To properly understand how Lex interprets such an input file, we must keep in mind that some parts of the file will be regular expression information that Lex uses to guide its construction of the C output code, while other parts of the file will be actual C code that we are supplying to Lex, and that Lex will insert verbatim in the output code at the appropriate location. The precise rules Lex uses for this will be given after we have discussed each of the three sections and given a few examples.

The definition section occurs before the first `%%`. It contains two things. First, any C code that must be inserted external to any function should appear in this section between the delimiters `%{` and `%}`. (Note the order of these characters!) Second, names for regular expressions must also be defined in this section. A name is defined by writing it on a separate line starting in the first column and following it (after one or more blanks) by the regular expression it represents.

The second section contains the rules. These consist of a sequence of regular expressions followed by the C code that is to be executed when the corresponding regular expression is matched.

The third and final section contains C code for any auxiliary routines that are called in the second section and not defined elsewhere. This section may also contain a main program, if we want to compile the Lex output as a standalone program. This section can also be missing, in which case the second `%%` need not be written. (The first `%%` is always necessary.)

We give a series of examples to illustrate the format of a Lex input file.

Example 2.20

The following Lex input specifies a scanner that adds line numbers to text, sending its output to the screen (or a file, if redirected):

```
%{
/* a Lex program that adds line numbers
   to lines of text, printing the new text
   to the standard output
*/
#include <stdio.h>
int lineno = 1;
%}
line .*\n
%%
{line} { printf("%5d %s",lineno++,yytext); }
%%
main()
{ yylex(); return 0; }
```

For example, running the program obtained from Lex on this input file itself gives the following output:

```
 1 %{
 2 /* a Lex program that adds line numbers
 3    to lines of text, printing the new text
 4    to the standard output
 5 */
 6 #include <stdio.h>
 7 int lineno = 1;
 8 %}
 9 line .*\n
10 %%
11 {line} { printf("%5d %s",lineno++,yytext); }
12 %%
13 main()
14 { yylex(); return 0; }
```

We comment on this Lex input file using these line numbers. First, lines 1 through 8 are between the delimiters `%{` and `%}`. This causes these lines to be inserted directly into the C code produced by Lex, external to any procedure. In particular, the comment in lines 2 through 5 will be inserted near the beginning of the program, and the **`#include`** directive and the definition of the integer variable **`lineno`** on lines 6 and 7 will be inserted externally, so that **`lineno`** becomes a global variable and is initialized to the value 1. The other definition that appears before the first `%%` is the definition of the name **`line`** which is defined to be the regular expression **`".*\n"`**, which matches 0 or more characters (not including a newline), followed by a newline. In other words, the regular expression defined by **`line`** matches every line of input. Following the `%%` on line 10, line 11 comprises the action section of the Lex input file. In this case we have written a single action to be performed whenever a **`line`** is matched (**`line`** is surrounded by curly brackets to distinguish it as a name, according to the Lex convention). Following the regular expression is the **action**, that is, the C code that is to be executed whenever the regular expression is matched. In this example, the action consists of a single C statement, which is contained within the curly brackets of a C block. (Keep in mind that the curly brackets surrounding the name **`line`** have a completely different function from the curly brackets that form a block in the C code of the following action.) This C statement is to print the line number (in a five-space field, right justified) and the string **`yytext`**, after which **`lineno`** is incremented. The name **`yytext`** is the internal name Lex gives to the string matched by the regular expression, which in this case consists of each line of input (including the newline).[5] Finally, the C code after the second double percent (lines 13 and 14) is inserted as is at the end of the C code produced by Lex. In this example, the code consists of the definition of a **`main`** procedure that calls the function **`yylex`**. This allows the C code produced by Lex to be compiled into an executable program. (**`yylex`** is the name given to the procedure constructed by Lex that implements the DFA associated with the regular expressions and actions given in the action section of the input file.) §

5. We list the Lex internal names that are discussed in this section in a table at the end of the section.

Example 2.21 Consider the following Lex input file:

```
%{
/* a Lex program that changes all numbers
   from decimal to hexadecimal notation,
   printing a summary statistic to stderr
*/
#include <stdlib.h>
#include <stdio.h>
int count = 0;
%}
digit [0-9]
number {digit}+
%%
{number} { int n = atoi(yytext);
           printf("%x", n);
           if (n > 9) count++; }
%%
main()
{ yylex();
  fprintf(stderr,"number of replacements = %d",
                 count);
  return 0;
}
```

It is similar in structure to the previous example, except that the **main** procedure prints the count of the number of replacements to **stderr** after calling **yylex**. This example is also different in that not all text is matched. Indeed, only numbers are matched in the action section, where the C code for the action first converts the matched string (**yytext**) to an integer **n**, then prints it in hexadecimal form (**printf("%x",...)**), and finally increments **count** if the number is greater than 9. (If the number is smaller than or equal to 9, then it looks no different in hex.) Thus, the only action specified is for strings that are sequences of digits. Yet Lex generates a program that also matches all nonnumeric characters, and passes them through to the output. This is an example of a **default action** by Lex. If a character or string of characters matches none of the regular expressions in the action section, Lex will match it by default and echo it to the output. (Lex can also be forced to generate a runtime error, but we will not study this here.) The default action can also be specifically indicated through the Lex internally defined macro **ECHO**. (We study this use in the next example.) §

Example 2.22 Consider the following Lex input file:

```
%{
/* Selects only lines that end or
   begin with the letter 'a'.
   Deletes everything else.
*/
#include <stdio.h>
%}
ends_with_a .*a\n
begins_with_a a.*\n
%%
{ends_with_a} ECHO;
{begins_with_a} ECHO;
.*\n ;
%%
main()
{ yylex(); return 0; }
```

This Lex input causes all input lines that begin or end with the character *a* to be written to the output. All other input lines are suppressed. The suppression of the input is caused by the rule below the **ECHO** rules. In this rule the "empty" action is specified for the regular expression **.*\n** by writing a semicolon for the C action code.

There is an additional feature of this Lex input that is worth noting. The listed rules are **ambiguous** in that a string may match more than one rule. In fact, *any* input line matches the expression **.*\n**, regardless of whether it is part of a line that begins or ends with an *a*. Lex has a priority system for resolving such ambiguities. First, Lex always matches the longest possible substring (so Lex always generates a scanner that follows the longest substring principle). Then, if the longest substring still matches two or more rules, Lex picks the first rule in the order they are listed in the action section. It is for this reason that the above Lex input file lists the **ECHO** actions first. If we had listed the actions in the following order,

```
.*\n ;
{ends_with_a} ECHO;
{begins_with_a} ECHO;
```

then the program produced by Lex would generate no output at all for any file, since every line of input will be matched by the first rule. §

Example 2.23 In this example, Lex generates a program that will convert all uppercase letters to lowercase, except for letters inside C-style comments (that is, anything inside the delimiters **/*...*/**):

```
%{
/* Lex program to convert uppercase to
   lowercase except inside comments
*/
#include <stdio.h>
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
#endif
%}
%%
[A-Z] {putchar(tolower(yytext[0]));
        /* yytext[0] is the single
           uppercase char found */
       }
"/*"   { char c;
         int done = FALSE;
         ECHO;
         do
         { while ((c=input())!='*')
             putchar(c);
           putchar(c);
           while ((c=input())=='*')
             putchar(c);
           putchar(c);
           if (c == '/') done = TRUE;
         } while (!done);
       }
%%
void main(void)
{ yylex();}
```

This example shows how code can be written to sidestep difficult regular expressions and implement a small DFA directly as a Lex action. Recall from the discussion in Section 2.2.4 that a regular expression for a C comment is extremely complex to write down. Instead, we write down a regular expression only for the string that begins a C comment—that is, `"/*"`—and then supply action code that will look for the ending string `"*/"`, while providing the appropriate action for other characters within the comment (in this case to just echo them without further processing). We do this by imitating the DFA from Example 2.9 (see Figure 2.4, page 53). Once we have recognized the string `"/*"`, we are in state 3, so our code picks up the DFA there. The first thing we do is cycle through characters (echoing them to the output) until we see an asterisk (corresponding to the ***other*** loop in state 3), as follows:

```
while ((c=input())!='*') putchar(c);
```

Here we have used yet another Lex internal procedure called **`input`**. The use of this procedure, rather than a direct input using **`getchar`**, ensures that the Lex input buffer is used, and that the internal structure of the input string is preserved. (Note, however, that we do use a direct output procedure **`putchar`**. This will be discussed further in Section 2.6.4.)

110 §.2.6.4

The next step in our code for the DFA corresponds to state 4. We loop again until we do *not* see an asterisk, and then, if the character is a forward slash, we exit; otherwise, we return to state 3. §

We end this subsection with a summary of the Lex conventions we have introduced in the examples.

AMBIGUITY RESOLUTION
Lex's output will always first match the longest possible substring to a rule. If two or more rules cause substrings of equal length to be matched, then Lex's output will pick the rule listed first in the action section. If no rule matches any nonempty substring, then the default action copies the next character to the output and continues.

INSERTION OF C CODE
(1) Any text written between **`%{`** and **`%}`** in the definition section will be copied directly to the output program external to any procedure. (2) Any text in the auxiliary procedures section will be copied directly to the output program at the end of the Lex code. (3) Any code that follows a regular expression (by at least one space) in the action section (after the first **`%%`**) will be inserted at the appropriate place in the recognition procedure **`yylex`** and will be executed when a match of the corresponding regular expression occurs. The C code representing an action may be either a single C statement or a compound C statement consisting of any declarations and statements surrounded by curly brackets.

INTERNAL NAMES
Table 2.3 lists the Lex internal names that we discuss in this chapter. Most of these have been discussed in the previous examples.

Table 2.3
Some Lex internal names

Lex Internal Name	Meaning/Use
`lex.yy.c` or `lexyy.c`	Lex output file name
`yylex`	Lex scanning routine
`yytext`	string matched on current action
`yyin`	Lex input file (default: `stdin`)
`yyout`	Lex output file (default: `stdout`)
`input`	Lex buffered input routine
`ECHO`	Lex default action (print `yytext` to `yyout`)

We note one feature from this table not mentioned previously, which is that Lex has its own internal names for the files from which it takes input and to which it sends output: **`yyin`** and **`yyout`**. Using the standard Lex input routine **`input`** will automatically take input from the file **`yyin`**. However, in the foregoing examples, we have bypassed the internal output file **`yyout`** and just written to the standard output using **`printf`** and **`putchar`**. A better implementation, allowing the assignment of output to an arbitrary file, would replace these uses with **`fprintf(yyout,...)`** and **`putc(...,yyout)`**.

2.6.3 A TINY Scanner Using Lex

Appendix B gives a listing of a Lex input file `tiny.l` that will generate a scanner for the TINY language, whose tokens were described in Section 2.5 (see Table 2.1). In the following we make a few remarks about this input file (lines 3000–3072).

First, in the definitions section, the C code we insert directly into the Lex output consists of three `#include` directives (`globals.h`, `util.h`, and `scan.h`) and the definition of the `tokenString` attribute. This is necessary to provide the interface between the scanner and the rest of the TINY compiler.

The further contents of the definition section comprise the definitions of the names for the regular expressions that define the TINY tokens. Note that the definition of `number` uses the previously defined name `digit`, and the definition of `identifier` uses the previously defined `letter`. The definitions also distinguish between newlines and other white space (blanks and tabs, lines 3019 and 3020), since a newline will cause `lineno` to be incremented.

The action section of the Lex input consists of listing the various tokens, together with a `return` statement that returns the appropriate token as defined in `globals.h`. In this Lex definition we have listed the rules for reserved words before the rule for an identifier. Had we listed the identifier rule first, the ambiguity resolution rules of Lex would cause an identifier to always be recognized instead of a reserved word. We could also write code as in the scanner of the previous section, in which only identifiers are recognized, and then reserved words are looked up in a table. This would indeed be preferable in a real compiler, since separately recognized reserved words cause the size of the tables in the scanner code generated by Lex to grow enormously (and hence the size of memory used by the scanner).

One quirk of the Lex input is that we have to write code to recognize comments to ensure that `lineno` is updated correctly, even though the regular expression for TINY comments is easy to write. Indeed, the regular expression is

```
"{"[^\}]*"}"
```

(Note the use of the backslash inside the square brackets to remove the metacharacter meaning of right curly bracket—quotes will not work here.)[6]

We note also that there is no code written to return the `EOF` token on encountering the end of the input file. The Lex procedure `yylex` has a default behavior on encountering `EOF`—it returns the value 0. It is for this reason that the token `ENDFILE` was written first in the definition of `TokenType` in `globals.h` (line 179), so that it will have value 0.

Finally, the `tiny.l` file contains a definition of the `getToken` procedure in the auxiliary procedures section (lines 3056–3072). While this code contains some ad hoc initializations of Lex internals (such as `yyin` and `yyout`) that would be better performed directly in the main program, it does permit us to use the Lex-generated scanner directly, without changing any other files in the TINY compiler. Indeed, after generating the C scanner file `lex.yy.c` (or `lexyy.c`), this file can be compiled and linked directly with the other TINY source files to produce a Lex-based version of the compiler. However, this version of the compiler lacks one service of the earlier version, in that no source code echoing with line numbers is provided (see Exercise 2.35).

6. Some versions of Lex have an internally defined variable `yylineno` that is automatically updated. Use of this variable instead of `lineno` would make it possible to eliminate the special code.

EXERCISES

2.1 Write regular expressions for the following character sets, or give reasons why no regular expression can be written:
- **a.** All strings of lowercase letters that begin and end in *a*.
- **b.** All strings of lowercase letters that either begin or end in *a* (or both).
- **c.** All strings of digits that contain no leading zeros.
- **d.** All strings of digits that represent even numbers.
- **e.** All strings of digits such that all the 2's occur before all the 9's.
- **f.** All strings of *a*'s and *b*'s that contain no three consecutive *b*'s.
- **g.** All strings of *a*'s and *b*'s that contain an odd number of *a*'s or an odd number of *b*'s (or both).
- **h.** All strings of *a*'s and *b*'s that contain an even number of *a*'s and an even number of *b*'s.
- **i.** All strings of *a*'s and *b*'s that contain exactly as many *a*'s as *b*'s.

2.2 Write English descriptions for the languages generated by the following regular expressions:
- **a.** `(a|b)*a(a|b|ε)`
- **b.** `(A|B|...|Z)(a|b|...|z)*`
- **c.** `(aa|b)*(a|bb)*`
- **d.** `(0|1|...|9|A|B|C|D|E|F)+(x|X)`

2.3 **a.** Many systems contain a version of **grep** (**g**lobal **r**egular **e**xpression **p**rint), a regular expression search program originally written for Unix.[7] Find a document describing your local grep, and describe its metasymbol conventions.
- **b.** If your editor accepts some sort of regular expressions for its string searches, describe its metasymbol conventions.

2.4 In the definition of regular expressions, we described the precedence of the operations, but not their associativity. For example, we did not specify whether `a|b|c` meant `(a|b)|c` or `a|(b|c)` and similarly for concatenation. Why was this?

2.5 Prove that $L(r^{**}) = L(r^*)$ for any regular expression r.

2.6 In describing the tokens of a programming language using regular expressions, it is not necessary to have the metasymbols ϕ (for the empty set) or ε (for the empty string). Why is this?

2.7 Draw a DFA corresponding to the regular expression ϕ.

2.8 Draw DFAs for each of the sets of characters of **(a)**–**(i)** in Exercise 2.1, or state why no DFA exists.

2.9 Draw a DFA that accepts the four reserved words `case`, `char`, `const`, and `continue` from the C language.

7. There are actually three versions of grep available on most Unix systems: "regular" grep, egrep (extended grep), and fgrep (fast grep).

2.10 Rewrite the pseudocode for the implementation of the DFA for C comments (Section 2.3.3) using the input character as the outer case test and the state as the inner case test. Compare your pseudocode to that of the text. When would you prefer to use this organization for code implementing a DFA?

2.11 Give a mathematical definition of the ε-closure of a set of states of an NFA.

2.12 **a.** Use Thompson's construction to convert the regular expression **(a|b)*a(a|b|**ε**)** into an NFA.
b. Convert the NFA of part (a) into a DFA using the subset construction.

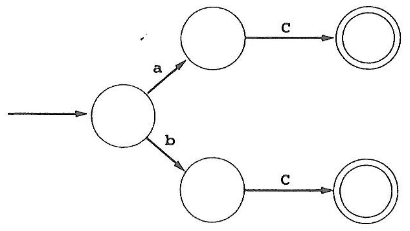
2.13 **a.** Use Thompson's construction to convert the regular expression **(aa|b)*(a|bb)*** into an NFA.
b. Convert the NFA of part (a) into a DFA using the subset construction.

2.14 Convert the NFA of Example 2.10 (Section 2.3.2) into a DFA using the subset construction.

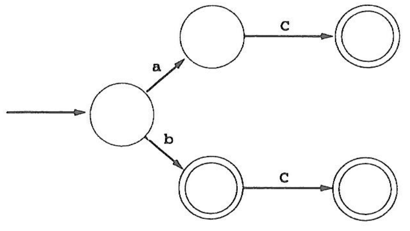
2.15 In Section 2.4.1 a simplification to Thompson's construction for concatenation was mentioned that eliminates the ε-transition between the two NFAs of the regular expressions being concatenated. It was also mentioned that this simplification needed the fact there were no transitions out of the accepting state in the other steps of the construction. Give an example to show why this is so. (Hint: Consider a new NFA construction for repetition that eliminates the new start and accepting states, and then consider the NFA for r^*s^*.)

2.16 Apply the state minimization algorithm of Section 2.4.4 to the following DFAs:

a.



b.



2.17 Pascal comments allow two distinct comment conventions: curly bracket pairs `{ ... }` (as in TINY) and parentheses-asterisk pairs `(* ... *)`. Write a DFA that recognizes both styles of comment.

2.18 **a.** Write out a regular expression for C comments in Lex notation. (Hint: See the discussion in Section 2.2.3.)
b. Prove your answer in part (a) is correct.

2.19 The following regular expression has been given as a Lex definition of C comments (see Schreiner and Friedman [1985, p. 25]):

```
"/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/"
```

Show that this expression is incorrect. (Hint: Consider the string `/**_/*/`.)

PROGRAMMING EXERCISES

2.20 Write a program that capitalizes all comments in a C program.

2.21 Write a program that capitalizes all reserved words outside of comments in a C program. (A list of the reserved words of C can be found in Kernighan and Ritchie [1988, p. 192].)

2.22 Write a Lex input file that will produce a program that capitalizes all comments in a C program.

2.23 Write a Lex input file that will produce a program that capitalizes all reserved words outside of comments in a C program (see Exercise 2.21).

2.24 Write a Lex input file that will produce a program that counts characters, words, and lines in a text file and reports the counts. Define a word to be any sequence of letters and/or digits, without punctuation or spaces. Punctuation and white space do not count as words.

2.25 The Lex code of Example 2.23 (Section 2.6.2) can be shortened by using a global flag **inComment** to distinguish behavior inside comments from behavior elsewhere. Rewrite the code of the example to do this.

2.26 Add nested C comments to the Lex code of Example 2.23.

2.27 **a.** Rewrite the scanner for TINY to use binary search in the lookup of reserved words.
b. Rewrite the scanner for TINY to use a hash table for the lookup of reserved words.

2.28 Remove the 40-character limit on identifiers in the TINY scanner by dynamically allocating space for **tokenString**.

2.29 **a.** Test the behavior of the TINY scanner when the source program has lines that exceed the buffer size of the scanner, finding as many problems as you can.
b. Rewrite the TINY scanner to remove the problems that you found in part (a) (or at least improve its behavior). (This will require rewriting the **getNextChar** and **ungetNextChar** procedures.)

2.30 An alternative to the use of the **ungetNextChar** procedure in the TINY scanner to implement nonconsuming transitions is to use a Boolean flag to indicate that the current character is to be consumed, so that no backup in the input is required. Rewrite the TINY scanner to implement this method, and compare it to the existing code.

2.31 Add nested comments to the TINY scanner by using a counter called **nestLevel**.

2.32 Add Ada-style comments to the TINY scanner. (An Ada comment begins with two dashes and continues to the end of the line.)

2.33 Add the lookup of reserved words in a table to the Lex scanner for TINY (you may use linear search as in the handwritten TINY scanner or either of the search methods suggested in Exercise 2.27).

2.34 Add Ada-style comments to the Lex code for the TINY scanner. (An Ada comment begins with two dashes and continues to the end of the line.)

2.35 Add source code line echoing (using the **EchoSource** flag) to the Lex code for the TINY scanner, so that, when the flag is set, each line of source code is printed to the listing file with the line number. (This requires more extensive knowledge of Lex internals than we have studied.)

NOTES AND REFERENCES

The mathematical theory of regular expressions and finite automata is discussed in detail in Hopcroft and Ullman [1979], where some references to the historical development of the theory can be found. In particular, one can find there a proof of the equivalence of finite automata and regular expressions (we have used only one direction of the equivalence in this chapter). One can also find a discussion of the pumping lemma there, and its consequences for the limitations of regular expressions in describing patterns. A more detailed description of the state minimization algorithm can also be found there, together with a proof of the fact that such DFAs are essentially unique. The description of a one-step construction of a DFA from a regular expression (as opposed to the two-step approach described here) can be found in Aho, Hopcroft, and Ullman [1986]. A method for compressing tables in a table-driven scanner is also given there. A description of Thompson's construction using rather different NFA conventions from those described in this chapter is given in Sedgewick [1990], where descriptions of table lookup algorithms such as binary search and hashing for reserved word recognition can be found. (Hashing is also discussed in Chapter 6.) Minimal perfect hash functions, mentioned in Section 2.5.2, are discussed in Cichelli [1980] and Sager [1985]. A utility called **gperf** is distributed as part of the Gnu compiler package. It can quickly generate perfect hash functions for even large sets of reserved words. While these are generally not minimal, they are still quite useful in practice. Gperf is described in Schmidt [1990].

The original description of the Lex scanner generator is in Lesk [1975], which is still relatively accurate for more recent versions. Later versions, especially Flex (Paxson [1990]), have solved some efficiency problems and are competitive with even finely tuned handwritten scanners (Jacobson [1987]). A useful description of Lex can also be found in Schreiner and Friedman [1985], together with more examples of simple Lex programs to solve a variety of pattern-matching tasks. A brief description of the grep family of pattern matchers (Exercise 2.3) can be found in Kernighan and Pike [1984], and a more extensive discussion in Aho [1979]. The offside rule mentioned in Section 2.2.3 (a use of white space to provide formatting) is discussed in Landin [1966] and Hutton [1992].

Chapter 3

Context-Free Grammars and Parsing

Parsing is the task of determining the syntax, or structure, of a program. For this reason, it is also called **syntax analysis**. The syntax of a programming language is usually given by the **grammar rules** of a **context-free grammar**, in a manner similar to the way the lexical structure of the tokens recognized by the scanner is given by regular expressions. Indeed, a context-free grammar uses naming conventions and operations very similar to those of regular expressions. The major difference is that the rules of a context-free grammar are **recursive**. For instance, the structure of an if-statement must in general allow other if-statements to be nested inside it, something that is not allowed in regular expressions. The consequences of this seemingly elementary change to the power of the representation are enormous. The class of structures recognizable by context-free grammars is increased significantly over those of regular expressions. The algorithms used to recognize these structures are also quite different from scanning algorithms, in that they must use recursive calls or an explicitly managed parsing stack. The data structures used to represent the syntactic structure of a language must now also be recursive rather than linear (as they are for lexemes and tokens). The basic structure used is usually some kind of tree, called a **parse tree** or **syntax tree**.

In a similar manner to the previous chapter, we need to study the theory of context-free grammars before we study parsing algorithms and the details of actual parsers using these algorithms. However, contrary to the situation with scanners, where there is essentially only one algorithmic method (represented by finite automata), parsing involves a choice from among a number of different methods, each of which has distinct properties and capabilities. There are in fact two general