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

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

```
 */

%}
%%

[\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
```

```
%{
/*
 * 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.

```
        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:

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.

```
%}
%%

[\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); }
```

```
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 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

```
%%
```

```
^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

```
        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 */
```

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:

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**, **PREPOSITION**, 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

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 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

```
%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 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 **yy-parse()** reset the state and begin processing again. Our example prints a message if it sees a "sub-ject VERB object" list of input tokens. What happens if it sees "subject subject" or some other in-valid 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 pro-vided 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.

```
%%

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)
```

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:

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;
```

```
    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; }
```

```
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

[1] You can also use a vertical bar within a pattern, e.g., **foo|bar** is a pattern that matches either the string “foo” or the string “bar.” We leave some space between the pattern and the vertical bar to indicate that “bar” is the action, not part of the pattern.

[2] Actually, we could have left out the main program because the lex library contains a default main routine just like this one.

[3] We say symbol rather than token here, because we reserve the name “token” for symbols returned from the lexer, and the symbol to the left of the arrow did not come from the lexer. All tokens are symbols, but not all symbols are tokens.