Chapter 2. Using Flex

In this chapter we'll take a closer look at flex as a standalone tool, with some examples that exercise most of its C language capabilities. All of flex's facilities are described in Chapter 5, and the usage of flex scanners in C++ programs is described in Chapter 9.

Regular Expressions

The patterns at the heart of every flex scanner use a rich regular expression language. A *regular expression* is a pattern description using a *metalanguage*, a language that you use to describe what you want the pattern to match. Flex's regular expression language is essentially POSIX-extended regular expressions (which is not surprising considering their shared Unix heritage). The metalanguage uses standard text characters, some of which represent themselves and others of which represent patterns. All characters other than the ones listed below, including all letters and digits, match themselves.

The characters with special meaning in regular expressions are:

.

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

[]

A *character class* that matches any character within the brackets. If the first character is a circumflex (`^`), it changes the meaning to match any character *except* the ones within the brackets. A dash inside the square brackets indicates a character range; for example, `[0-9]` means the same thing as `[0123456789]` and `[a-z]` means any lowercase letter. A `-` or `]` as the first character after the `[` is interpreted literally to let you include dashes and square brackets in character classes. POSIX introduced other special square bracket constructs that are useful when handling non-English alphabets, described later in this chapter. Other metacharacters do not have any special meaning within square brackets except that C escape sequences starting with `\` are recognized. Character ranges are interpreted relative to the character coding in use, so the range `[A-z]` with ASCII character coding would match all uppercase and lowercase letters, as well as six punctuation characters whose codes fall between the code for `Z` and the code for `a`. In practice, useful ranges are ranges of digits, of uppercase letters, or of lowercase letters.

[a-z]{-}[jv]

A differenced character class, with the characters in the first class omitting the characters in the second class (only in recent versions of flex).

`^`

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

`$`

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

`{}`

If the braces contain one or two numbers, indicate the minimum and maximum number of times the previous pattern can match. For example, `A{1,3}` matches one to three occurrences of the letter *A*, and `0{5}` matches 00000. If the braces contain a name, they refer to a named pattern by that name.

`\`

Used to escape metacharacters and as part of the usual C escape sequences; for example, `\n` is a newline character, while `\*` is a literal asterisk.

`*`

Matches zero or more copies of the preceding expression. For example, `[ \t]*` is a common pattern to match optional spaces and tabs, that is, whitespace, which matches “ ”, “ <tab><tab>”, or an empty string.

`+`

Matches one or more occurrences of the preceding regular expression. For example, `[0-9]+` matches strings of digits such as `1`, `111`, or `123456` but not an empty string.

`?`

Matches zero or one occurrence of the preceding regular expression. For example, `-?[0-9]+` matches a signed number including an optional leading minus sign.

`|`

The *alternation* operator; matches either the preceding regular expression or the following regular expression. For example, `faith|hope|charity` matches any of the three virtues.

“...”

Anything within the quotation marks is treated literally. Metacharacters other than C escape sequences lose their meaning.

As a matter of style, it's good practice to quote any punctuation characters intended to be matched literally.

()

Groups a series of regular expressions together into a new regular expression. For example, `(01)` matches the character sequence 01, and `a(bc|de)` matches *abc* or *ade*. Parentheses are useful when building up complex patterns with *, +, ?, and `|`.

/

Trailing context, which means to match the regular expression preceding the slash but only if followed by the regular expression after the slash. For example, `0/1` matches `0` in the string `01` but would not match anything in the string `0` or `02`. The material matched by the pattern following the slash is not "consumed" and remains to be turned into subsequent tokens. Only one slash is permitted per pattern.

The repetition operators affect the smallest preceding expression, so `abc+` matches *ab* followed by one or more *c*'s. Use parentheses freely to be sure your expressions match what you want, such as `(abc)+` to match one or more repetitions of *abc*.

Regular Expression Examples

We can combine these characters to make quite complex and useful regular expression patterns. For example, consider the surprisingly difficult job of writing a pattern to match Fortran-style numbers, which consist of an optional sign, a string of digits that may contain a decimal point, optionally an exponent that is the letter `E`, an optional sign, and a string of digits. A pattern for an optional sign and a string of digits is simple enough:

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

Note that we put the hyphen as the first thing in `[-+]` so it wouldn't be taken to mean a character range.

Writing the pattern to match a string of digits with an optional decimal point is harder, because the decimal point can come at the beginning or end of the number. Here's a few near misses:

```
[-+]?[0-9.]+              matches too much, like 1.2.3.4
[-+]?[0-9]+\.?[0-9]+     matches too little, misses .12 or 12.
[-+]?[0-9]*\.?[0-9]+     doesn't match 12.
[-+]?[0-9]+\.?[0-9]*     doesn't match .12
[-+]?[0-9]*\.?[0-9]*     matches nothing, or a dot with no digits at all
```

It turns out that no combination of character classes, ?, *, and + will match a number with an optional decimal point. Fortunately, the alternation operator | does the trick by allowing the pattern to combine two versions, each of which individually isn't quite sufficient:

```
[-+]?([0-9]*\.?[0-9]+|[0-9]+\.)
[-+]?([0-9]*\.?[0-9]+|[0-9]+\.[0-9]*)  This is overkill but also works
```

The second example is internally ambiguous, because there are many strings that match either of the alternates, but that is no problem for flex's matching algorithm. (Flex also allows two *different* patterns to match the same input, which is also useful but requires more care by the programmer.)

Now we need to add on the optional exponent, for which the pattern is quite simple:

```
E(+|-)?[0-9]+
```

(We did the two sign characters as an alternation here rather than a character class; it's purely a matter of taste.) Now we glue the two together to get a Fortran number pattern:

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

Since the exponent part is optional, we used parens and a question mark to make it an optional part of the pattern. Note that our pattern now includes nested optional parts, which work fine and as shown here are often very useful.

This is about as complex a pattern as you'll find in most flex scanners. It's worth reiterating that complex patterns do *not* make the scanner any slower.[4] Write your patterns to match what you need to match, and trust flex to handle them.

How Flex Handles Ambiguous Patterns

Most flex programs are quite ambiguous, with multiple patterns that can match the same input. Flex resolves the ambiguity with two simple rules:

- Match the longest possible string every time the scanner matches input.
- In the case of a tie, use the pattern that appears first in the program.

These turn out to do the right thing in the vast majority of cases. Consider this snippet from a scanner for C source code:

```
"+"                     { return ADD; }
"="                     { return ASSIGN; }
"+="                    { return ASSIGNADD; }

"if"                    { return KEYWORDIF; }
"else"                  { return KEYWORDELSE; }
[a-zA-Z_][a-zA-Z0-9_]*  { return IDENTIFIER; }
```

For the first three patterns, the string `+=` is matched as one token, since `+=` is longer than `+`. For the last three patterns, so long as the patterns for keywords precede the pattern that matches an identifier, the scanner will match keywords correctly.

Context-Dependent Tokens

In some languages, scanning is context dependent. For example, in Pascal, `1.` is usually a floating-point number, but in a declaration, `1..2` is two integers separated by a `..` token. Flex provides *start states*, which can turn patterns on and off dynamically and are generally sufficient to handle such context dependencies. We discuss start states later in this chapter.

File I/O in Flex Scanners

Flex scanners will read from the standard input unless you tell them otherwise. In practice, most scanners read from files. We'll modify the word count program from Example 1-1 to read from files, like the real `wc` program does.

The I/O options available in scanners generated by flex and its predecessor lex have undergone extensive evolution over the past 30 years, so there are several different ways to manage a scanner's input and output. Unless you make other arrangements, a scanner reads from the stdio `FILE` called `yyin`, so to read a single file, you need only set it before the first call to `yylex`. In Example 2-1, we add the ability to specify an input file to the word count program from Example 1-1.

Example 2-1. Word count, reading one file

```
/* even more like Unix wc */
%option noyywrap
%{
int chars = 0;
int words = 0;
int lines = 0;
%}

%%
```

```
[a-zA-Z]+  { words++; chars += strlen(yytext); }
\n         { chars++; lines++; }
.          { chars++; }

%%

main(argc, argv)
int argc;
char **argv;
{
  if(argc > 1) {
    if(!(yyin = fopen(argv[1], "r"))) {
      perror(argv[1]);
      return (1);
    }
  }

  yylex();
  printf("%8d%8d%8d\n", lines, words, chars);
}
```

The only differences from Example 1-1 are in the code in the third section. The main routine opens a filename passed on the command line, if the user specified one, and assigns the `FILE` to `yyin`. Otherwise, `yyin` is left unset, in which case `yylex` automatically sets it to `stdin`.

THE FLEX LIBRARY

Lex and flex have always come with a small library now known as `-lfl` that defines a default `main` routine, as well as a default version of `yywrap`, a wart left over from the earliest days of lex.

When a lex scanner reached the end of `yyin`, it called `yywrap()`. The idea was that if there was another input file, `yywrap` could adjust `yyin` and return 0 to resume scanning. If that was really the end of the input, it returned 1 to the scanner to say that it was done. Although subsequent versions of lex and flex have faithfully preserved `yywrap`, in 30 years I have never seen a use of `yywrap` that wouldn't be better handled by flex's other I/O management features. In practice, everyone used the default `yywrap` from the flex library, which always returns 1, or put a one-line equivalent in their programs. Modern versions of flex let you say `%option noyywrap` at the top of your scanner to tell it not to call `yywrap`, and from here on, we'll always do that.

The default main program is slightly useful for testing and for quick flex hacks, although, again, in any nontrivial flex program, you always have your own main program that at least sets up the scanner's input. Here's what the library version does:

```
int main()
{
    while (yylex() != 0) ;
    return 0;
}
```

Most flex programs now use `%option noyywrap` and provide their own main routine, so they don't need the flex library.

Reading Several Files

The real version of `wc` handles multiple files, so Example 2-2 is an improved version of our program that does so as well. For programs with simple I/O needs that read each input file from beginning to end, flex provides the routine `yyrestart(f)`, which tells the scanner to read from stdio file `f`.

Example 2-2. Word count, reading many files

```
/* fb2-2 read several files */
%option noyywrap

%{
int chars = 0;
int words = 0;
int lines = 0;

int totchars = 0;
int totwords = 0;
int totlines = 0;
%}

%%

[a-zA-Z]+  { words++; chars += strlen(yytext); }
\n        { chars++; lines++; }
.          { chars++; }

%%

main(argc, argv)
int argc;
char **argv;
{
  int i;

  if(argc < 2) { /* just read stdin */
    yylex();
    printf("%8d%8d%8d\n", lines, words, chars);
    return 0;
  }

  for(i = 1; i < argc; i++) {
    FILE *f = fopen(argv[i], "r");

    if(!f) {
      perror(argv[i]);
      return (1);
    }
    yyrestart(f);
    yylex();
    fclose(f);
    printf("%8d%8d%8d %s\n", lines, words, chars, argv[i]);
    totchars += chars; chars = 0;
    totwords += words; words = 0;
    totlines += lines; lines = 0;
  }
  if(argc > 2) /* print total if more than one file */
    printf("%8d%8d%8d total\n", totlines, totwords, totchars);
  return 0;
}
```

For each file, it opens the file, uses `yyrestart()` to make it the input to the scanner, and calls `yylex()` to scan it. Like the real `wc`, it also tracks the overall total items read and reports the overall totals if there was more than one input file.

The I/O Structure of a Flex Scanner

Basically, a flex scanner reads from an input source and optionally writes to an output sink. By default, the input and output are stdin and stdout, but as we've seen, they're often changed to something else.

Input to a Flex Scanner

In programs that include scanners, the performance of the scanner frequently determines the performance of the entire program. Early versions of lex read from `yyin` one character at a time. Since then, flex has developed a flexible (perhaps overly so) three-level input system that allows programmers to customize it at each level to handle any imaginable input structure.

In most cases, a flex scanner reads its input using stdio from a file or the standard input, which can be the user console. There's a subtle but important difference between reading from a file and reading from the console—readahead. If the scanner is reading from a file, it should read big chunks to be as fast as possible. But if it's reading from the console, the user is probably typing one line at a time and will expect the scanner to process each line as soon as it's typed. In this case, speed doesn't matter, since a very slow scanner is still a lot faster than a fast typist, so it reads one character at a time. Fortunately, a flex scanner checks to see whether its input is from the terminal and generally does the right thing automatically.[5]

To handle its input, a flex scanner uses a structure known as a `YY_BUFFER_STATE`, which describes a single input source. It contains a string buffer and a bunch of variables and flags. Usually it contains a `FILE*` for the file it's reading from, but it's also possible to create a `YY_BUFFER_STATE` not connected a file to scan a string already in memory.

The default input behavior of a flex scanner is approximately this:

```
YY_BUFFER_STATE bp;
extern FILE* yyin;

 ... whatever the program does before the first call to the scanner

if(!yyin) yyin = stdin;  default input is stdin
bp = yy_create_buffer(yyin,YY_BUF_SIZE );
          YY_BUF_SIZE defined by flex, typically 16K
yy_switch_to_buffer(bp);   tell it to use the buffer we just made

yylex();  or yyparse() or whatever calls the scanner
```

If `yyin` isn't already set, set it to `stdin`. Then use `yy_create_buffer` to create a new buffer reading from `yyin`, use `yy_switch_to_buffer` to tell the scanner to read from it, then scan.

When reading from several files in sequence, after opening each file, call `yyrestart(fp)` to switch the scanner input to the stdio file `fp`. For the common case where a new file is assigned to `yyin`, `YY_NEW_FILE` is equivalent to `yyrestart(yyin)`.[6]

There are several other functions to create scanner buffers, including `yy_scan_string("string")` to scan a null-terminated string and `yy_scan_buffer(char *base, size)` to scan a buffer of known size. Later in this chapter we'll also see functions to maintain a stack of buffers, which is handy when handling nested include files. They're all listed in Chapter 5, under "Input Management."

Finally, for maximum flexibility, you can redefine the macro that flex uses to read input into the current buffer:

```
#define YY_INPUT(buf,result,max_size) ...
```

Whenever the scanner's input buffer is empty, it invokes `YY_INPUT`, where `buf` and `maxsize` are the buffer and its size, respectively, and `result` is where to put the actual amount read or zero at EOF. (Since this is a macro, it's `result`, not `*result`.) The ability to redefine `YY_INPUT` predates the addition of `YY_BUFFER_STATE`, so most of what people used to do with the former is now better done with the latter. At this point, the main use for a custom `YY_INPUT` is in event-driven systems where the input arrives from something that can't be preloaded into a string buffer and that stdio can't handle.

Whenever the scanner reaches the end of an input file, it matches the pseudopattern `<<EOF>>`, which is often used to clean up, switch to other files, and so forth.

Flex offers two macros that can be useful in action code, `input()` and `unput()`. Each call to `input()` returns the next character from the input stream. It's sometimes a convenient way to read through a little input without having to write patterns to match it. Each call to `unput(c)` pushes character `c` back into the input stream. It's an alternative to the `/` operator to peek ahead into the input but not to process it.

To summarize, the three levels of input management are:

- Setting `yyin` to read the desired file(s)
- Creating and using `YY_BUFFER_STATE` input buffers
- Redefining `YY_INPUT`

Flex Scanner Output

Scanner output management is much simpler than input management and is completely optional. Again, harking back to the earliest versions of lex, unless you tell it otherwise, flex acts as though there is a default rule at the end of the scanner that copies otherwise unmatched input to `yyout`.

```
.    ECHO;

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

This is of some use in flex programs that do something with part of the input and leave the rest untouched, as in the English to American translator in Example 1-2, but in general it is more likely to be a source of bugs than to be useful. Flex lets you say `%option nodefault` at the top of the scanner to tell it not to add a default rule and rather to report an error if the input rules don't cover all possible input. I recommend that scanners always use `nodefault` and include their own default rule if one is needed.

Start States and Nested Input Files

We'll try out our knowledge of flex I/O with a simple program that handles nested include files and prints them out. To make it a little more interesting, it prints the input files with the line number of each line in its file. To do that, the program keeps a stack of nested input files and line numbers, pushing an entry each time it encounters a `#include` and popping an entry off the stack when it gets to the end of a file.

We also use a very powerful flex feature called *start states* that let us control which patterns can be matched when. The `%x` line near the top of the file defines `IFILE` as a start state that we'll use when we're looking for the filename in a `#include` statement. At any point, the scanner is in one start state and can match patterns active in that start state only. In effect, the state defines a different scanner, with its own rules.

You can define as many start states as needed, but in this program we need only one in addition to the `INITIAL` state that flex always defines. Patterns are tagged with start state names in angle brackets to indicate in which state(s) the pattern is active. The `%x` marks `IFILE` as an `exclusive` start state, which means that when that state is active, only patterns specifically marked with the state can match. (There are also *inclusive* start states declared with `%s`, in which patterns not marked with any state can also match. Exclusive states are usually more useful.) In action code, the macro `BEGIN` switches to a different start state. Example 2-3 shows a code skeleton for a scanner that handles included files.

Example 2-3. Skeleton for include files

```
/* fb2-3 skeleton for include files */
%option noyywrap yylineno
%x IFILE

%{
  struct bufstack {
    struct bufstack *prev;      /* previous entry */
    YY_BUFFER_STATE bs;         /* saved buffer */
    int lineno;                 /* saved line number */
    char *filename;             /* name of this file */
    FILE *f;                    /* current file */
  } *curbs = 0;

  char *curfilename;            /* name of current input file */

  int newfile(char *fn);
  int popfile(void);
%}
%%
   match #include statement up through the quote or <
^"#"[ \t]*include[ \t]*[\"<]  { BEGIN IFILE; }

  handle filename up to the closing quote, >, or end of line
<IFILE>[^ \t\n\">]+        {
                             { int c;
                               while((c = input()) && c != '\n') ;
                             }
                             yylineno++;
                             if(!newfile(yytext))
                               yyterminate(); /* no such file */
                             BEGIN INITIAL;
                           }

  handle bad input in IFILE state
<IFILE>.|\n                { fprintf(stderr, "%4d bad include line\n", yylinen
                                     yyterminate();
                           }

   pop the file stack at end of file, terminate if it's the outermost file
<<EOF>>                    { if(!popfile()) yyterminate(); }

   print the line number at the beginning of each line
   and bump the line number each time a \n is read
^.                         { fprintf(yyout, "%4d %s", yylineno, yytext); }
^\n                        { fprintf(yyout, "%4d %s", yylineno++, yytext); }
\n                         { ECHO; yylineno++; }
.                          { ECHO; }

%%

main(int argc, char **argv)
{
  if(argc < 2) {
    fprintf(stderr, "need filename\n");
    return 1;
  }
  if(newfile(argv[1]))
    yylex();
}

int
  newfile(char *fn)
{
  FILE *f = fopen(fn, "r");
  struct bufstack *bs = malloc(sizeof(struct bufstack));

  /* die if no file or no room */
  if(!f) { perror(fn); return 0; }
  if(!bs) { perror("malloc"); exit(1); }

  /* remember state */
  if(curbs)curbs->lineno = yylineno;
  bs->prev = curbs;

  /* set up current entry */
  bs->bs = yy_create_buffer(f, YY_BUF_SIZE);
  bs->f = f;
  bs->filename = strdup(fn);
  yy_switch_to_buffer(bs->bs);
  curbs = bs;
  yylineno = 1;
  curfilename = bs->filename;
  return 1;
}

int
  popfile(void)
{
  struct bufstack *bs = curbs;
  struct bufstack *prevbs;

  if(!bs) return 0;

  /* get rid of current entry */
  fclose(bs->f);
  free(bs->fn);
  yy_delete_buffer(bs->bs);

  /* switch back to previous */
  prevbs = bs->prev;
  free(bs);

  if(!prevbs) return 0;

  yy_switch_to_buffer(prevbs->bs);
  curbs = prevbs;
  yylineno = curbs->lineno;
  curfilename = curbs->filename;
```

```
    return 1;
  }
```

The first part of the program defines the start state and also has the C code to declare the `bufstack` structure that will hold an entry in the list of saved input files.

In the patterns, the first pattern matches a `#include` statement up through the double quote that precedes the filename. The pattern permits optional whitespace in the usual places. It switches to `IFILE` state to read the next input filename. In `IFILE` state, the second pattern matches a filename, characters up to a closing quote, whitespace, or end-of-line. The filename is passed to `newfile` to stack the current input file and set up the next level of input, but first there's the matter of dealing with whatever remains of the `#include` line. One possibility would be to use another start state and patterns that absorb the rest of the line, but that would be tricky, since the action switches to the included file, so the start state and pattern would have to be used *after* the end of the included file. Instead, this is one of the few places where `input()` makes a scanner simpler. A short loop reads until it finds the `\n` at the end of the line or EOF. Then, when scanning returns to this file after the end of the included one, it resumes at the beginning of the next line.

Since an exclusive start state in effect defines its own mini-scanner, that scanner has to be prepared for any possible input. The next pattern deals with the case of an ill-formed `#include` line that doesn't have a filename after the double quote. It simply prints an error message and uses the macro `yyterminate()`, which immediately returns from the scanner.[7] This definition of `#include` is fairly casual and makes no effort to verify that the punctuation around the filename matches or that there isn't extra junk after the filename. It's not hard to write code to check those issues and diagnose errors, and a more polished version of this program should do so.[8]

Next is the special pattern `<<EOF>>`, which matches at the end of each input file. We call `popfile()`, defined later, to return to the previous input file. If it returns 0, meaning that was the last file, we terminate. Otherwise, the scanner will resume reading the previous file when it resumes scanning.

The last four patterns do the actual work of printing out each line with a preceding line number. Flex provides a variable called `yylineno` that is intended to track line numbers, so we might as well use it. The pattern `^.` matches any character at the beginning of a line, so the action prints the current line number and the character. Since a dot doesn't match a newline, `^\n` matches a newline at the beginning of a line, that is, an empty line, so the code prints out the line number and the new line and increments the line number. A newline or other character not at the beginning of the line is just printed out with `ECHO`, incrementing the line number for a new line.

The routine `newfile(fn)` prepares to read from the file named `fn`, saving any previous input file. It does so by keeping a linked list of `bufstack` structures, each of which has a link to the previous `bufstack` along with the saved `yylineno` and filename. It opens the file; creates and switches to a flex buffer; and saves the previous open file, filename, and buffer. (In this program nothing uses the filename after the file is open, but we'll reuse this code later in this chapter in a program that does.)

The routine `popfile` undoes what `newfile` did. It closes the open file, deletes the current flex buffer, and then restores the buffer, filename, and line number from the prior stack entry. Note that it doesn't call `yyrestart()` when it restores the prior buffer; if it did, it would lose any input that had already been read into the buffer.

This is a fairly typical albeit somewhat simplistic example of code for handling include files. Although flex can handle a stack of input buffers using the routines `yypush_buffer_state` and `yypop_buffer_state`, I rarely find them to be useful since they don't handle the other information invariably associated with the stacked files.

Symbol Tables and a Concordance Generator

Nearly every flex or bison program uses a *symbol table* to keep track of the names used in the input. We'll start with a very simple program that makes a *concordance*, which is a list of the line numbers where each word in the input appears, and then we'll modify it to read C source to make a C cross-referencer.

Managing Symbol Tables

Many long and dense chapters have been written in compiler texts on the topic of symbol tables, but this (I hope) is not one of them. The symbol table for the concordance simply tracks each word and the files and line numbers of each. Example 2-4 shows the declarations part of the concordance generator.

Example 2-4. Concordance generator

```
/* fb2-4 text concordance */
%option noyywrap nodefault yylineno case-insensitive

/* the symbol table */
```

```
%{
  struct symbol {               /* a word */
    char *name;
    struct ref *reflist;
  };

  struct ref {
    struct ref *next;
    char *filename;
    int flags;
    int lineno;
  };

  /* simple symtab of fixed size */
  #define NHASH 9997
  struct symbol symtab[NHASH];

  struct symbol *lookup(char*);
  void addref(int, char*, char*,int);

  char *curfilename;            /* name of current input file */

%}
%%
```

The `%option` line has two options we haven't seen before, both of which are quite useful. The `%yylineno` option tells flex to define an integer variable called `yylineno` and to maintain the current line number in it. What that means is that every time the scanner reads a newline character, it increments `yylineno`, and if the scanner backs up over a newline (using some features we'll get to later), it decrements it. It's still up to you to initialize `yylineno` to 1 at the beginning of each file and to save and restore it if you're handling include files. Even with those limitations, it's still easier than doing line numbers by hand. (In this example, there's only a single pattern that matches `\n`, which wouldn't be hard to get right, but it's quite common to have several patterns that match, causing hard-to-track bugs when some but not all of them update the line number.)

The other new option is `case-insensitive`, which tells flex to build a scanner that treats upper- and lowercase the same. What this means is that a pattern like `abc` will match *abc*, *Abc*, *ABc*, *AbC*, and so forth. It does *not* have any effect on your input; in particular, the matched string in `yytext` is not case folded or otherwise modified.

The symbol table is just an array of `symbol` structures, each of which contains a pointer to the name (i.e., the word in the concordance) and a list of references. The references are a linked list of line numbers and pointers to the filename. We also define `curfilename`, a static pointer to the name of the current file, for use when adding references.

```
%%
 /* rules for concordance generator */
 /* skip common words */
a |
an |
and |
are |
as |
at |
be |
but |
for |
in |
is |
it |
of |
on |
or |
that |
the |
this |
to                     /* ignore */

[a-z]+(\'(s|t))?   { addref(yylineno, curfilename, yytext, 0); }
.|\n                   /* ignore everything else */
%%
```

Concordances usually don't index common short words, so the first set of patterns matches and ignores them. An action consisting solely of a vertical bar tells flex that the action for this rule is the same as the action for the next rule. The action on the last ignored word `to` does nothing, which is all we need to do to ignore a word.

The next rule is the meat of the scanner and matches a reasonable approximation of an English word. It matches a string of letters, `[a-z]+`, optionally followed by an apostrophe and either `s` or `t`, to match words such as *owner's* and *can't*. Each matched word is passed to `addref()`, described in a moment, along with the current filename and line number.

The final pattern is a catchall to match whatever the previous patterns didn't.

Note that this scanner is extremely ambiguous, but flex's rules for resolving ambiguity make it do what we want. It prefers longer matches to shorter ones, so the word *toad* will be matched by the main word pattern, not *to*. If two patterns make an exact match, it prefers the earlier one in the program, which is why we put the ignore rules first and the catchall last.

```
/* concordance main routine */
main(argc, argv)
```

```
int argc;
char **argv;
{
  int i;

  if(argc < 2) { /* just read stdin */
    curfilename = "(stdin)";
    yylineno = 1;
    yylex();
  } else
  for(i = 1; i < argc; i++) {
    FILE *f = fopen(argv[i], "r");

    if(!f) {
      perror(argv[i]);
      return (1);
    }
    curfilename = argv[i];      /* for addref */

    yyrestart(f);
    yylineno = 1;
    yylex();
    fclose(f);
  }

  printrefs();
}
```

The main routine looks a lot like the one in the word count program that reads multiple files. It opens each file in turn, uses `yyrestart` to arrange to read the file, and calls `yylex`. The additions are setting `curfilename` to the name of the file, for use when building the list of references, and setting `yylineno` to 1 for each file. (Otherwise, the line numbers would continue from one file to another, which might be desirable in some situations, but not here.) Finally, `printrefs` alphabetizes the symbol table and prints the references.

Using a Symbol Table

The code section of the scanner includes a simple symbol table routine, a routine to add a word to the symbol table, and a routine to print out the concordance after all the input is run.

This symbol table is minimal but quite functional. It contains one routine, `lookup`, which takes a string and returns the address of the table entry for that name, creating a new entry if there isn't one already. The lookup technique is known as *hashing with linear probing*. It uses a hash function to turn the string into an entry number in the table, then checks the entry, and, if it's already taken by a different symbol, scans linearly until it finds a free entry.

The hash function is also quite simple: For each character, multiply the previous hash by 9 and then `xor` the character, doing all the arithmetic as unsigned, which ignores overflows. The lookup routine computes the symbol table entry index as the hash value modulo the size of the symbol table, which was chosen as a number with no even factors, again to mix the hash bits up.

```
/* hash a symbol */
static unsigned
symhash(char *sym)
{
  unsigned int hash = 0;
  unsigned c;

  while(c = *sym++) hash = hash*9 ^ c;

  return hash;
}

struct symbol *
lookup(char* sym)
{
  struct symbol *sp = &symtab[symhash(sym)%NHASH];
  int scount = NHASH;           /* how many have we looked at */

  while(--scount >= 0) {
    if(sp->name && !strcasecmp(sp->name, sym)) return sp;

    if(!sp->name) {             /* new entry */
      sp->name = strdup(sym);
      sp->reflist = 0;
      return sp;
    }

    if(++sp >= symtab+NHASH) sp = symtab; /* try the next entry */
  }
  fputs("symbol table overflow\n", stderr);
  abort(); /* tried them all, table is full */

}
```

Note that whenever `lookup` makes a new entry, it calls `strdup` to make a copy of the string to put into the symbol table entry. Flex and bison programs often have hard-to-track string storage management bugs, because it is easy to forget that the string in `yytext` will be there only until the next token is scanned.

This simple hash function and lookup routine works pretty well. I ran a group of text files through an instrumented version of the concordance program, with 4,429 different words and a total of 70,775 lookups. The average lookup took 1.32 probes, not far from the ideal minimum of 1.0.

```
void
addref(int lineno, char *filename, char *word, int flags)
{
  struct ref *r;
  struct symbol *sp = lookup(word);

  /* don't do dups of same line and file */
  if(sp->reflist &&
     sp->reflist->lineno == lineno &&
     sp->reflist->filename == filename) return;

  r = malloc(sizeof(struct ref));
  if(!r) {fputs("out of space\n", stderr); abort(); }
  r->next = sp->reflist;
  r->filename = filename;
  r->lineno = lineno;
  r->flags = flags;
  sp->reflist = r;
}
```

Next is `addref`, the routine called from inside the scanner to add a reference to a particular word; it's implemented as a linked list of reference structures chained from the symbol. In order to make the report a little shorter, it doesn't add a reference if the symbol already has a reference to the same line number and filename. Note that in this routine, we don't make a copy of the filename, because we know that the caller handed us a string that won't change. We don't copy the word either, since `lookup` will handle that if needed. Each reference has a `flags` value that isn't used here but will be when we reuse this code in the next example.

```
/* print the references
 * sort the table alphabetically
 * then flip each entry's reflist to get it into forward order
 * and print it out
 */

/* aux function for sorting */
static int
symcompare(const void *xa, const void *xb)
{
  const struct symbol *a = xa;
  const struct symbol *b = xb;

  if(!a->name) {
    if(!b->name) return 0;      /* both empty */
    return 1;                   /* put empties at the end */
  }
  if(!b->name) return -1;
  return strcmp(a->name, b->name);
}

void
```

```
printrefs()
{
  struct symbol *sp;

  qsort(symtab, NHASH, sizeof(struct symbol), symcompare); /* sort the symbol t

  for(sp = symtab; sp->name && sp < symtab+NHASH; sp++) {
    char *prevfn = NULL;        /* last printed filename, to skip dups */

    /* reverse the list of references */
    struct ref *rp = sp->reflist;
    struct ref *rpp = 0;        /* previous ref */
    struct ref *rpn;    /* next ref */

    do {
      rpn = rp->next;
      rp->next = rpp;
      rpp = rp;
      rp = rpn;
    } while(rp);

    /* now print the word and its references */
    printf("%10s", sp->name);
    for(rp = rpp; rp; rp = rp->next) {
      if(rp->filename == prevfn) {
        printf(" %d", rp->lineno);
      } else {
        printf(" %s:%d", rp->filename, rp->lineno);
        prevfn = rp->filename;
      }
    }
    printf("\n");
  }
}
```

The final routines sort and print the symbol table. The symbol table is created in an order that depends on the hash function, which is not one that is useful to human readers, so we sort the symbol table alphabetically using the standard `qsort` function. Since the symbol table probably won't be full, the sort function puts unused symbol entries after used ones, so the sorted entries will end up at the front of the table.

Then `printrefs` runs down the table and prints out the references to each word. The references are in a linked list, but since each reference was pushed onto the front of the list, it is in reverse order. So, we make a pass over the list to flip the links and put it in forward order, and we then print it out.[9] To make the concordance somewhat more readable, we print the filename only if it's different from the one in the previous entry. We just compare the pointer to the filename, on the reasonable assumption that all entries for the same file will point to the same copy of the filename.

C Language Cross-Reference

The final example in this chapter takes all the techniques we've learned so far and uses them in one program, a fairly realistic C language cross-referencer (Example 2-5). It uses nested input files to handle `#include` statements, start states to handle includes and comments, a lexical hack to track when a mention of a symbol is a definition rather than a reference, and a symbol table to keep track of it all.

Example 2-5. C cross-referencer

```
/* fb2-5 C cross-ref */
%option noyywrap nodefault yylineno

%x COMMENT
%x IFILE

/* some complex named patterns */
/* Universal Character Name */
UCN     (\\u[0-9a-fA-F]{4}|\\U[0-9a-fA-F]{8})
/* float exponent */
EXP     ([Ee][-+]?[0-9]+)
/* integer length */
ILEN    ([Uu](L|l|LL|ll)?|(L|l|LL|ll)[Uu]?)
```

The options here are the same as for the concordance, except that there's no case folding, since C treats upper- and lowercase text differently. The two exclusive start states are `COMMENT`, used to skip text inside C comments, and `IFILE`, used in `#include`.

Next come three named patterns, for use later in the rule section. There's a style of flex programming that names every tiny subpattern, for example, `DIGIT` for `[0-9]`. I don't find that useful, but I do find it useful to name patterns that are both fairly complex and used inside other larger patterns. The first pattern matches a universal character name, which is a cumbersome way of putting non-ASCII characters into strings and identifiers. A UCN is `\u` followed by four hex digits or else `\U` followed by eight hex digits. The second pattern is for the exponent of a floating-point number, the letter `E` in upper- or lowercase, an optional sign, and a string of digits. The third pattern matches the length and type suffix on an integer constant, which is an optional `U` for unsigned or an optional `L` or `LL` for length, in either order, each in either upper- or lowercase. Each pattern is enclosed in parentheses to avoid an old lex/flex incompatibility: When flex interpolates a named pattern, it acts as though the pattern was enclosed in parens, but lex didn't, leading to some very obscure bugs.

```
/* the symbol table */
%{
    struct symbol {                 /* a variable name */
      struct ref *reflist;
      char *name;
    };

    struct ref {
      struct ref *next;
      char *filename;
      int flags;                    /* 01 - definition */
      int lineno;
    };

    /* simple symtab of fixed size */
    #define NHASH 9997
    struct symbol symtab[NHASH];

    struct symbol *lookup(char*);
    void addref(int, char*, char*, int);

    char *curfilename;              /* name of current input file */

  /* include file stack */
    struct bufstack {
      struct bufstack *prev;        /* previous entry */
      YY_BUFFER_STATE bs;           /* saved buffer */
      int lineno;                   /* saved line number in this file */
      char *filename;               /* name of this file */
      FILE *f;                      /* current file */
    } *curbs;

    int newfile(char *fn);
    int popfile(void);

    int defining;                   /* names are probably definitions */

  %}
```

The rest of the front section should look familiar. The symbol table is the same as the one in the previous example. The file stack is the same as in Example 2-3.

Last is a new variable, `defining`, which is set when a mention of a name is likely to be a definition rather than a reference.

Next comes the rules section, which is much longer than any rules section we've seen before; this one is more typical of practical flex programs. Many of the token-matching rules are long and complicated but were actually quite easy to write by transliterating the BNF descriptions in the C standard. Although flex can't handle general BNF (you need bison for that), the tokens were deliberately designed to be matched by regular expressions, so the transliterations all work.

```
%%
 /* comments */
"/*"                    { BEGIN(COMMENT); }
<COMMENT>"*/"           { BEGIN(INITIAL); }
<COMMENT>([^*]|\n)+|.
<COMMENT><<EOF>>        { printf("%s:%d: Unterminated comment\n",
                          curfilename, yylineno); return 0; }

 /* C++ comment, a common extension */
"//".*\n
```

An exclusive start state makes it easy to match C comments. The first rule starts the `COMMENT` state when it sees `/*`, and the second rule switches back to the normal `INITIAL` state on `*/`. The third rule matches everything in between. Although the complexity of patterns doesn't affect the speed of a flex scanner, it is definitely faster to match one big pattern than several little ones. So, this rule could have just been `.|\n`, but the `([^*]|\n)+` can match a long string of text at once. Note that it has to exclude `*` so that the second rule can match `*/`. The `<COMMENT><<EOF>>` rule catches and reports unterminated comments. Next is a bonus rule that matches C++-style comments, a common extension to C compilers.

PATTERNS FOR C COMMENTS

Although it's possible to match C comments with a single flex pattern, it's generally not a great idea to do so. For reference, here's the pattern:

```
/\*([^*]|\*+[^/*])*\*+/
```

It matches the two characters that begin a comment, `/\*`; then a sequence of nonstars or strings of stars followed by something other than a star or slash, `([^*]|\*+[^/*])*`; followed by at least one star and a closing slash, `\*+/`.

There are two reasons to prefer the approach with multiple patterns and a start state. One is that comments can potentially be very long, if one comments out a couple of pages of code, but a flex token is limited to the size of the input buffer, typically 16K. (This is the kind of bug that is likely to be missed in testing.) The other is that it's much easier to catch and diagnose unclosed comments. It'd be possible to use a modified version of the previous pattern to match unclosed comments, but they'd be even more likely to run afoul of the 16K limit.

```
 /* declaration keywords */
_Bool |
_Complex |
_Imaginary |
auto |
char |
const |
double |
enum |
extern |
float |
inline |
int |
long |
register |
restrict |
short |
signed |
static |
struct |
typedef |
union |
unsigned |
void |
volatile { defining = 1; }

 /* keywords */
break
case
continue
default
do
else
for
goto
if
return
sizeof
switch
while
```

Next are patterns to match all of the C keywords. The keywords that introduce a definition or declaration set the `defining` flag; the other keywords are just ignored.

Another way to handle keywords is to put them into the symbol table with a flag saying they're keywords, treat them as ordinary symbols in the scanner, and recognize them via the symbol table lookup. This is basically a space versus time trade-off. Putting them into the scanner makes the scanner bigger but recognizes the keywords without an extra lookup. On modern computers the size of the scanner tables is rarely an issue, so it is easier to put them in the scanner; even with all the keywords, the tables in this program are less than 18K bytes.

```
 /* constants */

 /* integers */
0[0-7]*{ILEN}?
[1-9][0-9]*{ILEN}?
```

```
0[Xx][0-9a-fA-F]+{ILEN}?

 /* decimal float */
([0-9]*\.[0-9]+|[0-9]+\.){EXP}?[flFL]?
[0-9]+{EXP}[flFL]?

 /* hex float */
0[Xx]([0-9a-fA-F]*\.[0-9a-fA-F]+|[0-9a-fA-F]+\.?)[Pp][-+]?[0-9]+[flFL]?
```

Next come the patterns for numbers. The syntax for C numbers is surprisingly complicated, but the named `ILEN` and `EXP` subpatterns make the rules manageable. There's one pattern for each form of integer, octal, decimal, and hex, each with an optional unsigned and/or integer prefix. (This could have been done as one larger pattern, but it seems easier to read this way, and of course it's the same speed.)

Decimal floating-point numbers are very similar to the Fortran example earlier in the chapter. The first pattern matches a number that includes a decimal point and has an optional exponent. The second matches a number that doesn't have a decimal point, in which case the exponent is mandatory to make it floating point. (Without the exponent, it'd just be an integer.)

Finally comes the hex form of a floating-point number, with a binary exponent separated by `P` rather than `E`, so it can't use the `EXP` pattern.

```
 /* char const */
\'([^'\\]|\\['"?\\abfnrtv]|\\[0-7]{1,3}|\\[Xx][0-9a-fA-F]+|{UCN})+\'

 /* string literal */
L?\"([^"\\]|\\['"?\\abfnrtv]|\\[0-7]{1,3}|\\[Xx][0-9a-fA-F]+|{UCN})*\"
```

Next come very messy patterns for character and string literals. A character literal is a single quote followed by one or more of an ordinary character other than a quote or a backslash, a single character backslash escape such as `\n`, an octal escape with up to three octal digits, a hex escape with an arbitrary number of hex digits, or a UCN, all followed by a close quote. A string literal is the same syntax, except that it's enclosed in double quotes, has an optional prefix `L` to indicate a wide string, and doesn't have to contain any characters. (Note the `+` at the end of the character constant and the `*` at the end of the string.)

```
 /* punctuators */
"{"|"<%"|";"         { defining = 0; }

"["|"]"|"("|")"|"{"|"}"|"."|"->"
"++"|"--"|"&"|"*"|"+"|"-"|"~"|"!"
"/"|"%"|"<<"|">>"|"<"|">"|"<="|">="|"=="|"!="|"^"|"|"|"&&"|"||"
"?"|":"|";"|"..."
```

```
"="|"*="|"/="|"%="|"+="|"-="|"<<="|">>="|"&="|"^="|"|="
","|"#"|"##"
"<:"|":>"|"%>"|"%:"|"%:%:"
```

C calls all of the operators and punctuation *punctuators*. For our purposes, we separately treat three that usually indicate the end of the names in a variable or function definition, and we ignore the rest.

```
 /* identifier */
([_a-zA-Z]|{UCN})([_a-zA-Z0-9]|{UCN})* {
                         addref(yylineno, curfilename, yytext, defining); }

 /* whitespace */
[ \t\n]+
 /* continued line */
\\$
```

The C syntax of an identifier is a letter, underscore, or UCN, optionally followed by more letters, underscores, UCNs, and digits. When we see an identifier, we add a reference to it to the symbol table.

Two more patterns match and ignore whitespace and a backslash at the end of the line.

```
 /* some preprocessor stuff */
"#"" "*if.*\n
"#"" "*else.*\n
"#"" "*endif.*\n
"#"" "*define.*\n
"#"" "*line.*\n

 /* recognize an include */
^"#"[ \t]*include[ \t]*[\"<] { BEGIN IFILE; }
<IFILE>[^>\"]+  {
                       { int c;
                         while((c = input()) && c != '\n') ;
                       }
                       newfile(strdup(yytext));
                       BEGIN INITIAL;
                }

<IFILE>.|\n     { fprintf(stderr, "%s:%d bad include line\n",
                          curfilename, yylineno);
                  BEGIN INITIAL;
                }

<<EOF>>         { if(!popfile()) yyterminate(); }
```

There's no totally satisfactory solution to handling preprocessor commands in a cross-referencer. One possibility would be to run the source program through the preprocessor first, but that would mean the cross-reference wouldn't see any of the symbols handled by the preprocessor, just the code it expanded to. We take a very simple approach here and just ignore all preprocessor commands other than `#include`. (A reasonable alternative would be to read whatever follows `#define` and `#if` and add that to the cross-reference.) The code to handle include files is nearly the same as the example earlier in this chapter, slightly modified to handle C's include syntax. One pattern matches `#include` up to the quote or `<` that precedes the filename, and then it switches to `IFILE` state. The next pattern collects the file name, skips the rest of the line, and switches to the new file. It's a little sloppy and doesn't try to ensure that the punctuation before and after the filename matches. The next two patterns complain if there's no filename and return to the previous file at the end of each included one.

```
 /* invalid character */
.               { printf("%s:%d: Mystery character '%s'\n",
                         curfilename, yylineno, yytext);
                }
%%
```

The final pattern is a simple `.` to catch anything that none of the previous patterns did. Since the patterns cover every token that can appear in a valid C program, this pattern shouldn't ever match.

The contents of the code section are very similar to code we've already seen. The first two routines, `symhash` and `lookup`, are identical to the versions in the previous example and aren't shown again here. The version of `addref` is the same as in the earlier include example. This version of `printrefs` is slightly different. It has a line to print a `*` for each reference that's flagged as being a definition.

```
void
printrefs()
{
  struct symbol *sp;

  qsort(symtab, NHASH, sizeof(struct symbol), symcompare); /* sort the symbol t

  for(sp = symtab; sp->name && sp < symtab+NHASH; sp++) {
    char *prevfn = NULL;        /* last printed filename, to skip dups */

    /* reverse the list of references */
    struct ref *rp = sp->reflist;
    struct ref *rpp = 0;        /* previous ref */
    struct ref *rpn;    /* next ref */

    do {
      rpn = rp->next;
      rp->next = rpp;
      rpp = rp;
      rp = rpn;
    } while(rp);

    /* now print the word and its references */
    printf("%10s", sp->name);
    for(rp = rpp; rp; rp = rp->next) {
      if(rp->filename == prevfn) {
        printf(" %d", rp->lineno);
      } else {
        printf(" %s:%d", rp->filename, rp->lineno);
        prevfn = rp->filename;
      }
      if(rp->flags & 01) printf("*");
    }
    printf("\n");
  }
}
```

The versions of `newfile` and `popfile` are the same as the ones earlier in this chapter, so they aren't repeated here. In this program, if `newfile` can't open a file, it just prints an error message, returning 1 if it opened the file and 0 if it didn't, and the include code in the rules section just goes on to the next line. This wouldn't be a good idea in a regular compiler. For a cross-referencer, it has the reasonable effect of processing include files in the same directory that are specific to the current program, while skipping library files in other directories.

Finally, the main program just calls `newfile` and, if it succeeds, `yylex` for each file.

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

  if(argc < 2) {
    fprintf(stderr, "need filename\n");
    return 1;
  }
  for(i = 1; i < argc; i++) {
    if(newfile(argv[i]))
      yylex();
  }

  printrefs();
  return 0;
}
```

This concludes our first realistically large flex program. It has a fairly complex set of patterns, has somewhat complex file I/O, and does something with the text it reads.

Exercises

1. Example 2-3 matches characters one at a time. Why doesn't it match them a line at a time with a pattern like `^.*\n`? Suggest a pattern or combination of patterns that would match larger chunks of text, keeping in mind the reason `^.*` won't work.
2. The concordance program treats upper- and lowercase text separately. Modify it to handle them together. You can do this without making extra copies of the words. In `symhash()`, use `tolower` to hash lowercase versions of the characters, and use `strcasecmp()` to compare words.
3. The symbol table routine in the concordance and cross-referencer programs uses a fixed-size symbol table and dies if it fills up. Modify the routine so it doesn't do that. The two standard techniques to allow variable-sized hash tables are *chaining* and *rehashing*. Chaining turns the hash table into a table of pointers to a list of symbol entries. Lookups run down the chain to find the symbol, and if it's not found, allocate a new entry with `malloc()` and add it to the chain. Rehashing creates an initial fixed-size symbol table, again using `malloc()`. When the symbol table fills up, create a new larger symbol table and copy all the entries into it, using the hash function to decide where each entry goes in the new table. Both techniques work, but one would make it a lot messier to produce the cross-reference. Which one? Why?

---

[4] There's one exception: If there are any `/` operators at all in a flex program, the whole scanner is slightly slower because of the added logic to handle backing up over the input that is matched but not consumed by trailing context. But in practice such scanners are usually still plenty fast.

---

[5] There's a separate issue for interactive scanners related to whether the scanning process itself needs to peek ahead at the character after the one being processed, but we'll save that for later in this chapter. Fortunately, flex generally does the right thing in that case, too.

---

[6] If the previous `yyin` was read all the way to EOF, `YY_NEW_FILE` isn't strictly necessary, but it's good to use anyway for defensive programming. This is particularly the case if a scanner or parse error might return from the scanner or parser without reading the whole file.

[7] It returns the value `YY_NULL`, defined as 0, which a bison parser interprets as the end of input.

[8] Or to put it another way, the error diagnostics are left as an exercise for the reader.

[9] This trick of building the list in the wrong order and then reversing it is a handy one that we'll see again when building parse trees. It turns out to be quite efficient, since the reversal step takes just one pass over the list and requires no extra space in each individual entry.