      }
        while(!feof(yyin));
}

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

We have expanded our **sentence** rule by introducing a traditional grammar formulation from elementary school English class: a sentence can be either a simple sentence or a compound sentence which contains two or more independent clauses joined with a coordinating conjunction. Our current lexical

analyzer does not distinguish between a coordinating conjunction e.g., “and,” “but,” “or,” and a subordinating conjunction (e.g., “if”).

We have also introduced *recursion* into this grammar. Recursion, in which a rule refers directly or indirectly to itself, is a powerful tool for describing grammars, and we use the technique in nearly every yacc grammar we write. In this instance the **compound_sentence** and **verb** rules introduce the recursion. The former rule simply states that a **compound_sentence** is two or more simple sentences joined by a conjunction. The first possible match,

```
simple_sentence CONJUNCTION simple_sentence
```

defines the “two clause” case while

```
compound_sentence CONJUNCTION simple_sentence
```

defines the “more than two clause case.” We will discuss recursion in greater detail in later chapters.

Although our English grammar is not particularly useful, the techniques for identifying words with lex and then for finding the relationship among the words with yacc are much the same as we’ll use in the practical applications in later chapters. For example, in this C language statement,

```
if( a == b ) break; else func(&a);
```

a compiler would use lex to identify the tokens **if**, **(**, **a**, **==**, and so forth, and

then use yacc to establish that “a == b” is the expression part of an **if** statement, the **break** statement was the “true” branch, and the function call its “false” branch.

Running Lex and Yacc

We conclude by describing how we built these tools on our system.

We called our various lexers *ch1-N.l*, where *N* corresponded to a particular lex specification example. Similarly, we called our parsers *ch1-M.y*, where again *M* is the number of an example. Then, to build the output, we did the following in UNIX:

```
% lex ch1-n
        .l
% yacc -d ch1-m
        .y
% cc -c lex.yy.c y.tab.c
```

```
% cc -o example-m.n
        lex.yy.o y.tab.o -ll
```

The first line runs lex over the lex specification and generates a file, *lex.yy.c*, which contains C code for the lexer. In the second line, we use yacc to generate both *y.tab.c* and *y.tab.h* (the latter is the file of token definitions created by the *-d* switch.) The next line compiles each of the two C files. The final line links them together and uses the routines in the lex library *libl.a*, normally in */usr/lib/libl.a* on most UNIX systems. If you are not using AT&T lex and yacc, but one of the other implementations, you may be able to simply substitute the command names and little else will change. (In particular, Berkeley yacc and flex will work merely by changing the *lex* and *yacc* commands to *byacc* and *flex*, and removing the *-ll* linker flag.) However, we know of far too many differences to assure the reader that this is true. For example, if we use the GNU replacement bison instead of yacc, it would generate two files called *ch1-M.tab.c* and *ch1-M.tab.h*. On systems with more restrictive naming, such as MS-DOS, these names will change

(typically *ytab.c* and *ytab.h*.) See Appendices A through H for details on the various lex and yacc implementations.

Lex vs. Hand-written Lexers

People have often told us that writing a lexer in C is so easy that there is no

point in going to the effort to learn lex. Maybe and maybe not. Example 1-9 shows a lexer written in C suitable for a simple command language that handles commands, numbers, strings, and new lines, ignoring white space and comments. Example 1-10 is an equivalent lexer written in lex. The lex version is a third the length of the C lexer. Given the rule of thumb that the number of bugs in a program is roughly proportional to its length, we'd expect the C version of the lexer to take three times as long to write and debug.

Example 1-9. A lexer written in C

```
#include <stdio.h>
#include <ctype.h>
char *progname;

#define NUMBER 400
#define COMMENT 401
#define TEXT 402
#define COMMAND 403

main(argc,argv)
int argc;
```

```
    char *argv[];
    {
    int val;
    while(val = lexer()) printf("value is %d\n",val);
    }

    lexer()
    {
        int c;

        while ((c=getchar()) == ' ' || c == '\t')
            ;
        if (c == EOF)
            return 0;
        if (c == '.' || isdigit(c)) {   /* number */
            while ((c = getchar()) != EOF && isdigit (c)) ;
        if (c == '.') while ((c = getchar()) != EOF && isdigit
(c);
            ungetc(c, stdin);
            return NUMBER;
        }
        if ( c =='#'){/* comment */
            while ((c = getchar()) != EOF && c != '\n');
            ungetc(c,stdin);
            return COMMENT;
        }
        if ( c =='"'){/* literal text */
            while ((c = getchar()) != EOF &&
           c != '"' && c != '\n') ;
            if(c == '\n') ungetc(c,stdin);
            return TEXT;
        }
        if ( isalpha(c)) { /* check to see if it is a command */
            while ((c = getchar()) != EOF && isalnum(c));
            ungetc(c, stdin);
            return COMMAND;
        }
        return c;
    }
```

Example 1-10. The same lexer written in lex

```
    %{
    #define NUMBER 400
    #define COMMENT 401
    #define TEXT 402
    #define COMMAND 403
    %}
    %%
    [ \t]+                  ;
    [0-9]+                  |
    [0-9]+\.[0-9]+          |
```

```
\.[0-9]+                { return NUMBER; }
#.*                      { return COMMENT; }
\"[^\"\n]*\"             { return TEXT;   }
[a-zA-Z][a-zA-Z0-9]+     { return COMMAND;}
\n                       { return '\n'; }
%%
#include <stdio.h>

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

while(val = yylex()) printf ("value is %d\n",val);
}
```

Lex handles some subtle situations in a natural way that are difficult to get right in a hand written lexer. For example, assume that you're skipping a C language comment. To find the end of the comment, you look for a "*", then check to see that the next character is a "/". If it is, you're done, if not you keep scanning. A very common bug in C lexers is not to consider the case that the next character is itself a star, and the slash might follow that. In practice, this means that some comments fail:

```
/** comment **/
```

(We've seen this exact bug in a sample, hand-written lexer distributed with one version of yacc!)

Once you get comfortable with lex, we predict that you'll find, as we did, that it's so much easier to write in lex that you'll never write another handwritten lexer.

In the next chapter we delve into the workings of lex more deeply. In the chapter following we'll do the same for yacc. After that we'll consider several larger examples which describe many of the more complex issues and features of lex and yacc.

Exercises

1. Extend the English-language parser to handle more complex syntax: prepositional phrases in the subject, adverbs modifying adjectives, etc.
2. Make the parser handle compound verbs better, e.g., "has seen." You might want to add new word and token types AUXVERB for auxiliary verbs.

3. Some words can be more than one part of speech, e.g., “watch,” “fly,” “time,” or “bear.” How could you handle them? Try adding a new word and token type NOUN_OR_VERB, and add it as an alternative to the rules for **subject**, **verb**, and **object**. How well does this work?
4. When people hear an unfamiliar word, they can usually guess from the context what part of speech it is. Could the lexer characterize new words on the fly? For example, a word that ends in “ing” is probably a verb, and one that follows “a” or “the” is probably a noun or an adjective.
5. Are lex and yacc good tools to use for building a realistic English-language parser? Why not?

Chapter 2. Using Lex

In the first chapter we demonstrated how to use lex and yacc. We now show how to use lex by itself, including some examples of applications for which lex is a good tool. We're not going to explain every last detail of lex here; consult Chapter 6, *A Reference for Lex Specifications* .

Lex is a tool for building *lexical analyzers* or *lexers.* A lexer takes an arbitrary input stream and tokenizes it, i.e., divides it up into lexical tokens. This tokenized output can then be processed further, usually by yacc, or it can be the "end product." In Chapter 1 we demonstrated how to use it as an intermediate step in our English grammar. We now look more closely at the details of a lex specification and how to use it; our examples use lex as the final processing step rather than as an intermediate step which passes information on to a yacc-based parser.

When you write a *lex specification*, you create a set of patterns which lex matches against the input. Each time one of the patterns matches, the lex program invokes C code that you provide which does something with the matched text. In this way a lex program divides the input into strings which we call tokens. Lex itself doesn't produce an executable program; instead it translates the lex specification into a file containing a C routine called **yylex()**. Your program calls **yylex()** to run the lexer.

Using your regular C compiler, you compile the file that lex produced along with any other files and libraries you want. (Note that lex and the C compiler don't even have to run on the same computer. The authors have often taken the C code from UNIX lex to other computers where lex is not available but C is.)

Regular Expressions

Before we describe the structure of a lex specification, we need to describe regular expressions as used by lex. Regular expressions are widely used within the UNIX environment, and lex uses a rich regular expression language.

A *regular expression* is a pattern description using a "meta" language, a language that you use to describe particular patterns of interest. The characters used in this metalanguage are part of the standard ASCII character set used in UNIX and MS-DOS, which can sometimes lead to confusion. The characters that form regular expressions are:

.

Matches any single character except the newline character (“\n”).

*

Matches zero or more copies of the preceding expression.

[]

A *character class* which matches any character within the brackets. If the first character is a circumflex (“^”) it changes the meaning to match any character *except* the ones within the brackets. A dash inside the square brackets indicates a character range, e.g., “[0-9]” means the same thing as “[0123456789]”. A “-” or “]” as the first character after the “[” is interpreted literally, to let you include dashes and square brackets in character classes. POSIX introduces other special square bracket constructs useful when handling non-English alphabets. See Appendix H, *POSIX Lex and Yacc*, for details. Other metacharacters have no special meaning within square brackets except that C escape sequences starting with “\” are recognized.

^

Matches the beginning of a line as the first character of a regular expression. Also used for negation within square brackets.

$

Matches the end of a line as the last character of a regular expression.

{}

Indicates how many times the previous pattern is allowed to match when containing one or two numbers. For example:

```
A{1,3}
```

matches one to three occurrences of the letter A. If they contain a name, they refer to a substitution by that name.

\

Used to escape metacharacters, and as part of the usual C escape sequences, e.g., “\n” is a newline character, while “*” is a literal asterisk.

+

Matches one or more occurrence of the preceding regular expression. For example:

```
[0-9]+
```

matches “1”, “111”, or “123456” but not an empty string. (If the plus sign were an asterisk, it would also match the empty string.)

?

Matches zero or one occurrence of the preceding regular expression. For example:

-?[0-9]+

matches a signed number including an optional leading minus.

|

Matches either the preceding regular expression or the following regular expression. For example:

cow|pig|sheep

matches any of the three words.

“...”

Interprets everything within the quotation marks literally—meta-characters other than C escape sequences lose their meaning.

/

Matches the preceding regular expression but only if followed by the following regular expression. For example:

0/1

matches “0” in the string “01” but would not match anything in the strings “0” or “02”. The material matched by the pattern following the slash is not “consumed” and remains to be turned into subsequent tokens. Only one slash is permitted per pattern.

()

Groups a series of regular expressions together into a new regular expression. For example:

(01)

represents the character sequence 01. Parentheses are useful when building up complex patterns with *, +, and |.

Note that some of these operators operate on single characters (e.g., []) while others operate on regular expressions. Usually, complex regular expressions are built up from simple regular expressions.

Examples of Regular Expressions

We are ready for some examples. First, we’ve already shown you a regular expression for a “digit”:

```
[0-9]
```

We can use this to build a regular expression for an integer:

```
[0-9]+
```

We require at least one digit. This would have allowed no digits at all:

```
[0-9]*
```

Let's add an optional unary minus:

```
-?[0-9]+
```

We can then expand this to allow decimal numbers. First we will specify a decimal number (for the moment we insist that the last character always be a digit):

```
[0-9]*\.[0-9]+
```

Notice the "\" before the period to make it a literal period rather than a wild card character. This pattern matches "0.0", "4.5", or ".31415". But it won't match "0" or "2". We'd like to combine our definitions to match them as well. Leaving out our unary minus, we could use:

```
([0-9]+)|([0-9]*\.[0-9]+)
```

We use the grouping symbols "()" to specify what the regular expressions are for the "|" operation. Now let's add the unary minus:

```
-?(([0-9]+)|([0-9]*\.[0-9]+))
```

We can expand this further by allowing a float-style exponent to be specified as well. First, let's write a regular expression for an exponent:

```
[eE] [-+]?[0-9]+
```

This matches an upper- or lowercase letter E, then an optional plus or minus sign, then a string of digits. For instance, this will match "e12" or "E-3". We can then use this expression to build our final expression, one that specifies a real number:

```
-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
```

Our expression makes the exponent part optional. Let's write a real lexer that uses this expression. Nothing fancy, but it examines the input and tells us each time it matches a number according to our regular expression.

Example 2-1 shows our program.

Example 2-1. Lex specification for decimal numbers

```
%%
[\n\t ]     ;

-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
printf("number\n"); }

.     ECHO;
%%
main()
{
      yylex();
}
```

Our lexer ignores whitespace and echoes any characters it doesn't recognize as parts of a number to the output. For instance, here are the results with something close to a valid number:

```
.65ea12

number

eanumber
```

We encourage you to play with this and all our examples until you are satisfied you understand how they work. For instance, try changing the expression to recognize a unary plus as well as a unary minus.

Another common regular expression is one used by many scripts and simple configuration files, an expression that matches a comment starting with a sharp sign, "#".[4] We can build this regular expression as:

```
#.*
```

The "." matches any character except newline and the "*" means match zero or more of the preceding expression. This expression matches anything on the comment line up to the newline which marks the end of the line.

Finally, here is a regular expression for matching quoted strings:

```
\"[^"\n]*["\n]
```

It might seem adequate to use a simpler expression such as:

```
\".*\"
```

Unfortunately, this causes lex to match incorrectly if there are two strings on the same input line. For instance:

```
"how" to "do"
```

would match as a single pattern since "*" matches as much as possible. Knowing this, we then might try:

```
\"[^"]*\"
```

This regular expression can cause lex to overflow its internal input buffer if the trailing quotation mark is not present, because the expression "[^"]*" matches any character except a quote, including "\n". So if the user leaves out a quote by mistake, the pattern could potentially scan through the entire input file looking for another quote. Since the token is stored in a fixed size buffer,[5] sooner or later the amount read will be bigger than the buffer and the lexer will crash. For

example:

```
"How", she said, "is it that I cannot find it.
```

would match the second quoted string continuing until it saw another quotation mark. This might be hundreds or thousands of characters later. So we add the new rule that a quoted string must not extend past one line and end up with the complex (but safer) regular expression shown above. Lex can handle longer strings, but in a different way. See the section on yymore in Chapter 6, *A Reference for Lex Specifications.*

A Word Counting Program

Let's look at the actual structure of a lex specification. We will use a basic word count program (similar to the UNIX program *wc*).

A lex specification consists of three sections: a definition section, a rules section, and a user subroutines section. The first section, the definition section, handles options lex will be using in the lexer, and generally sets up the execution environment in which the lexer operates.

The definition section for our word count example is:

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}
```

```
word [^ \t\n]+
eol \n
```

The section bracketed by "%{" and "%}" is C code which is copied verbatim into the lexer. It is placed early on in the output code so that the data definitions contained here can be referenced by code within the rules section. In our example, the code block here declares three variables used within the program to track the number of characters, words, and lines encountered.

The last two lines are *definitions*. Lex provides a simple substitution mechanism to make it easier to define long or complex patterns. We have added two definitions here. The first provides our description of a word: any non-empty combination of characters except space, tab, and newline. The second describes our end-of-line character, newline. We use these definitions in the second section of the file, the *rules section*.

The rules section contains the patterns and actions that specify the lexer. Here is our sample word count's rules section:

```
%%
{word}      { wordCount++; charCount += yyleng; }
{eol} { charCount++; lineCount++; }
.     charCount++;
```

The beginning of the rules section is marked by a "%%". In a pattern, lex replaces the name inside the braces {} with *substitution*, the actual regular expression in the definition section. Our example increments the number of words and characters after the lexer has recognized a complete word.

The actions which consist of more than one statement are enclosed in braces to make a C language compound statement. Most versions of lex take everything after the pattern to be the action, while others only read the first statement on the line and silently ignore anything else. To be safe, and to make the code clearer, always use braces if the action is more than one statement or more than one line long.

It is worth repeating that lex always tries to match the longest possible string. Thus, our sample lexer would recognize the string "well-being" as a single word.

Our sample also uses the lex internal variable **yyleng** which contains the length of the string our lexer recognized. If it matched well-being, **yyleng** would be 10.

When our lexer recognizes a newline, it will increment both the character count and the line count. Similarly, if it recognizes any other character it increments the character count. For this lexer, the only "other characters" it could recognize would be space or tab; anything else would match the first regular expression and be counted as a word.

The lexer always tries to match the longest possible string, but when there are two possible rules that match the same length, the lexer uses the earlier rule in the lex specification. Thus, the word "I" would be matched by the **{word}** rule, not by the . rule. Understanding and using this principle will make your lexers clearer and more bug free.

The third and final section of the lex specification is the user subroutines section. Once again, it is separated from the previous section by "%%". The user subroutines section can contain any valid C code. It is copied verbatim into the generated lexer. Typically this section contains support routines. For this example our "support" code is the main routine:

```
%%
main()
{
        yylex();
        printf("%d %d %d\n",lineCount, wordCount, charCount);
}
```

It first calls the lexer's entry point **yylex()** and then calls **printf()** to print the

results of this run. Note that our sample doesn't do anything fancy; it doesn't accept command-line arguments, doesn't open any files, but uses the lex default to read from the standard input. We will stick with this for most of our sample programs as we assume you know how to build C programs which do such things. However, it is worthwhile to look at one way to reconnect lex's input stream, as shown in Example 2-2.

Example 2-2. User subroutines for word count program ch2-02.l

```
main(argc,argv)
int argc;
char **argv;
{
      if (argc > 1) {
            FILE *file;

            file = fopen(argv[1], "r");
```

```
            if (!file) {
                fprintf(stderr,"could not open %s\n",argv[1]);
                exit(1);
            }
            yyin = file;
        }
        yylex();
        printf("%u %u %u\n",charCount, wordCount, lineCount);
        return 0;
}
```

This example assumes that the second argument the program is called with is

the file to open for processing.[6] A lex lexer reads its input from the standard I/O file **yyin**, so you need only change **yyin** as needed. The default value of **yyin** is **stdin**, since the default input source is standard input.

We stored this example in ch2-02.l, since it's the second example in Chapter 2, and lex source files traditionally end with *.l*. We ran it on itself, and obtained the following results:

```
% ch2-02 ch2-02.l
467 72 30
```

One big difference between our word count example and the standard UNIX word count program is that ours handles only a single file. We'll fix this by using lex's end-of-file processing handler.

When **yylex()** reaches the end of its input file, it calls **yywrap()**, which returns a value of 0 or 1. If the value is 1, the program is done and there is no more input. If the value is 0, on the other hand, the lexer assumes that **yywrap()** has opened another file for it to read, and continues to read from **yyin**. The default **yywrap()** always returns 1. By providing our own version of **yywrap()**, we can have our program read all of the files named on the command line, one at a time.

Handling multiple files requires considerable code changes. Example 2-3 shows our final word counter in its entirety.

Example 2-3. Multi-file word count program ch2-03.l

```
%{
/*
 * ch2-03.l
 *
 * The word counter example for multiple files
 *
 */
```

```
unsigned long charCount = 0, wordCount = 0, lineCount = 0;

#undef yywrap     /* sometimes a macro by default */


%}

word [^ \t\n]+
eol \n
%%
{word}      { wordCount++; charCount += yyleng; }
{eol} { charCount++; lineCount++; }
.     charCount++;
%%

char **fileList;
unsigned currentFile = 0;
unsigned nFiles;
unsigned long totalCC = 0;
unsigned long totalWC = 0;
unsigned long totalLC = 0;

main(argc,argv)
int argc;
char **argv;
{
        FILE *file;

        fileList = argv+1;
        nFiles = argc-1;

        if (argc == 2) {
            /*
             * we handle the single file case differently from
             * the multiple file case since we don't need to
             * print a summary line
             */
            currentFile = 1;
            file = fopen(argv[1], "r");
            if (!file) {
                  fprintf(stderr,"could not open %s\n",argv[1])
                  exit(1);
            }
            yyin = file;
            }
            if (argc > 2)
                yywrap();  /* open first file */

          yylex();
            /*
             * once again, we handle zero or one file
            * differently from multiple files.
```

```
             */
            if (argc > 2) {
                printf("%8lu %8lu %8lu %s\n", lineCount,
wordCount,
                      charCount, fileList[currentFile-1]);
                totalCC += charCount;
                totalWC += wordCount;
                totalLC += lineCount;
                printf("%8lu %8lu %8lu total\n",totalLC, totalWC,
totalCC);
            } else
                printf("%8lu %8lu %8lu\n",lineCount, wordCount,
charCount);

            return 0;
}

/*
 * the lexer calls yywrap to handle EOF conditions (e.g., to
 * connect to a new file, as we do in this case.)
 */

yywrap()
{
      FILE *file = NULL;

      if ((currentFile != 0) && (nFiles > 1) && (currentFile <
nFiles)) {
             /*
              * we print out the statistics for the previous file.
              */
             printf("%8lu %8lu %8lu %s\n", lineCount, wordCount,
                   charCount, fileList[currentFile-1]);
             totalCC += charCount;
             totalWC += wordCount;
             totalLC += lineCount;
             charCount = wordCount = lineCount = 0;
             fclose (yyin);    /* done with that file */
      }

      while (fileList[currentFile] != (char *)0) {
            file = fopen(fileList[currentFile++], "r");
            if (file != NULL) {
                  yyin = file;
                  break;
            }
            fprintf(stderr,
                  "could not open %s\n",
                  fileList[currentFile-1]);
      }
      return (file ? 0 : 1);  /* 0 means there's more input */
}
```

Our example uses **yywrap()** to perform the continuation processing. There are other possible ways, but this is the simplest and most portable. Each time the lexer calls **yywrap()** we try to open the next filename from the command line and assign the open file to **yyin**, returning 0 if there was another file and 1 if not.

Our example reports both the sizes for the individual files as well as a cumulative total for the entire set of files at the end; if there is only one file the numbers for the specified file are reported once.

We ran our final word counter on both the lex file, *ch2-03.l*, then on both the lex file and the generated C file *ch2-03.c*.

```
% ch2-03.pgm ch2-03.l
     107     337    2220
% ch2-03.pgm ch2-03.l ch2-03.c
     107     337    2220 ch2-03.l
     405    1382    9356 ch2-03.c
     512    1719   11576 total
```

The results will vary from system to system, since different versions of lex produce different C code. We didn't devote much time to beautifying the output; that is left as an exercise for the reader.

Parsing a Command Line

Now we turn our attention to another example using lex to parse command input. Normally a lex program reads from a file, using the predefined macro **input()**, which gets the next character from the input, and **unput()**, which puts a character back in the logical input stream. Lexers sometimes need to use **unput()** to peek ahead in the input stream. For example, a lexer can't tell that it's found the end of a word until it sees the punctuation after the end of the word, but since the punctuation isn't part of the word, it has to put the punctuation back in the input stream for the next token.

In order to scan the command line rather than a file, we must rewrite **input()** and **unput()**. The implementation we use here only works in AT&T lex, because other versions for efficiency reasons don't let you redefine the two routines. (Flex, for example, reads directly from the input buffer and never uses **input()**.) If you are using another version of lex, see the section "Input from Strings" in Chapter 6 to see how to accomplish the same thing.

We will take the command-line arguments our program is called with, and recognize three distinct classes of argument: **help, verbose**, and

a **filename**. Example 2-4 creates a lexer that reads the standard input, much as we did for our earlier word count example.

Example 2-4. Lex specification to parse command-line input ch2-04.l

```
%{
unsigned verbose;
char *progName;
%}

%%

-h     |
"-?"   |
-help { printf("usage is: %s [-help | -h | -? ] [-verbose | -v] "
        "[(-file| -f) filename]\n", progName);
      }
-v     |
-verbose { printf("verbose mode is on\n"); verbose = 1; }

%%

main(argc, argv)
int argc;
char **argv;
{
      progName = *argv;
      yylex();
}
```

The definition section includes a code literal block. The two variables, **verbose** and **progName**, are variables used later within the rules section.

In our rules section the first rules recognize the keyword *-help* as well as abbreviated versions *-h* and *-?*. Note the action following this rule which simply prints a usage string.[7] Our second set of rules recognize the keyword *-verbose* and the short variant *-v*. In this case we set the global variable **verbose**, which we defined above, to the value 1.

In our user subroutines section the **main()** routine stores the program name, which is used in our help command's usage string, and then calls **yylex()**.

This example does not parse the command-line arguments, as the lexer is still reading from the standard input, not from the command line. Example 2-5 adds code to replace the standard **input()** and **unput()** routines with our own. (This example is specific to AT&T lex. See Appendix E for the equivalent in flex.)

Example 2-5. Lex specification to parse a command line ch2-05.l

```
%{
#undef input
#undef unput
int input(void);
void unput(int ch);
unsigned verbose;
char *progName;
%}

%%

-h    |
"-?"  |
-help { printf("usage is: %s [-help | -h | -? ] [-verbose | -v]"
       " [(-file| -f) filename]\n", progName);
      }
-v    |
-verbose { printf("verbose mode is on\n"); verbose = 1; }

%%
char **targv;     /* remembers arguments */
char **arglim;    /* end of arguments */

main(int argc, char **argv)
{
      progName = *argv;
      targv = argv+1;
      arglim = argv+argc;
      yylex();
}

static unsigned offset = 0;

int
input(void)
{
      char c;

      if (targv >= arglim)
            return(0);  /* EOF */
      /* end of argument, move to the next */
      if  ((c = targv[0][offset++])   !=  '\0')
             return(c);
      targv++;
      offset = 0;
      return(' ');
}

/* simple unput only backs up, doesn't allow you to */
/* put back different text */
void
```

```
unput(int ch)
{

      /* AT&T lex sometimes puts back the EOF ! */
      if(ch == 0)
            return;     /* ignore, can't put back EOF */
      if (offset) {     /* back up in current arg */
            offset--;
            return;
      }

      targv--;     /* back to previous arg */
      offset = strlen(*targv);
}
```

In the definition section we **#undef** both **input** and **unput** since AT&T lex by default defines them as macros, and we redefine them as C functions.

Our rules section didn't change in this example. Instead, most of the changes are in the user subroutines section. In this new section we've added three variables—**targv**, which tracks the current argument, **arglim**, which marks the end of the arguments, and **offset**, which tracks the position in the current argument. These are set in **main()** to point at the argument vector passed from the command line.

The **input()** routine handles calls from the lexer to obtain characters. When the current argument is exhausted it moves to the next argument, if there is one, and continues scanning. If there are no more arguments, we treat it as the lexer's end-of-file condition and return a zero byte.

The **unput()** routine handles calls from the lexer to "push back" characters into the input stream. It does this by reversing the pointer's direction, moving backwards in the string. In this case we assume that the characters pushed back are the same as the ones that were there in the first place, which will always be true unless action code explicitly pushes back something else. In the general case, an action routine can push back anything it wants and a private version of **unput()** must be able to handle that.

Our resulting example still echoes input it doesn't recognize and prints out the two messages for the inputs it does understand. For instance, here is a sample run:

```
% ch2-05 -verbose foo
verbose mode is on
 foo %
```

Our input now comes from the command line and unrecognized input is echoed. Any text which is not recognized by the lexer "falls through" to the default rule, which echoes the unrecognized text to the output.

Start States

Finally, we add a *-file* switch and recognize a filename. To do this we use a *start state*, a method of capturing context sensitive information within the lexer. Tagging rules with start states tells the lexer only to recognize the rules when the start state is in effect. In this case, to recognize a filename after a *-file* argument, we use a start state to note that it's time to look for the filename, as shown in Example 2-6.

Example 2-6. Lex command scanner with filenames ch2-06.l

```
%{
#undef input
#undef unput
unsigned verbose;
unsigned fname;
char *progName;
%}

%s FNAME

%%
[ ]+          /* ignore blanks */ ;


-h    |
"-?"  |
-help { printf("usage is: %s [-help | -h | -? ] [-verbose | -v]"
       " (-file | -f) filename\n", progName);
      }
-v    |
-verbose { printf("verbose mode is on\n"); verbose = 1; }
-f    |
-file { BEGIN FNAME; fname = 1; }

<FNAME>[^ ]+ { printf("use file %s\n", yytext); BEGIN 0; fname =
2;}

[^ ] +  ECHO;
%%
char **targv;      /* remembers arguments */
char **arglim;     /* end of arguments */

main(int argc, char **argv)
{
      progName = *argv;
```

```
        targv = argv+1;
        arglim = argv+argc;
        yylex();
        if(fname < 2)
              printf("No filename given\n");
}
```

... same input() and unput() as Example 2-5 ...

In the definition section we have added the line "%s FNAME" which creates a new start state in the lexer. In the rules section we have added rules which begin with "<FNAME>". These rules are only recognized when the lexer is in state FNAME. Any rule which does not have an explicit state will match no matter what the current start state is. The *-file* argument switches to FNAME state, which enables the pattern that matches the filename. Once it's matched the filename, it switches back to the regular state.

Code within the actions of the rules section change the current state. You enter a new state with a BEGIN statement. For instance, to change to the FNAME state we used the statement "BEGIN FNAME;". To change back to the default state, we use "BEGIN 0". (The default, state zero, is also known as INITIAL.)

In addition to changing the lex state we also added a separate variable, **fname**, so that our example program can recognize if the argument is missing; note that the main routine prints an error message if **fname's** value hasn't been changed to 2.

Other changes to this example simply handle the filename argument. Our version of **input()** returns a blank space after each command-line argument. The rules ignore whitespace, yet without that blank space, the arguments *-file* and *file* would appear identical to the lexer.

We mentioned that a rule without an explicit start state will match regardless of what start state is active (Example 2-7).

Example 2-7. Start state example ch2-07.l

```
%s MAGIC

%%
<MAGIC>.+    { BEGIN 0; printf("Magic:"); ECHO; }
magic        BEGIN MAGIC;
%%

main()
{
        yylex();
```

```
}
```

We switch into state MAGIC when we see the keyword “magic.” Otherwise we simply echo the input. If we are in state MAGIC, we prepend the string “Magic:” to the next token echoed. We created an input file with three words in it: “magic,” “two,” and “three,” and ran it through this lexer.

```
% ch2-07 < magic.input


Magic:two
Three
```

Now, we change the example slightly, so that the rule with the start state follows the one without, as shown in Example 2-8.

Example 2-8. Broken start state example ch2-08.l

```
%{
     /* This example deliberately doesn't work! */
%}

%s MAGIC

%%
magic        BEGIN MAGIC;
.+           ECHO;
<MAGIC>.+    { BEGIN 0; printf ("Magic:"); ECHO; }
%%

main()
{
     yylex();
}
```

With the same input we get very different results:

```
% ch2-08 < magic.input


two
three
```

Think of rules without a start state as implicitly having a “wild card” start state—they match all start states. This is a frequent source of bugs. Flex and

other more recent versions of lex have “exclusive start states” which fix the wild card problem. See "Start States" in Chapter 6 for details.

A C Source Code Analyzer

Our final example examines a C source file and counts the number of different types of lines we see that contain code, that just contain comments, or are blank. This is a little tricky to do since a single line can contain both comments and code, so we have to decide how to count such lines.

First, we describe a line of whitespace. We will consider any line with nothing more than a newline as whitespace. Similarly, a line with any number of blank spaces or tabs, but nothing else, is whitespace. The regular expression describing this is:

```
^[ \t]*\n
```

The “^” operator denotes that the pattern must start at the beginning of a line. Similarly, we require that the entire line be only whitespace by requiring a newline, “\n”, at the end.

Now, we can complement this with the description of what a line of code or comments is—any line which isn’t entirely whitespace!

```
^[ \t]*\n
\n    /* whitespace lines matched by previous rule */
.     /* anything else */
```

We use the new rule “\n” to count the number of lines we see which aren’t all whitespace. The second new rule we use to discard characters in which we aren’t interested. Here is the rule we add to describe a comment:

```
^[ \t]*"/*".*"*/"[ \t]*\n
```

This describes a single, self contained comment on a single line, with optional text between the “/*” and the “*/”. Since “*” and “/” are both special pattern characters, we have to quote them when they occur literally. Actually this pattern isn’t quite right, since something like this:

```
/* comment */ /* comment
```

won’t match it. Comments might span multiple lines and the “.” operator excludes the “\n” character. Indeed, if we were to allow the “\n” character we would probably overflow the internal lex buffer on a long comment. Instead, we

circumvent the problem by adding a start state, COMMENT, and by entering that state when we see only the beginning of a comment. When we see the end of a comment we return to the default start state. We don't need to use our start state for one-line comments. Here is our rule for recognizing the beginning of a comment:

```
^[ \t]*"/*"
```

Our action has a BEGIN statement in it to switch to the COMMENT state. It is important to note that we are requiring that a comment begin on a line by itself. Not doing so would incorrectly count cases such as:

```
int counter; /* this is
      a strange comment */
```

because the first line isn't on a line alone. We need to count the first line as a line of code and the second line as a line of comment. Here are our rules to accomplish this:

```
.+"/*".*"*/".*\n
.*"/*".*"*/".+\n
```

The two expressions describe an overlapping set of strings, but they are not identical. The following expression matches the first rule, but not the second:

```
int counter; /* comment */
```

because the second requires there be text following the comment. Similarly, this next expression matches the second but not the first:

```
/* comment */ int counter;
```

They both would match the expression:

```
/* comment # 1 */ int counter; /* comment # 2 */
```

Finally, we need to finish up our regular expressions for detecting comments. We decided to use a start state, so while we are in the COMMENT state, we merely look for newlines:

```
<COMMENT>\n
```

and count them. When we detect the "end of comment" character, we either count it as a comment line, if there is nothing else on the line after the comment ends, or we continue processing:

```
<COMMENT>"*/"[ \t]*\n
<COMMENT>"*/"
```

The first one will be counted as a comment line; the second will continue

processing. As we put these rules together, there is a bit of gluing to do because we need to cover some cases in both the default start state and in the COMMENT start state. Example 2-9 shows our final list of regular expressions, along with their associated actions.

Example 2-9. C source analyzer ch2-09.l

```
%{
int comments, code, whiteSpace;
%}

%x COMMENT
%%
^[ \t]*"/*" { BEGIN COMMENT; /* enter comment eating state */ }
^[ \t]*"/*".*"*/"[ \t]*\n {
              comments++;     /* self-contained comment */
              }

<COMMENT>"*/"[ \t]*\n  { BEGIN 0; comments++;}
<COMMENT>"*/"          { BEGIN 0; }
<COMMENT>\n { comments++; }
<COMMENT>.\n    { comments++; }

^[ \t]*\n   { whiteSpace++; }

.+"/*".*"*/".*\n  { code++; }
.*"/*".*"*/".+\n  { code++ }
.+"/*".*\n        { code++; BEGIN COMMENT; }
.\n               { code++; }

.                 ; /* ignore everything else */
%%
main()
{
     yylex();
     printf("code: %d, comments %d, whitespace %d\n",
           code, comments, whiteSpace);
}
```

We added the rules “<COMMENT>\n” and “<COMMENT>.\n” to handle the case of a blank line in a comment, as well as any text within a comment. Forcing them to match an end-of-line character means they won’t match something like

```
/* this is the beginning of a comment
```

```
and this is the end */ int counter;
```

as two lines of comments. Instead, this will count as one line of comment and one line of code.

Summary

In this chapter we covered the fundamentals of using lex. Even by itself, lex is often sufficient for writing simpler applications such as the word count and lines-of-code count utilities we developed in this chapter.

Lex uses a number of special characters to describe regular expressions. When a regular expression matches an input string, it executes the corresponding action, which is a piece of C code you specify. Lex matches these expressions first by determining the longest matching expression, and then, if two matches are the same length, by matching the expression which appears first in the lex specification. By judiciously using start states, you can further refine when specific rules are active, just as we did for the line-of-code count utility.

We also discussed special purpose routines used by the lex-generated state machines, such as **yywrap()**, which handles end-of-file conditions and lets you handle multiple files in sequence. We used this to allow our word count example to examine multiple files.

This chapter focused upon using lex alone as a processing language. Later chapters will concentrate on how lex can be integrated with yacc to build other types of tools. But lex, by itself, is capable of handling many otherwise tedious tasks without needing a full-scale yacc parser.

Exercises

1. Make the word count program smarter about what a word is, distinguishing real words which are strings of letters (perhaps with a hyphen or apostrophe) from blocks of punctuation. You shouldn't need to add more than ten lines to the program.
2. Improve the C code analyzer: count braces, keywords, etc. Try to identify function definitions and declarations, which are names followed by "(" outside of any braces.
3. Is lex really as fast as we say? Race it against *egrep, awk, sed,* or other pattern matching programs you have. Write a lex specification that looks for lines containing some string and prints the lines out. (For a fair comparison, be sure to print the whole line.) Compare the time it takes to

scan a set of files to that taken by the other programs. If you have more than one version of lex, do they run at noticably different speeds?

Chapter 3. Using Yacc

The previous chapter concentrated on lex alone. In this chapter we turn our attention to yacc, although we use lex to generate our lexical analyzers. Where lex recognizes regular expressions, yacc recognizes entire grammars. Lex divides the input stream into pieces (tokens) and then yacc takes these pieces and groups them together logically.

In this chapter we create a desk calculator, starting with simple arithmetic, then adding built-in functions, user variables, and finally user-defined functions.

Grammars

Yacc takes a grammar that you specify and writes a parser that recognizes valid "sentences" in that grammar. We use the term "sentence" here in a fairly general way—for a C language grammar the sentences are syntactically valid C programs.[8]

As we saw in Chapter 1, a grammar is a series of *rules* that the parser uses to recognize syntactically valid input. For example, here is a version of the grammar we'll use later in this chapter to build a calculator.

```
statement → NAME = expression

expression → NUMBER + NUMBER | NUMBER - NUMBER
```

The vertical bar, "|", means there are two possibilities for the same symbol, i.e., an *expression* can be either an addition or a subtraction. The symbol to the left of the → is known as the *left-hand side* of the rule, often abbreviated LHS, and the symbols to the right are the *right-hand side*, usually abbreviated RHS. Several rules may have the same left-hand side; the vertical bar is just a short hand for this. Symbols that actually appear in the input and are returned by the lexer are *terminal* symbols or *tokens*, while those that appear on the left-hand side of some rule are *non-terminal* symbols or non-terminals. Terminal and non-terminal symbols must be different; it is an error to write a rule with a token on the left side.

The usual way to represent a parsed sentence is as a tree. For example, if we parsed the input "fred = 12 + 13" with this grammar, the tree would look like Figure 3-1. "12 + 13" is an *expression*, and "fred = *expression*" is a *statement*. A yacc parser doesn't actually create this tree as a data structure, although it is not hard to do so yourself.

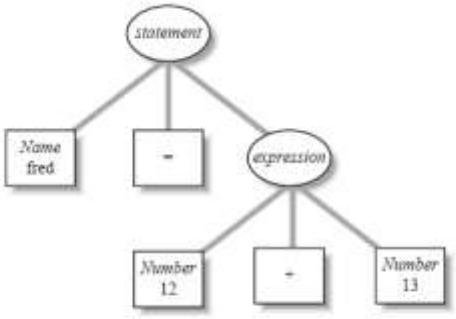


Figure 3-1. A parse tree

Every grammar includes a *start* symbol, the one that has to be at the root of the parse tree. In this grammar, *statement* is the start symbol.

Recursive Rules

Rules can refer directly or indirectly to themselves; this important ability makes it possible to parse arbitrarily long input sequences. Let's extend our grammar to handle longer arithmetic expressions:

```
expression → NUMBER
  | expression + NUMBER
  | expression - NUMBER
```

Now we can parse a sequence like "fred = 14 + 23 − 11 + 7" by applying the expression rules repeatedly, as in Figure 3-2. Yacc can parse recursive rules very efficiently, so we will see recursive rules in nearly every grammar we use.

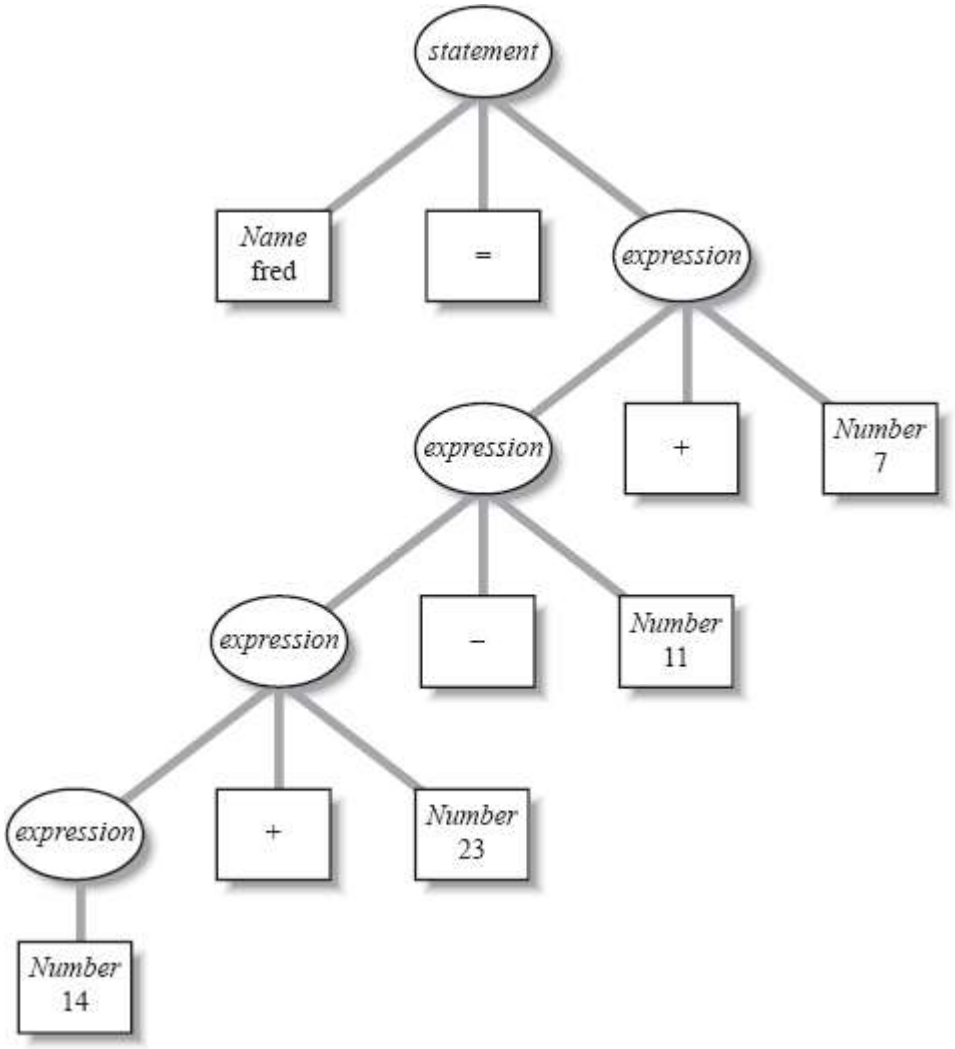
Shift/Reduce Parsing

A yacc parser works by looking for rules that might match the tokens seen so far. When yacc processes a parser, it creates a set of *states* each of which reflects a possible position in one or more partially parsed rules. As the parser reads tokens, each time it reads a token that doesn't complete a rule it pushes the token on an internal stack and switches to a new state reflecting the token it just read. This action is called a *shift*. When it has found all the symbols that constitute the right-hand side of a rule, it pops the right-hand side symbols off the stack, pushes the left-hand side symbol onto the stack, and switches to a new state reflecting the new symbol on the stack. This action is called a *reduction*, since it usually reduces the number of items on the stack. (Not always, since it is possible to have rules with empty right-hand sides.) Whenever yacc reduces a rule, it executes user code associated with the rule. This is how you actually do something with the material that the parser parses.

Let's look how it parses the input "fred = 12 + 13" using the simple rules in Figure 3-1. The parser starts by shifting tokens on to the internal stack one at a time:

Let's look how it parses the input "fred = 12 + 13" using the simple rules in Figure 3-1. The parser starts by shifting tokens on to the internal stack one at a time:

```
fred
fred =
fred = 12
fred = 12 +
fred = 12 + 13
```



At this point it can reduce the rule "expression → NUMBER + NUMBER" so it pops the 12, the plus, and the 13 from the stack and replaces them with *expression*:

```
fred = expression
```

Now it reduces the rule "statement → NAME = expression", so it pops fred, =, and *expression* and replaces them with *statement*. We've reached the end of the input and the stack has been reduced to the start symbol, so the input was valid according to the grammar.

What Yacc Cannot Parse

Although yacc's parsing technique is general, you can write grammars which yacc cannot handle. It cannot deal with ambiguous grammars, ones in which the same input can match more than one parse tree.[9] It also cannot deal with grammars that need more than one token of lookahead to tell whether it has matched a rule. Consider this extremely contrived example:

```
                phrase → cart_animal AND CART
                       | work_animal AND PLOW
cart_animal → HORSE | GOAT work_animal → HORSE | OX
```

This grammar isn't ambiguous, since there is only one possible parse tree for any valid input, but yacc can't handle it because it requires two symbols of lookahead. In particular, in the input "HORSE AND CART" it cannot tell whether HORSE is a **cart_animal** or a **work_animal** until it sees CART, and yacc cannot look that far ahead.

If we changed the first rule to this:

```
phrase → cart_animal CART
       | work_animal PLOW
```

yacc would have no trouble, since it can look one token ahead to see whether an input of HORSE is followed by CART, in which case the horse is a **cart_animal** or by PLOW in which case it is a **work_animal**.

In practice, these rules are not as complex and confusing as they may seem here. One reason is that yacc knows exactly what grammars it can parse and what it cannot. If you give it one that it cannot handle it will tell you, so there is no problem of overcomplex parsers silently failing. Another reason is that the grammars that yacc can handle correspond pretty well to ones that people really write. As often as not, a grammatical construct that confuses yacc will confuse people as well, so if you have some latitude in your language design you should consider changing the language to make it both more understandable to yacc and to its users.

For more information on shift/reduce parsing, see Chapter 8. For a discussion of what yacc has to do to turn your specification into a working C program, see the classic compiler text by Aho, Sethi, and Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, often known as the "dragon book" because of the cover illustration.

A Yacc Parser

A yacc grammar has the same three-part structure as a lex specification. (Lex copied its structure from yacc.) The first section, the definition section, handles control information for the yacc-generated parser (from here on we will call it the parser), and generally sets up the execution environment in which the parser will operate. The second section contains the rules for the parser, and the third section is C code copied verbatim into the generated C program.

We'll first write parser for the simplest grammar, the one in Figure 3-1, then extend it to be more useful and realistic.

The Definition Section

The definition section includes declarations of the tokens used in the grammar, the types of values used on the parser stack, and other odds and ends. It can also include a literal block, C code enclosed in %{ %} lines. We start our first parser by declaring two symbolic tokens.

```
%token NAME NUMBER
```

You can use single quoted characters as tokens without declaring them, so we don't need to declare "=", "+", or "–".

The Rules Section

The rules section simply consists of a list of grammar rules in much the same format as we used above. Since ASCII keyboards don't have a → key, we use a colon between the left- and right-hand sides of a rule, and we put a semicolon at the end of each rule:

```
%token NAME NUMBER
%%
statement: NAME '=' expression
        |    expression
        ;
expression: NUMBER '+' NUMBER
        |     NUMBER '-' NUMBER
        ;
```

Unlike lex, yacc pays no attention to line boundaries in the rules section, and you will find that a lot of whitespace makes grammars easier to read. We've added one new rule to the parser: a statement can be a plain expression as well as an assignment. If the user enters a plain expression, we'll print out its result.

The symbol on the left-hand side of the first rule in the grammar is normally the start symbol, though you can use a %**start** declaration in the definition section to override that.

Symbol Values and Actions

Every symbol in a yacc parser has a *value*. The value gives additional information about a particular instance of a symbol. If a symbol represents a number, the value would be the particular number. If it represents a literal text string, the value would probably be a pointer to a copy of the string. If it represents a variable in a program, the value would be a pointer to a symbol table entry describing the variable. Some tokens don't have a useful value, e.g., a token representing a close parenthesis, since one close parenthesis is the same as another.

Non-terminal symbols can have any values you want, created by code in the parser. Often the action code builds a parse tree corresponding to the input, so that later code can process a whole statement or even a whole program at a time.

In the current parser, the value of a *NUMBER* or an *expression* is the numerical value of the number or expression, and the value of a *NAME* will be a symbol table pointer.

In real parsers, the values of different symbols use different data types, e.g., *int* and *double* for numeric symbols, *char* * for strings, and pointers to structures for higher level symbols. If you have multiple value types, you have to list all the value types used in a parser so that yacc can create a C *union* typedef called *YYSTYPE* to contain them. (Fortunately, yacc gives you a lot of help ensuring that you use the right value type for each symbol.)

In the first version of the calculator, the only values of interest are the numerical values of input numbers and calculated expressions. By default yacc makes all values of type *int*, which is adequate for our first version of the calculator.

Whenever the parser reduces a rule, it executes user C code associated with the rule, known as the rule's *action*. The action appears in braces after the end of the rule, before the semicolon or vertical bar. The action code can refer to the values of the right-hand side symbols as **$1, $2**, ..., and can set the value of the left-hand side by setting **$$**. In our parser, the value of an *expression* symbol is the value of the expression it represents. We add some code to evaluate and print expressions, bringing our grammar up to that used in Figure 3-2.

```
%token NAME NUMBER

%%
```

```
statement:   NAME '='  expression
       |     expression          { printf("= %d\n", $1); }
       ;

expression:  expression '+' NUMBER { $$ = $1 + $3; }
       |     expression '-' NUMBER { $$ = $1 - $3; }
       |     NUMBER                { $$ = $1; }
       ;
```

The rules that build an expression compute the appropriate values, and the rule

that recognizes an expression as a statement prints out the result. In the expression building rules, the first and second numbers' values are **$1** and **$3**, respectively. The operator's value would be **$2**, although in this grammar the operators do not have interesting values. The action on the last rule is not strictly necessary, since the default action that yacc performs after every reduction, before running any explicit action code, assigns the value **$1** to **$$**.

The Lexer

To try out our parser, we need a lexer to feed it tokens. As we mentioned in Chapter 1, the parser is the higher level routine, and calls the lexer **yylex()** whenever it needs a token from the input. As soon as the lexer finds a token of interest to the parser, it returns to the parser, returning the token code as the value. Yacc defines the token names in the parser as C preprocessor names in *y.tab.h* (or some similar name on MS-DOS systems) so the lexer can use them.

Here is a simple lexer to provide tokens for our parser:

```
%{
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+      { yylval = atoi(yytext);  return NUMBER; }
[ \t] ;           /* ignore whitespace */
\n    return 0;   /* logical EOF */
```

```
.       return yytext[0];
%%
```

Strings of digits are numbers, whitespace is ignored, and a newline returns an end of input token (number zero) to tell the parser that there is no more to read. The last rule in the lexer is a very common catch-all, which says to return any character otherwise not handled as a single character token to the parser. Character tokens are usually punctuation such as parentheses, semicolons, and single-character operators. If the parser receives a token that it doesn't know about, it generates a syntax error, so this rule lets you handle all of the single-character tokens easily while letting yacc's error checking catch and complain about invalid input.

Whenever the lexer returns a token to the parser, if the token has an associated value, the lexer must store the value in **yylval** before returning. In this first example, we explicitly declare **yylval**. In more complex parsers, yacc defines **yylval** as a *union* and puts the definition in *y.tab.h*.

We haven't defined **NAME** tokens yet, just **NUMBER** tokens, but that is OK for the moment.

Compiling and Running a Simple Parser

On a UNIX system, yacc takes your grammar and creates *y.tab.c*, the C language parser, and *y.tab.h*, the include file with the token number definitions. Lex creates *lex.yy.c*, the C language lexer. You need only compile them together with the yacc and lex libraries. The libraries contain usable default versions of all of the supporting routines, including a **main()** that calls the parser **yyparse()** and exits.

```
% yacc -d ch3-01.y    # makes y.tab.c and "y.tab.h
% lex ch3-01.l        # makes lex.yy.c
% cc -o ch3-01 y.tab.c lex.yy.c -ly -ll     # compile and
link C files
% ch3-01
99 + 12
= 111
% ch3-01
2 + 3-14+33
= 24
% ch3-01
100 + -50
syntax error
```

Our first version seems to work. In the third test, it correctly reports a syntax error when we enter something that doesn't conform to the grammar.

Arithmetic Expressions and Ambiguity

Let's make the arithmetic expressions more general and realistic, extending

the *expression* rules to handle multiplication and division, unary negation, and parenthesized expressions:

```
expression: expression '+' expression { $$ = $1 + $3; }
      |     expression '-' expression { $$ = $1 - $3; }
      |     expression '*' expression { $$ = $1 * $3; }
      |     expression '/' expression
                  {     if($3 == 0)
                              yyerror("divide by zero");
                        else
                              $$ = $1 / $3;
                  }
      |     '-' expression          { $$ = -$2; }
      |     '(' expression ')'      { $$ = $2; }
      |     NUMBER                  { $$ = $1; }
      ;
```

The action for division checks for division by zero, since in many implementations of C a zero divide will crash the program. It calls **yyerror()**, the standard yacc error routine, to report the error.

But this grammar has a problem: it is extremely ambiguous. For example, the input 2+3*4 might mean (2+3)*4 or 2+(3*4), and the input 3−4−5−6 might mean 3−(4−(5−6)) or (3−4)−(5−6) or any of a lot of other possibilities. Figure 3-3 shows the two possible parses for 2+3*4.

If you compile this grammar as it stands, yacc will tell you that there are 16 shift/reduce conflicts, states where it cannot tell whether it should shift the token on the stack or reduce a rule first.

For example, when parsing "2+3*4", the parser goes through these steps (we abbreviate *expression* as *E* here):

```
2                                   shift NUMBER

E                                   reduce E → NUMBER
E +                                 shift +
E + 3                               shift NUMBER
```

```
E + E                                    reduce E → NUMBER
```

At this point, the parser looks at the "*", and could either reduce "2+3" using:

```
    expression:                          expression '+'
expression
```

to an *expression*, or shift the "*" expecting to be able to reduce:

```
    expression:                          expression '*'
expression
```

later on.

The problem is that we haven't told yacc about the precedence and associativity of the operators. *Precedence* controls which operators to execute first in an expression. Mathematical and programming tradition (dating back past the first Fortran compiler in 1956) says that multiplication and division take precedence over addition and subtraction, so a+b*c means a+(b*c) and d/e−f means (d/e)−f. In any expression grammar, operators are grouped into levels of precedence from lowest to highest. The total number of levels depends on the language. The C language is notorious for having too many precedence levels, a total of fifteen levels.

Associativity controls the grouping of operators at the same precedence level. Operators may group to the left, e.g., a−b−c in C means (a−b)−c, or to the right, e.g., a=b=c in C means a=(b=c). In some cases operators do not group at all, e.g., in Fortran A.LE.B.LE.C is invalid.

There are two ways to specify precedence and associativity in a grammar, implicitly and explicitly. To specify them *implicitly*, rewrite the grammar using separate non-terminal symbols for each precedence level. Assuming the usual precedence and left associativity for everything, we could rewrite our expression rules this way:

```
expression: expression '+' mulexp
      |     expression '-' mulexp
      |     mulexp
      ;

mulexp:           mulexp '*' primary
      |     mulexp '/' primary
```

```
        |       primary
        ;

primary:    '(' expression ')'
        |       '-' primary
        |       NUMBER
        ;
```

This is a perfectly reasonable way to write a grammar, and if yacc didn't have explicit precedence rules, it would be the only way.

But yacc also lets you specify precedences *explicitly*. We can add these lines to the definition section, resulting in the grammar in Example 3-1.

```
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
```

Each of these declarations defines a level of precedence. They tell yacc that "+"

and "–" are left associative and at the lowest precedence level, "*" and "/" are left associative and at a higher precedence level, and **UMINUS**, a pseudo-token standing for unary minus, has no associativity and is at the highest precedence. (We don't have any right associative operators here, but if we did they'd use %**right**.) Yacc assigns each rule the precedence of the rightmost token on the right-hand side; if the rule contains no tokens with precedence assigned, the rule has no precedence of its own. When yacc encounters a shift/reduce conflict due to an ambiguous grammar, it consults the table of precedences, and if all of the rules involved in the conflict include a token which appears in a precedence declaration, it uses precedence to resolve the conflict.

In our grammar, all of the conflicts occur in the rules of the form *expression OPERATOR expression*, so setting precedences for the four operators allows it to resolve all of the conflicts. This parser using precedences is slightly smaller and faster than the one with the extra rules for implicit precedence, since it has fewer rules to reduce.

Example 3-1. The calculator grammar with expressions and precedence ch3-02.y

```
%token NAME NUMBER
%left '-' '+'
%left '*' '/'
```

```
%nonassoc UMINUS

%%

statement:  NAME '=' expression
      |     expression       { printf("= %d\n", $1); }
      ;

expression: expression '+' expression { $$ = $1 + $3; }
      |     expression '-' expression { $$ = $1 - $3; }
      |     expression '*' expression { $$ = $1 * $3; }
      |     expression '/' expression
                           {    if($3 == 0)
                                      yyerror("divide by zero");
                                else
                                      $$ = $1 / $3;
                           }

      |     '-' expression %prec UMINUS { $$ = -$2; }
      |     '(' expression ')'     { $$ = $2; }
      |     NUMBER                 { $$ = $1; }
      ;
%%
```

The rule for negation includes "%prec UMINUS". The only operator this rule includes is "–", which has low precedence, but we want unary minus to have higher precedence than multiplication rather than lower. The %**prec** tells yacc to use the precedence of **UMINUS** for this rule.

When Not to Use Precedence Rules

You can use precedence rules to fix any shift/reduce conflict that occurs in the grammar. This is usually a terrible idea. In expression grammars the cause of the conflicts is easy to understand, and the effect of the precedence rules is clear. In other situations precedence rules fix shift/reduce problems, but it is usually difficult to understand just what effect they have on the grammar.

We recommend that you use precedence in only two situations: in expression grammars, and to resolve the "dangling else" conflict in grammars for if-then-else language constructs. (See Chapter 7 for examples of the latter.) Otherwise, if you can, you should fix the grammar to remove the conflict. Remember that conflicts mean that yacc can't properly parse a grammar, probably because it's ambiguous, which means there are multiple possible parses for the same input. Except in the two cases above, this usually points to a mistake in your language design. If a grammar is ambiguous to yacc, it's almost certainly ambiguous to humans, too. See Chapter 8 for more information on finding and repairing conflicts.

Variables and Typed Tokens

Next we extend our calculator to handle variables with single letter names. Since there are only 26 single letters (lowercase only for the moment) we can simply store the variables in a 26 entry array, which we call **vbltable**. To make the calculator more useful, we also extend it to handle multiple expressions, one per line, and to use floating point values, as shown in Examples 3-2 and 3-3.

Example 3-2. Calculator grammar with variables and real values ch3-03.y

```
%{
double vbltable[26];
%{

%union {
      double dval;
      int vblno;
}

%token <vblno> NAME
%token <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%type <dval> expression
%%
statement_list:  statement '\n'
      |    statement_list statement '\n'
      ;

statement: NAME '=' expression   { vbltable[$1] = $3; }
      |    expression       { printf("= %g\n", $1); }
      ;

expression: expression '+' expression { $$ = $1 + $3; }
      |     expression '-' expression { $$ = $1 - $3; }
      |     expression '*' expression { $$ = $1 * $3; }
      |     expression '/' expression
                        {    if($3 == 0.0)
                                    yyerror("divide by zero");
                             else
                                    $$ = $1 / $3;
                         }
      |     '-' expression %prec UMINUS { $$ = -$2; }
      |     '(' expression ')'     { $$ = $2; }
      |     NUMBER
      |     NAME              { $$ = vbltable[$1]; }
      ;
%%
```

Example 3-3. Lexer for calculator with variables and real values ch3-03.l

```
%{
#include "y.tab.h"
#include <math.h>
extern double vbltable[26];
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
      yylval.dval = atof(yytext); return NUMBER;
      }

[ \t] ;             /* ignore whitespace */

[a-z] { yylval.vblno = yytext[0] - 'a'; return NAME; }

"$"   { return 0; /* end of input */ }

\n    |
.     return yytext[0];
%%
```

Symbol Values and %union

We now have multiple types of symbol values. Expressions have *double* values, while the value for variable references and **NAME** symbols are integers from 0 to 25 corresponding to the slot in **vbltable**. Why not have the lexer return the value of the variable as a *double*, to make the parser simpler? The problem is that there are two contexts where a variable name can occur: as part of an expression, in which case we want the *double* value, and to the left of an equal sign, in which case we need to remember which variable it is so we can update **vbltable**.

To define the possible symbol types, in the definition section we add a %**union** declaration:

```
%union {
      double dval;
      int vblno;
}
```

The contents of the declaration are copied verbatim to the output file as the contents of a C **union** declaration defining the type **YYSTYPE** as a C typedef. The generated header file *y.tab.h* includes a copy of the definition so that you can use it in the lexer. Here is the *y.tab.h* generated from this grammar:

```
#define NAME 257
#define NUMBER 258
#define UMINUS 259
typedef union {
      double dval;
      int vblno;
} YYSTYPE;
extern YYSTYPE yylval;
```

The generated file also declares the variable **yylval**, and defines the token numbers for the symbolic tokens in the grammar.

Now we have to tell the parser which symbols use which type of value. We do that by putting the appropriate field name from the union in angle brackets in the lines in the definition section that defines the symbol:

```
%token <vblno> NAME
%token <dval> NUMBER

%type <dval> expression
```

The new declaration %**type** sets the type for non-terminals which otherwise

need no declaration. You can also put bracketed types in %**left, %right**, or %**nonassoc**. In action code, yacc automatically qualifies symbol value references with the appropriate field name, e.g., if the third symbol is a **NUMBER**, a reference to **$3** acts like **$3.dval**.

The new, expanded parser was shown in Example 3-2. We've added a new start symbol **statement_list** so that the parser can accept a list of statements, each ended by a newline, rather than just one statement. We've also added an action for the rule that sets a variable, and a new rule at the end that turns a **NAME** into an **expression** by fetching the value of the variable.

We have to modify the lexer a little (Example 3-3). The literal block in the lexer no longer declares **yylval**, since its declaration is now in *y.tab.h*. The lexer doesn't have any automatic way to associate types with tokens, so you have to put in explicit field references when you set **yylval**. We've used the real number pattern from Chapter 2 to match floating point numbers. The action code uses **atof()** to read the number, then assigns the value to **yylval.dval**, since the parser expects the number's value in the **dval** field. For variables, we return the

index of the variable in the variable table in **yylval.vblno**. Finally, we've made "\n" a regular token, so we use a dollar sign to indicate the end of the input.

A little experimentation shows that our modified calculator works:

```
% ch3-03
2/3
= 0.666667
a = 2/7
a
= 0.285714
z = a+1
z
= 1.28571
a/z
= 0.222222
$
```

Symbol Tables

Few users will be satisfied with single character variable names, so now we add the ability to use longer variable names. This means we need a *symbol table*, a structure that keeps track of the names in use. Each time the lexer reads a name from the input, it looks the name up in the symbol table, and gets a pointer to the corresponding symbol table entry. Elsewhere in the program, we use symbol table pointers rather than name strings, since pointers are much easier and faster to use than looking up a name each time we need it.

Since the symbol table requires a data structure shared between the lexer and parser, we created a header file *ch3hdr.h* (see Example 3-4). This symbol table is an array of structures each containing the name of the variable and its value. We also declare a routine **symlook()** which takes a name as a text string and returns a pointer to the appropriate symbol table entry, adding it if it is not already there.

Example 3-4. Header for parser with symbol table ch3hdr.h

```
#define NSYMS 20  /* maximum number of symbols */

struct symtab {
      char *name;
      double value;
} symtab[NSYMS];

struct symtab *symlook();
```

The parser changes only slightly to use the symbol table, as shown in Example 3-5. The value for a **NAME** token is now a pointer into the symbol table rather than an index as before. We change the %**union** and call the pointer field **symp**. The %**token** declaration for **NAME** changes appropriately, and the actions that assign to and read variables now use the token value as a pointer so they can read or write the value field of the symbol table entry.

The new routine **symlook()** is defined in the user subroutines section of the yacc specification, as shown in Example 3-6. (There is no compelling reason for this; it could as easily have been in the lex file or in a file by itself.) It searches through the symbol table sequentially to find the entry corresponding to the name passed as its argument. If an entry has a **name** string and it matches the one that **symlook()** is searching for, it returns a pointer to the entry, since the name has already been put into the table. If the **name** field is empty, we've looked at all of the table entries that are in use, and haven't found this symbol, so we enter the name into the heretofore empty table entry.

We use **strdup()** to make a permanent copy of the name string. When the lexer calls **symlook()**, it passes the name in the token buffer **yytext**. Since each subsequent token overwrites **yytext**, we need to make a copy ourselves here. (This is a common source of errors in lex scanners; if you need to use the contents of **yytext** after the scanner goes on to the next token, always make a copy.) Finally, if the current table entry is in use but doesn't match, **symlook()** goes on to search the next entry.

This symbol table routine is perfectly adequate for this simple example, but more realistic symbol table code is somewhat more complex. Sequential search is too slow for symbol tables of appreciable size, so use hashing or some other faster search function. Real symbol tables tend to carry considerably more information per entry, e.g., the type of a variable, whether it is a simple variable, an array or structure, and how many dimensions if it is an array.

Example 3-5. Rules for parser with symbol table ch3-04.y

```
%{
#include "ch3hdr.h"
#include <string.h>
%}

%union {
      double dval;
      struct symtab *symp;
}
%token <symp> NAME
%token <dval> NUMBER
%left '-' '+'
```

```
%left '*' '/'
%nonassoc UMINUS

%type <dval> expression
%%
statement_list:  statement '\n'
      |    statement_list statement '\n'
      ;

statement: NAME '=' expression   { $1->value = $3; }
      |    expression        { printf("= %g\n", $1); }
      ;

expression: expression '+' expression { $$ = $1 + $3; }
      |     expression '-' expression { $$ = $1 - $3; }
      |     expression '*' expression { $$ = $1 * $3; }
      |     expression '/' expression
                        {    if($3 == 0.0)
                                   yyerror("divide by zero"
                             else
                                   $$ = $1 / $3;
                        }
      |     '-' expression %prec UMINUS  { $$ = -$2; }
      |     '(' expression ')'      { $$ = $2; }
      |     NUMBER
      |     NAME              { $$ = $1->value; }
      ;
%%
```

Example 3-6. Symbol table routine ch3-04.pgm

```
/* look up a symbol table entry, add if not present */
struct symtab *
symlook(s)
char *s;
{
      char *p;
      struct symtab *sp;
      for(sp = symtab;  sp < &symtab[NSYMS]; sp++) {
            /* is it already here? */
            if(sp->name && !strcmp(sp->name, s))
                  return sp;

            /* is it free */
            if(!sp->name) {
                   sp->name = strdup(s);
                   return sp;
            }
            /* otherwise continue to next */
      }
      yyerror("Too many symbols");
      exit(1);    /* cannot continue */
} /* symlook */
```

The lexer also changes only slightly to accommodate the symbol table (Example 3-7). Rather than declaring the symbol table directly, it now also includes *ch3hdr.h*. The rule that recognizes variable names now matches "[A-Za-z][A-Za-z0-9]*", any string of letters and digits starting with a letter. Its action calls **symlook()** to get a pointer to the symbol table entry, and stores that in **yylval.symp**, the token's value.

Example 3-7. Lexer with symbol table ch3-04.l

```
%{
#include "y.tab.h"
#include "ch3hdr.h"
#include <math.h>
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
            yylval.dval = atof(yytext);
            return NUMBER;
      }
[ \t] ;             /* ignore whitespace */

[A-Za-z][A-Za-z0-9]*    {     /* return symbol pointer */
            yylval.symp = symlook(yytext);
            return NAME;
      }

"$"   { return 0; }
\n    |
.     return yytext[0];
%%
```

There is one minor way in which our symbol table routine is better than those in most programming languages: since we allocate string space dynamically, there is no fixed limit on the length of variable names:[10]

```
      % ch3-04
      foo = 12
      foo /5
      = 2.4
      thisisanextremelylongvariablenamewhichnobodywouldwanttotype
  = 42
      3 *
  thisisanextremelylongvariablenamewhichnobodywouldwanttotype
      = 126
      $
      %
```

Functions and Reserved Words

The next addition we make to the calculator adds mathematical functions for square root, exponential, and logarithm. We want to handle input like this:

```
        s2 = sqrt(2)
s2
= 1.41421
s2*s2
= 2
```

The brute force approach makes the function names separate tokens, and adds separate rules for each function:

```
%token SQRT LOG EXP
 . . .
%%
expression: . . .
      |     SQRT '(' expression ')'{ $$ = sqrt($3); }
      |     LOG '(' expression ')' { $$ = log($3); }
      |     EXP '(' expression ')' { $$ = exp($3); }
```

In the scanner, we have to return a **SQRT** token for "sqrt" input and so forth:

```
sqrt    return SQRT;
log     return LOG;
exp     return EXP;

[A-Za-z][A-Za-z0-9]*    {     . . .
```

(The specific patterns come first so they match before than the general symbol pattern.)

This works, but it has problems. One is that you must hard-code every function into the parser and the lexer, which is tedious and makes it hard to add more functions. Another is that function names are *reserved words*, i.e., you cannot use **sqrt** as a variable name. This may or may not be a problem, depending on your intentions.

Reserved Words in the Symbol Table

First we'll take the specific patterns for function names out of the lexer and put them in the symbol table. We add a new field to each symbol table

entry: **funcptr**, a pointer to the C function to call if this entry is a function name.

```
struct symtab {
        char *name;
        double (*funcptr)();
        double value;
} symtab[NSYMS];
```

We have to put the functionnames in the symbol table before the parser starts, so we wrote our own **main()** which calls the new routine **addfunc()** to add each of the function names to the symbol table, thencalls **yyparse()**. The code for **addfunc()** merely gets the sybol table entry for a name and sets the **funcptr** field.

```
main()
{
        extern double sqrt(), exp(), log();

        addfunc("sqrt", sqrt);
        addfunc("exp", exp);
        addfunc("log", log);
        yyparse();
}

addfunc(name, func)
char *name;
double (*func)();
{
        struct symtab *sp = symlook(name);
        sp->funcptr = func;
}
```

We define a token **FUNC** to represent function names. The lexer will return **FUNC** when it sees a function name and **NAME** when it sees a variable name. The value for either is the symbol table pointer.

In the parser, we replace the separate rules for each function with one general function rule:

```
%token <symp> NAME FUNC
%%
expression: ...
        |     FUNC  '(' expression ')'{ $$ = ($1->funcptr)
($3); }
```

When the parser sees a function reference, it can consult the symbol table entry for the function to find the actual internal function reference.

In the lexer, we take out the patterns that matched the function names explicitly, and change the action code for names to return **FUNC** if the symbol table entry says that a name is a function name:

```
[A-Za-z][A-Za-z0-9]*   {
            struct symtab *sp = symlook(yytext);

            yylval.symp = sp;
            if (sp->funcptr)  /* is it a function? */
                  return FUNC;
            else
                  return NAME;
        }
```

These changes produce a program that works the same as the one above, but the function names are in the symbol table. The program can, for example, enter new function names as the parse progresses.

Interchangeable Function and Variable Names

A final change is technically minor, but changes the language significantly. There is no reason why function and variable names have to be disjoint! The parser can tell a function call from a variable reference by the syntax.

So we put the lexer back the way it was, always returning a **NAME** for any kind of name. Then we change the parser to accept a **NAME** in the function position:

```
%token <symp> NAME
%%
```

```
expression: ...
        |     NAME '(' expression ')' { ... }
```

The entire program is in Examples 3-8 through 3-11. As you can see in Example 3-9, we had to add some error checking to make sure that when the user calls a function, it's a real function.

Now the calculator operates as before, except that the names of functions and variables can overlap.

```
% ch3-05
sqrt(3)
= 1.73205
foo(3)
foo not a function
= 0
sqrt = 5
sqrt(sqrt)
= 2.23607
```

Whether you want to allow users to use the same name for two things in the same program is debatable. On the one hand it can make programs harder to understand, but on the other hand users are otherwise forced to invent names that do not conflict with the reserved names.

Either can be taken to extremes. COBOL has over 300 reserved words, so nobody can remember them all, and programmers resort to strange conventions like starting every variable name with a digit to be sure they don't conflict. On the other hand, PL/I has no reserved words at all, so you can write:

```
IF IF = THEN THEN ELSE = THEN; ELSE ELSE = IF;
```

Example 3-8. Final calculator header ch3hdr2.h

```
#define NSYMS 20   /* maximum number of symbols */

struct symtab {
      char *name;
      double (*funcptr)();
      double value;
} symtab [NSYMS];

struct symtab *symlook();
```

Example 3-9. Rules for final calculator parser ch3-05.y

```
%{
#include "ch3hdr2.h"
```

```
#include <string.h>
#include <math.h>
%}

%union {
      double dval;
      struct symtab *symp;
}
%token <symp> NAME
%token <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%type <dval> expression
%%
statement_list:  statement '\n'
      |     statement_list statement '\n'
      ;

statement:  NAME '=' expression    { $1->value = $3; }
      |     expression        { printf("= %g\n", $1); }
      ;

expression: expression '+' expression { $$ = $1 + $3; }
      |     expression '-' expression { $$ = $1 - $3; }
      |     expression '*' expression { $$ = $1 * $3; }
      |     expression '/' expression
                        {   if($3 == 0.0)
                                    yyerror("divide by zero");
                            else
                                    $$ = $1 / $3;
                        }
      |     '-' expression %prec UMINUS { $$ = -$2; }
      |     '(' expression ')'      { $$ = $2; }
      |     NUMBER
      |     NAME              { $$ = $1->value; }
      |     NAME '(' expression ')' {
                  if($1->funcptr)
                        $$ = ($1->funcptr)($3);
                  else {
                       printf("%s not a function\n", $1->name);
                       $$ = 0.0;
                  }
            }
      ;
%%
```

Example 3-10. User subroutines for final calculator parser ch3-05.y

```
/* look up a symbol table entry, add if not present */
struct symtab *
symlook(s)
```

```
char *s;
{
      char *p;
      struct symtab *sp;

      for(sp = symtab; sp < &symtab[NSYMS]; sp++) {
            /* is it already here? */
            if(sp->name &&  !strcmp(sp->name, s))
                  return sp;

            /* is it free */
            if(!sp->name) {
                  sp->name = strdup(s);
                  return sp;
            }
            /* otherwise continue to next */
      }
      yyerror("Too many symbols");
      exit(1);    /* cannot continue */
} /* symlook */

addfunc(name, func)
char *name;
double (*func)();
{
      struct symtab *sp = symlook(name);
      sp->funcptr = func;
}

main()
{
      extern double sqrt(), exp(), log();

      addfunc("sqrt", sqrt);
      addfunc("exp", exp);
      addfunc("log", log);
      yyparse();
}
```

Example 3-11. Final calculator lexer ch3-05.l

```
%{
#include "y.tab.h"
#include "ch3hdr2.h"
#include <math.h>
%}

%%

([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
           yylval.dval = atof(yytext);
           return NUMBER;
      }
```

```
[ \t];            /* ignore whitespace */

[A-Za-z][A-Za-z0-9]*   {     /* return symbol pointer */
            struct symtab *sp = symlook(yytext);
            yylval.symp = sp;
            return NAME;
      }

"$"   { return 0; }

\n    |
.     return yytext[0];
%%
```

Building Parsers with Make

About the third time you recompile this example, you will probably decide that some automation in the recompilation process is in order, using the UNIX *make* program. The *Makefile* that controls the rocess is shown in Example -12.

Example 3-12. Makefile for the calculator

```
#LEX = flex -I
#YACC = byacc

CC = cc -DYYDEBUG=1

ch3-05: y.tab.o lex.yy.o
      $(CC) -o ch3-05 y.tab.o lex.yy.o -ly -ll -lm

lex.yy.o:   lex.yy.c y.tab.h

lex.yy.o y.tab.o: ch3hdr2.h

y.tab.c y.tab.h: ch3-05.y
      $(YACC) -d ch3-05.y

lex.yy.c : ch3-05.l
      $(LEX) ch3lex.l
```

At the top are two commented assignments that substitute flex for lex and Berkeley yacc for AT&T yacc. Flex needs the *-I* flag to tell it to generate an interactive scanner, one that doesn't try to look ahead past a newline. The **CC** macro sets the preprocessor symbol **YYDEBUG** which compiles in some debugging code useful in testing the parser.

The rule for compiling everything together into ch3 refers to three libraries: the yacc library *-ly*, the lex library *-ll*, and the math library *-lm*. The yacc library provides **yyerror()** and, in early versions of the calculator, **main()**. The lex library provides some internal support routines that a lex scanner needs. (Scanners generated by flex don't need the library, but it does no harm to leave it in.) The math library provides **sqrt()**, **exp()**, and **log()**.

If we were to use bison, the GNU version of yacc, we'd have to change the rule that generates *y.tab.c* because bison uses different default filenames:

```
y.tab.c y.tab.h:  ch3yac.y
        bison -d ch3yac.y
        mv ch3yac.tab.c y.tab.c
        mv ch3yac.tab.h "y.tab.h
```

(Or we could change the rest of the *Makefile* and the code to use bison's more memorable names, or else use *-y* which tells bison to use the usual yacc filenames.)

For more details on *make*, see Steve Talbott's *Managing Projects with Make*, published by O'Reilly & Associates.

Summary

In this chapter, we've seen how to create a yacc grammar specification, put it together with a lexer to produce a working calculator, and extended the calculator to handle symbolic variable and function names. In the next two chapters, we'll work out larger and more realistic applications, a menu generator and a processor for the SQL database language.

Exercises

1. Add more functions to the calculator. Try adding two argument functions, e.g., modulus or arctangent, with a rule like this:

   ```
   expression: NAME '(' expression ',' expression ')'
   ```

   You should probably put a separate field in the symbol table for the two-argument functions, so you can call the appropriate version of **atan()** for one or two arguments.
2. Add a string data type, so you can assign strings to variables and use them in expressions or function calls. Add a STRING token for quoted literal strings. Change the value of an **expression** to a structure containing a tag for the type of value along with the value. Alternatively, extend the

grammar with a **stringexp** non-terminal for a string expression with a string (*char* *) value.

3. If you added a **stringexp** non-terminal, what happens if the user types this?

```
42 + "grapefruit"
```

   How hard is it to modify the grammar to allow mixed type expressions?

4. What do you have to do to handle assigning string values to variables?
5. How hard is it to overload operators, e.g., using "+" to mean catenation if the arguments are strings?
6. Add commands to the calculator to save and restore the variables to and from disk files.
7. Add user-defined functions to the calculator. The hard part is storing away the definition of the function in a way that you can re-execute when the user calls the function. One possibility is to save the stream of tokens that define the function. For example:

```
8.    statement: NAME '(' NAME ')' '=' { start_save($1, $3); }
9.        expression
           { end_save(); define_func($1, $3); }
```

   The functions **start_save()** and **end_save()** tell the lexer to save a list of all of the tokens for the expression. You need to identify references within the defining expression to the dummy argument **$3**.

   When the user calls the function, you play the tokens back:

```
     expression: USERFUNC '(' expression ')' { start_replay($1,
  $3); }
        expression/* replays the function */
        { $$ = $6; }/* use its value */
```

   While playing back the function, insert the argument value **$3** into the replayed expression in place of the dummy argument.

10. If you keep adding features to the calculator, you'll eventually end up with your own unique programming language. Would that be a good idea? Why or why not?