UNIX Programming Tools

2nd Edition

lex & yacc

John R. Levine,
Tony Mason & Doug Brown

O'REILLY®

lex & yacc, 2nd Edition

Doug Brown

John Levine

Tony Mason

O'REILLY

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

A Note Regarding Supplemental Files

Supplemental files and examples for this book can be found at http://examples.oreilly.com/9781565920002/. Please use a standard desktop web browser to access these files, as they may not be accessible from all ereader devices.

All code files or examples referenced in the book will be available online. For physical books that sh with an accompanying disc, whenever possible, we've posted all CD/DVD content. Note that while w provide as much of the media content as we are able via free download, we are sometimes limited by licensing restrictions. Please direct any questions or concerns to booktech@oreilly.com.

Preface

Lex and yacc are tools designed for writers of compilers and interpreters, although they are also usef for many applications that will interest the noncompiler writer. Any application that looks for pattern in its input, or has an input or command language is a good candidate for lex and yacc. Furthermore, they allow for rapid application prototyping, easy modification, and simple maintenance of programs To stimulate your imagination, here are a few things people have used lex and yacc to develop:

- The desktop calculator bc
- The tools eqn and pic, typesetting preprocessors for mathematical equations and complex pictures
- PCC, the Portable C Compiler used with many UNIX systems, and GCC, the GNU C Compiler
- A menu compiler
- A SQL data base language syntax checker
- The lex program itself

What's New in the Second Edition

We have made extensive revisions in this new second edition. Major changes include:

- Completely rewritten introductory Chapters 1–3
- New Chapter 5 with a full SQL grammar
- New, much more extensive reference chapters
- Full coverage of all major MS-DOS and UNIX versions of lex and yacc, including AT&T lex and yacc, Berkeley yacc, flex, GNU bison, MKS lex and yacc, and Abraxas PCYACC
- Coverage of the new POSIX 1003.2 standard versions of lex and yacc

Scope of This Book

Chapter 1, Lex and Yacc , gives an overview of how and why lex and yacc are used to create compile and interpreters, and demonstrates some small lex and yacc applications. It also introduces basic terms we use throughout the book.

Chapter 2, Using Lex , describes how to use lex. It develops lex applications that count words in files analyze program command switches and arguments, and compute statistics on C programs.

Chapter 3, Using Yacc , gives a full example using lex and yacc to develop a fully functional desktop calculator.

Chapter 4, A Menu Generation Language , demonstrates how to use lex and yacc to develop a menu generator.

Chapter 5, Parsing SQL , develops a parser for the full SQL relational data base language. First we us the parser as a syntax checker, then extend it into a simple preprocessor for SQL embedded in C

programs.

Chapter 6, A Reference for Lex Specifications, and Chapter 7, A Reference for Yacc Grammars, provide detailed descriptions of the features and options available to the lex and yacc programmer. These chapters and the two that follow provide technical information for the now experienced lex and yacc programmer to use while developing new lex and yacc applications.

Chapter 8, Yacc Ambiguities and Conflicts, explains yacc ambiguities and conflicts, which are problems that keep yacc from parsing a grammar correctly. It then develops methods that can be used to locate and correct such problems.

Chapter 9, Error Reporting and Recovery, discusses techniques that the compiler or interpreter designer can use to locate, recognize, and report errors in the compiler input.

Appendix A, AT&T Lex, describes the command-line syntax of AT&T lex and the error messages it reports and suggests possible solutions.

Appendix B, AT&T Yacc, describes the command-line syntax of AT&T yacc and lists errors reported by yacc. It provides examples of code which can cause such errors and suggests possible solutions.

Appendix C, Berkeley Yacc, describes the command-line syntax of Berkeley yacc, a widely used free version of yacc distributed with Berkeley UNIX, and lists errors reported by Berkeley yacc with suggested solutions.

Appendix D, GNU Bison, discusses differences found in bison, the Free Software Foundation's implementation of yacc.

Appendix E, Flex, discusses flex, a widely used free version of lex, lists differences from other versions, and lists errors reported by flex with suggested solutions.

Appendix F, MKS Lex and Yacc, discusses the MS-DOS and OS/2 version of lex and yacc from Mortice Kern Systems.

Appendix G, Abraxas Lex and Yacc, discusses PCYACC, the MS-DOS and OS/2 versions of lex and yacc from Abraxas Software.

Appendix H, POSIX Lex and Yacc, discusses the versions of lex and yacc defined by the IEEE POSIX 1003.2 standard.

Appendix I, MGL Compiler Code, provides the complete source code for the menu generation language compiler discussed in Chapter 4.

Appendix J, SQL Parser Code, provides the complete source code and a cross-reference for the SQL parser discussed in Chapter 5.

The Glossary lists technical terms language and compiler theory.

The Bibliography lists other documentation on lex and yacc, as well as helpful books on compiler design.

We presume the reader is familiar with C, as most examples are in C, lex, or yacc, with the remainder being in the special purpose languages developed within the text.

Availability of Lex and Yacc

Lex and yacc were both developed at Bell Laboratories in the 1970s. Yacc was the first of the two, developed by Stephen C. Johnson. Lex was designed by Mike Lesk and Eric Schmidt to work with yacc. Both lex and yacc have been standard UNIX utilities since 7th Edition UNIX. System V and older versions of BSD use the original AT&T versions, while the newest version of BSD uses flex (se below) and Berkeley yacc. The articles written by the developers remain the primary source of information on lex and yacc.

The GNU Project of the Free Software Foundation distributes bison, a yacc replacement; bison was written by Robert Corbett and Richard Stallman. The bison manual, written by Charles Donnelly and Richard Stallman, is excellent, especially for referencing specific features. Appendix D discusses bison.

BSD and GNU Project also distribute flex (Fast Lexical Analyzer Generator), "a rewrite of lex intended to right some of that tool's deficiencies," according to its reference page. Flex was originall written by Jef Poskanzer; Vern Paxson and Van Jacobson have considerably improved it and Vern currently maintains it. Appendix E covers topics specific to flex.

There are at least two versions of lex and yacc available for MS-DOS and OS/2 machines. MKS (Mortice Kern Systems Inc.), publishers of the MKS Toolkit, offers lex and yacc as a separate produc that supports many PC C compilers. MKS lex and yacc comes with a very good manual. Appendix F covers MKS lex and yacc. Abraxas Software publishes PCYACC, a version of lex and yacc which comes with sample parsers for a dozen widely used programming languages. Appendix G covers Abraxas' version lex and yacc.

Sample Programs

The programs in this book are available free from UUNET (that is, free except for UUNET's usual connect-time charges). If you have access to UUNET, you can retrieve the source code using UUCP o FTP. For UUCP, find a machine with direct access to UUNET, and type the following command:

```
uucp uunet\!~/nutshell/lexyacc/progs.tar.Z yourhost\!~/yourname/
```

The backslashes can be omitted if you use the Bourne shell (sh) instead of the C shell (csh). The file should appear some time later (up to a day or more) in the directory /usr/spool/uucppublic/**yourname**. If you don't have an account but would like one so that you can get electronic mail, then contact UUNET at 703-204-8000.

To use ftp, find a machine with direct access to the Internet. Here is a sample session, with command in boldface.

```
% ftp ftp.oreilly.com
Connected to ftp.oreilly.com.
220 FTP server (Version 5.99 Wed May 23 14:40:19 EDT 1990) ready.
Name (ftp.oreilly.com:yourname): anonymous
331 Guest login ok, send ident as password.
Password: ambar@ora.com
        (use your user name and host here)
230 Guest login ok, access restrictions apply.
ftp> cd published/oreilly/nutshell/lexyacc
```

```
250 CWD command successful.
ftp> binary (you must specify binary transfer for compressed files)
200 Type set to I.
ftp> get progs.tar.Z
200 PORT command successful.
150 Opening BINARY mode data connection for progs.tar.Z.
226 Transfer complete.
ftp> quit
221 Goodbye.
%
```

The file is a compressed tar archive. To extract files once you have retrieved the archive, type:

```
% zcat  progs.tar.Z | tar xf -
```

System V systems require the following tar command instead:

```
% zcat progs.tar.Z | tar xof -
```

Conventions Used in This Handbook

The following conventions are used in this book:

Bold

is used for statements and functions, identifiers, and program names.

Italic

is used for file, directory, and command names when they appear in the body of a paragraph as well as for data types and to emphasize new terms and concepts when they are introduced.

`Constant Width`

is used in examples to show the contents of files or the output from commands.

`Constant Bold`

is used in examples to show command lines and options that you type literally.

Quotes

are used to identify a code fragment in explanatory text. System messages, signs, and symbols ar quoted as well.

%

is the Shell prompt.

[]

surround optional elements in a description of program syntax. (Don't type the brackets themselves.)

Acknowledgments

This first edition of this book began with Tony Mason's MGL and SGL compilers. Tony developed most of the material in this book, working with Dale Dougherty to make it a "Nutshell." Doug Brown contributed Chapter 8, Yacc Ambiguities and Conflicts . Dale also edited and revised portions of the

book. Tim O'Reilly made it a better book by withholding his editorial blessing until he found what he was looking for in the book. Thanks to Butch Anton, Ed Engler, and Mike Loukides for their comments on technical content. Thanks also to John W. Lockhart for reading a draft with an eye for stylistic issues. And thanks to Chris Reilley for his work on the graphics. Finally, Ruth Terry brought the book into print with her usual diligence and her sharp eye for every editorial detail. Though she was trying to work odd hours to also care for her family, it seemed she was caring for this book all hours of the day.

For the second edition, Tony rewrote chapters 1 and 2, and Doug updated Chapter 8. John Levine wrote Chapters 3, 5, 6, 7, and most of the appendices, and edited the rest of the text. Thanks to the technical reviewers, Bill Burke, Warren Carithers, Jon Mauney, Gary Merrill, Eugene Miya, Andy Oram, Bill Torcaso, and particularly Vern Paxson whose detailed page-by-page suggestions made the fine points much clearer. Margaret Levine Young's blue pencil (which was actually pink) tightened up the text and gave the book editorial consistency. She also compiled most of the index. Chris Reilly again did the graphics, and Donna Woonteiler did the final editing and shepherded the book through the production process.

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 possibl 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 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 yo 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 b 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 identifyin 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, th

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 th 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 ": 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 othe names used in the program in its symbol table. Each name is stored along with information describin 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 declarin Whenever we see a line starting with the name of a part of speech, we set the state to declare that kin 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 variable **state**. We use this enumerated type both in the state variable to track what we're defining an 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 a 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, usin **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 tabl 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 o 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] F 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 fundamenta 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. Howeve 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 t

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 yac 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 thes 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 th 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 intrinsi meaning to yacc, although well-chosen token names tell the reader what they represent. Although yac lets you use any valid C identifier name for a yacc symbol, universal custom dictates that token name 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 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 le and then for finding the relationship among the words with yacc are much the same as we’ll use in th practical applications in later chapters. For example, in this C language statement,

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

sample content of lex & yacc

- read Solo: A Memoir of Hope pdf
- download The Noodle Narratives: The Global Rise of an Industrial Food into the Twenty-First Century
- download La force de l'Ã¢ge here
- read Myth and Reality
- read online Soft Errors in Modern Electronic Systems (Frontiers in Electronic Testing) book
- download online Mani-Pedi STAT: Memoirs of a Jersey Girl Who Almost Lost Everything online

- http://hasanetmekci.com/ebooks/Solo--A-Memoir-of-Hope.pdf
- http://aneventshop.com/ebooks/Romantic-Image--Routledge-Classics-.pdf
- http://honareavalmusic.com/?books/Brando-Unzipped--Marlon-Brando--Bad-Boy--Megastar--Sexual-Outlaw.pdf
- http://sidenoter.com/?ebooks/Myth-and-Reality.pdf
- http://www.freightunlocked.co.uk/lib/Soft-Errors-in-Modern-Electronic-Systems--Frontiers-in-Electronic-Testing-.pdf
- http://studystrategically.com/freebooks/Mani-Pedi-STAT--Memoirs-of-a-Jersey-Girl-Who-Almost-Lost-Everything.pdf