Chapter 5. A Reference for Flex Specifications

In this chapter we describe the syntax of flex programs, along with the various options and support functions available. POSIX lex is almost an exact subset of flex, so we note which parts of flex are extensions beyond what POSIX requires.

After the section on the structure of a lex program, the sections in this chapter are in alphabetical order by feature.

Structure of a Flex Specification

A flex program consists of three parts: the definition section, the rules section, and the user subroutines.

```
...definition section ...
%%

... rules section ...
%%
... user subroutines ...
```

The parts are separated by lines consisting of two percent signs. The first two parts are required, although a part may be empty. The third part and the preceding `%%` line may be omitted.

Definition Section

The definition section can include options, the literal block, definitions, start conditions, and translations. (There is a section on each in this reference.) Lines that start with whitespace are copied verbatim to the C file. Typically this is used to include comments enclosed in `/*` and `*/`, preceded by whitespace.

Rules Section

The rules section contains pattern lines and C code. Lines that start with whitespace, or material enclosed in `%{` and `%}`, are C code that is copied verbatim to `yylex()`. C code at the beginning of the rules section will be at the beginning of `yylex()` and can include declarations of variables used in the scanner, and code to run each time `yylex()` is called.

C code lines are copied verbatim to the generated C file. Lines at the beginning of the rules section are placed near the beginning of the generated `yylex()` function and should be declarations of variables used by code associated with the patterns and initialization code for the scanner. C code lines anywhere else should contain only comments, since it's unpredictable where in the scanner they'll be. (This is how you put comments in the rules section outside of actions.)

A line that starts with anything else is a pattern line. Pattern lines contain a pattern followed by some whitespace and C code to execute when the input matches the pattern. If the C code is more than one statement or spans multiple lines, it must be enclosed in braces (`{ }` or `%{ %}`).

When a flex scanner runs, it matches the input against the patterns in the rules section. Every time it finds a match (the matched input is called a *token*), it executes the C code associated with that pattern. If a pattern is followed by a single vertical bar, instead of C code, the pattern uses the same C code as the next pattern in the file. When an input character matches no pattern, the lexer acts as though it matched a pattern whose code is `ECHO;`, which writes a copy of the token to the output.

User Subroutines

The contents of the user subroutines section are copied verbatim by flex to the C file. This section typically includes routines called from the rules. If you redefine `yywrap()`, the new version or supporting subroutines might be here.

In a large program, it is often more convenient to put the supporting code in a separate source file to minimize the amount of material that is recompiled when you change the lex file.

BEGIN

The `BEGIN` macro switches among start states. You invoke it, usually in the action code for a pattern, as follows:

```
BEGIN statename;
```

The scanner starts in state 0 (zero), also known as `INITIAL`. All other states must be named in `%s` or `%x` lines in the definition section. (See

Start States.)

Notice that even though `BEGIN` is a macro, the macro itself doesn't take any arguments, and the state name need not be enclosed in parentheses, although it is good style to do so.

C++ Scanners

Although flex has an option to create a C++ scanner, the manual says it's experimental, and the code is buggy and doesn't work very well.[19] All is not lost for C++ programmers, though. If you generate a C lexer, it will compile with a C++ compiler, and you can tell a C++ bison parser to call a C lexer.

Context Sensitivity

Flex provides several ways to make your patterns sensitive to left and right context, that is, to the text that precedes or follows the token.

Left Context

There are three ways to handle left context: the special beginning-of-line pattern character, start states, and explicit code.

The character `^` at the beginning of a pattern tells lex to match the pattern only at the beginning of the line. The `^` doesn't match any characters; it just specifies the context.

Start states can be used to require that one token precede another:

```
%s MYSTATE
%%
first { BEGIN MYSTATE; }
. . .
<MYSTATE>second  { BEGIN 0; }
```

In this lexer, the second token is recognized only after the first token. There may be intervening tokens between the first and second.

In some cases you can fake left context sensitivity by setting flags to pass context information from one token's routine to another:

```
 %{
  int flag = 0;
```

```
    %}
    %%
    a     { flag = 1; }
    b     { flag = 2; }
    zzz   {
            switch(flag) {
             case 1:     a_zzz_token(); break;
             case 2:     b_zzz_token(); break;
             default: plain_zzz_token(); break;
            }
            flag = 0;
           }
```

Right Context

There are three ways to make token recognition depend on the text to the right of the token: the special end-of-line pattern character, the slash operator, and `yyless()` .

The `$` character at the end of a pattern makes the token match only at the end of a line, that is, immediately before a `\n` character. Like the `^` character, `$` doesn't match any characters; it just specifies context. It is exactly equivalent to `/\n` and, therefore, can't be used with trailing context.

The `/` character in a pattern lets you include explicit trailing context. For instance, the pattern `abc/de` matches the token `abc` , but only if it is immediately followed by `de` . The `/` itself matches no characters. Lex counts trailing context characters when deciding which of several patterns has the longest match, but the characters do not appear in `yytext` , nor are they counted in `yyleng` .

The `yyless()` function tells lex to “push back” part of the token that was just read. The argument to `yyless()` is the number of token characters to keep. For example:

```
abcde { yyless(3); }
```

The previous has nearly the same effect as `abc/de` does because the call to `yyless()` keeps three characters of the token and puts back the other two. The only differences are that in this case the token in `yytext` contains all five characters and `yyleng` contains five instead of three.

Definitions (Substitutions)

Definitions (or substitutions) allow you to give a name to all or part of a regular expression and refer to it by name in the rules section. This can be useful to break up complex expressions and to document what your expressions are supposed to be doing. A definition takes this form:

```
NAME   expression
```

The name can contain letters, digits, hyphens, and underscores, and it must not start with a digit.

In the rules section, patterns may include references to substitutions with the name in braces, for example, `{NAME}`. The expression corresponding to the name is substituted into the pattern as though it were enclosed in parentheses. For example:

```
DIG       [0-9]
...
%%
{DIG}+            process_integer();
{DIG}+\.{DIG}*   |
\.{DIG}+         process_real() ;
```

ECHO

In the C code associated with a pattern, the macro `ECHO` writes the token to the current output file `yyout`. It is equivalent to the following:

```
fprintf(yyout, "%s", yytext);
```

The default action in flex for input text that doesn't match any pattern is to write the text to the output, equivalent to `ECHO`. In flex, `%option nodefault` or the command-line flag `-s` or `--nodefault` makes the default action abort, which is useful in the common case that the scanner is supposed to include patterns to handle all possible input.

Input Management

Flex offers a variety of ways to manage the source of the text to be scanned. At the beginning of your program, you can assign any open stdio file to `yyin` to have the scanner read from that file. If that's not adequate, flex provides several different ways to change the input source.

Stdio File Chaining

You can tell the lexer to read from any stdio file by calling `yyrestart(file)`. Also, when a lexer built with `%option yywrap` reaches the end of the input file, it calls `yywrap()`, which can switch to a different input file. See the sections and yywrap() for more details.

Input Buffers

Flex scanners read input from an input buffer. An input buffer can be associated with a stdio file, in which case the lexer reads input from the file, or it can just be associated with a string in memory. The type `YY_BUFFER_STATE` is a pointer to a flex input buffer.

```
YY_BUFFER_STATE bp;
FILE *f;

f = fopen(..., "r");
bp = yy_create_buffer(f,YY_BUF_SIZE );  new buffer reading from f

yy_switch_to_buffer(bp);  use the buffer we just made
...
yy_flush_buffer(bp);  discard buffer contents
...
void yy_delete_buffer (bp);  free buffer
```

Call `yy_create_buffer` to make a new input buffer associated with an open stdio file. Its second argument is the size of the buffer, which should be `YY_BUF_SIZE`.

Call `yy_switch_to_buffer` to make the scanner read from a buffer. You can switch buffers as needed. The current buffer is `YY_CURRENT_BUFFER`. You can call `yy_flush_buffer` to discard whatever is in the buffer, which is occasionally useful in error recovery in interactive scanners to get back to a known state. Call `yy_delete_buffer` to free a buffer no longer in use.

Input from Strings

Normally flex reads from a file, but sometimes you want it to read from some other source, such as a string in memory.

```
bp = yy_scan_bytes(char *bytes, len);  scan a copy of bytes
bp = yy_scan_string("string");   scan a copy of null-terminated string

bp = yy_scan_buffer (char *base, yy_size_t size);  scan (size-2) bytes in place
```

The routines `yy_scan_bytes` and `yy_scan_string` create a buffer with a copy of the text to be scanned. A slightly faster routine is `yy_scan_buffer`, which scans text in place, but the last two bytes of the buffer *must* be nulls (\0), which are not scanned. The type `yy_size_t` is flex's internal type used for sizes of objects.

Once a string buffer is created, use `yy_switch_to_buffer` to tell the scanner to read from it, and use `yy_delete_buffer` to free the buffer and (if appropriate) the copy of the text. The scanner treats the end of the buffer as an end-of-file.

File Nesting

Many input languages have features to allow input files to include other files, such as `#include` in C. Flex provides a pair of functions to manage a stack of input buffers:

```
void yypush_buffer_state(bp);  switch to bp, stack old buf

void yypop_buffer_state();  delete current buffer, return to previous
```

In practice, these functions are inadequate for any but the simplest input nesting, since they don't maintain any auxiliary information such as the line number or name of the current file. Maintaining your own stack of input files is not hard. See Example 2-3 for sample code.

Also helpful to maintain an input stack is the special token pattern `<<EOF>>`, which matches at the end of a file after the call to `yywrap()`.

input()

The `input()` function conceptually provides characters to the lexer. When the lexer matches characters, it conceptually calls `input()` to fetch each character. Flex bypasses `input()` for performance reasons, but the effect is the same.

The most likely place to call `input()` is in an action routine to do something special with the text that follows a particular token. For example, here is a way to handle C comments:

```
"/*" {   int c1 = 0, c2 = input ();

         for(;;) {
               if(c2 == EOF)
                     break;
```

```
                if(c1 == '*' && c2 == '/')
                        break;
                c1 = c2;
                c2 = input();
        }
    }
```

The calls to `input()` process the characters until either end-of-file or the characters `*/` occur. This approach is an alternative to exclusive start states (see Start States) to handle C-style comments. It is the best way to handle very long quoted strings and other tokens that might be too long for flex to buffer itself in its typical 16K input buffer.

If you use a C++ compiler, `input` is called `yyinput` instead to avoid name collisions with C++ libraries.

YY_INPUT

Flex scanners read input into a buffer using the macro `YY_INPUT(buf,result, max_size)`. Whenever the scanner needs more input and the 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`.) When the buffer is first set up, it calls `isatty()` to see whether the input source is the console and, if so, reads one character at a time rather than large chunks.

The main situation where redefining `YY_INPUT` is useful is when reading from an input source that is neither a string nor a stdio file.

Flex Library

Flex comes with a small library of helpful routines. You can link in the library by giving the `-lfl` flag at the end of the `cc` command line on Unix systems, or the equivalent on other systems. It contains versions of `main()` and `yywrap()`.

Flex comes with a minimal `main` program, which can be useful for quickie programs and for testing, and comes with a stub `yywrap`. They're so simple we reproduce them here:

```
main(int ac,  char **av)
{
        return yylex();
}
```

```
int yywrap() { return 1; }
```

Interactive and Batch Scanners

A flex scanner sometimes needs to look ahead one character in the input to see whether the current token is done. (Think of a number that is a string of digits.) It turns out that the scanner runs a little faster if it always looks ahead even when it doesn't need to, but that would cause very unpleasant results when a scanner is reading directly from the console. For example, if the current token is a newline, `\n`, and it reads ahead, the scanner will wait until you type another new line before going ahead.

To minimize the unpleasantness, the scanner can run either in batch mode, where it always looks ahead, or in interactive mode, where it looks ahead only when it needs to do so (which is slightly slower). If you use the standard input routines, the scanner will check to see whether the input source is a terminal using `isatty()` and, if so, switch to interactive mode. You can use `%option batch` or `%option interactive` to force it always to use one or the other mode. Forcing batch mode can make sense if you know that your scanner will never need to be interactive, for example, if you read the input yourself or if it always reads from a file.

Line Numbers and yylineno

If you keep track of the line number in the input file, you can report it in error messages. If you set `%option yylineno`, flex defines `yylineno` to contain the current line number and automatically updates it each time it reads a `\n` character. The lexer does not initialize `yylineno`, so you need to set it to 1 each time you start reading a file. Lexers that handle nested include files have to save and restore the line number associated with each file if they want to track line numbers per file.

Literal Block

A literal block in the definition section is C code bracketed by the lines `%{` and `%}`.

```
%{
... C code and declarations...
%}
```

The contents of each literal block are copied verbatim to the generated C source file. Literal blocks in the definition section are copied before the beginning of `yylex()`. The literal block usually contains declarations of variables and functions used by code in the rules section, as well as `#include` lines for header files.

If a literal block starts with `%top{` rather than `%{`, it's copied near the front of the generated program, typically for `#include` files or `#define` lines to set `YY_BUF_SIZE`.

A literal block at the beginning of the rules section is copied near the beginning of `yylex()` after the declarations of local variables, so it can contain more declarations and setup code. A literal block elsewhere in the rules section is copied to an unspecified place in `yylex`, so it should contain only comments.

See also YY_USER_ACTION.

Multiple Lexers in One Program

You may want to have lexers for two partially or entirely different token syntaxes in the same program. For example, an interactive debugging interpreter might have one lexer for the programming language and use another for the debugger commands.

There are two basic approaches to handling two lexers in one program: combine them into a single lexer or put two complete lexers into the program.

Combined Lexers

You can combine two lexers into one by using start states. All of the patterns for each lexer are prefixed by a unique set of start states. When the lexer starts, you need a little code to put the lexer into the appropriate initial state for the particular lexer in use, for example, the following code (which will be copied at the front of `yylex()`):

```
%s INITA INITB INITC
%%
%{
        extern first_tok, first_lex;

        if(first_lex) {
                BEGIN first_lex;
                first_lex = 0;
        }
        if(first_tok) {
                int holdtok = first_tok;
                first_tok = 0;
                return holdtok;
        }
    %}
```

In this case, before you call the lexer, you set `first_lex` to the initial state for the lexer. You will usually use a combined lexer in conjunction with a combined yacc parser, so you'll also usually have code to force an initial token to tell the parser which grammar to use. See Variant and Multiple Grammars.

The advantages of this approach are that the object code is somewhat smaller, since there is only one copy of the lexer code, and the different rule sets can share rules. The disadvantages are that you have to be careful to use the correct start states everywhere, you cannot have both lexers active at once (i.e., you can't call `yylex()` recursively unless you use the reentrant lexer option), and it is difficult to use different input sources for the different lexers.

Multiple Lexers

The other approach is to include two complete lexers in your program. The trick is to change the names that lex uses for its functions and variables so the two lexers can be generated separately by flex and then compiled together into one program.

Flex provides a command-line switch and program option to change the prefix used on the names in the scanner generated by lex. For example, these options tell flex to use the prefix "foo" rather than "yy" and to put the generated scanner in `foolex.c`.

```
%option prefix="foo"
%option outfile="foolex.c"
```

You can also set options on the command line:

```
$ flex --outfile=foolex.c --prefix=foo foo.l
```

Either way, the generated scanner has entry point `foolex()`, reads from stdio file `fooin`, and so forth. Somewhat confusingly, flex will generate a set of `#define` macros at the front of the lexer that redefine the standard "yy" names to the chosen prefix. This lets you write your lexer using the standard names, but the externally visible names will all use the chosen prefix.

```
#define yy_create_buffer foo_create_buffer
#define yy_delete_buffer foo_delete_buffer
#define yy_flex_debug foo_flex_debug
#define yy_init_buffer foo_init_buffer
#define yy_flush_buffer foo_flush_buffer
#define yy_load_buffer_state foo_load_buffer_state
#define yy_switch_to_buffer foo_switch_to_buffer
#define yyin fooin
#define yyleng fooleng
#define yylex foolex
#define yylineno foolineno
#define yyout fooout
#define yyrestart foorestart
#define yytext footext
#define yywrap foowrap
#define yyalloc fooalloc
#define yyrealloc foorealloc
#define yyfree foofree
```

Options When Building a Scanner

Flex offers several hundred options when building a scanner. Most can be written as

```
%option name
```

at the front of the scanner or as `--name` on the command line. To turn an option off, precede it with `no`, as in `%option noyywrap` or `--noyywrap`. In most cases, putting the options in `%option` lines is preferable to putting them on the command line, since a scanner typically won't work if the options are wrong. For a full list of options, see the section "Index of Scanner Options" in the info documentation that comes with flex.

Portability of Flex Lexers

Flex lexers are fairly portable among C implementations. There are two levels at which you can port a lexer: the original flex specification or the C source file generated by flex.

Porting Generated C Lexers

Flex generates portable C code, and you can usually move the code to any C compiler without trouble. Be sure to use `%option noyywrap` or to include your own version of `yywrap()` to avoid needing the flex library. For portability to very old C compilers, `%option noansi-definitions` and `%option noansi-prototypes` tell flex to generate K&R procedure definitions and prototypes, respectively.

Buffer sizes

You may want to adjust the size of some buffers. Flex uses two input buffers, each by default 16K, which may be too big for some microcomputer implementations. You can define the macro `YY_BUF_SIZE` in the definition section:

```
    %{
    #define YY_BUF_SIZE 4096
    %}
```

If your lexer uses `REJECT`, it will also allocate a backup state buffer four times as large as `YY_BUF_SIZE` (eight times on 64-bit machines). Don't use `REJECT` if space is an issue.

Character sets

The knottiest portability problem involves character sets. The C code generated by every flex implementation uses character codes as indexes into tables in the lexer. If both the original and target machines use the same character code, such as ASCII, the ported lexer will work. You may have to deal with different line end conventions: Unix systems end a line with a plain `\n`, while Microsoft Windows and other systems use `\r\n`. You often can have lexers ignore `\r` and treat `\n` as the line end in either case.

When the original and target machines use different character sets, for example, ASCII and EBCDIC, the lexer won't work at all, since all of the character codes used as indexes will be wrong. Sophisticated users have sometimes been able to post-process the tables to rebuild them for other character sets, but in general the only reasonable approach is to find a version of flex that runs on the target machine or else to redefine the lexer's input routine to translate the input characters into the original character set. See Input from Strings for how to change the input routine.

Reentrant Scanners

The normal code for a flex scanner places its state information in static variables so that each call to `yylex()` resumes where the previous one left off, using the existing input buffer, input file, start state, and so forth. In some situations, it can be useful to have multiple copies of the scanner active at once, typically in threaded programs that handle multiple independent input sources. For this situation, flex provides `%option reentrant` or `--reentrant`.

```
yyscan_t scanner;

if(yylex_init(&scanner)) { printf("no scanning today\n"); abort(); }
while((yylex(scanner))
   ... do something ...;
yylex_destroy(scanner);
```

In a reentrant scanner, all of the state information about the scan in progress is kept in a `yyscan_t` variable, which is actually a pointer to the structure with all the state. You create the scanner with `yylex_init()`, passing the address of the `yyscan_t` as an argument, and it returns 0 on success or 1 if it can't allocate the structure. Then you pass the `yyscan_t` to every call to `yylex()` and finally delete it with `yylex_destroy`. Each call to `yylex_init` creates a separate scanner, and several scanners can be active at once, passing the appropriate structure to each call to `yylex()`.

In a reentrant scanner, some variables commonly used in `yylex` are redefined as macros, so you can use them the same as in an ordinary scanner. These variables are `yyin`, `yyout`, `yyextra`, `yyleng`, `yytext`, `yylineno`, `yycolumn`, and `yy_flex_debug`. The macros `BEGIN`, `YY_START`, `YYSTATE`, `yymore()`, `unput()`, and `yyless()` are also modified, so you can use them the same as in an ordinary scanner. All of the routines that create and manipulate input buffers take an additional `yyscan_t` argument, for example, `yyrestart(file, scanner)`. The other routines that take a `yyscan_t` argument are `yy_switch_to_buffer`, `yy_create_buffer`, `yy_delete_buffer`, `yy_flush_buffer`, `yypush_buffer_state`, `yypop_buffer_state`, `yy_scan_buffer`, `yy_scan_string`, and `yy_scan_bytes`.

Extra Data for Reentrant Scanners

When using a reentrant scanner, you'll often have some other per-scanner data, such as a symbol table for the names the scanner has matched. You can use `yylex_init_extra` rather than `yylex_init`, passing it a pointer to your own per-scanner data. Within `yylex()`, your pointer is available as `yyextra`. The extra data is of type `YY_EXTRA_TYPE`, which is normally `void *`, but you can `#define` it to another type if you want.

```
yyscan_t scanner;
symbol *symp;

symp = symtabinit();  make per-scanner symbol table

if(yylex_init_extra(symp, &scanner)) { printf("no scanning today\n"); abort
while((yylex(scanner))
   ... do something ...;
yylex_destroy(scanner);

... inside yylex ...
[a-z]+  { symlookup(yyextra, yylval); }
```

Access to Reentrant Scanner Data

In a normal scanner, code outside `yylex()` can refer directly to `yyin`, `yyout`, and other global variables, but in a reentrant scanner, they're part of the per-scanner data structure. Flex provides access routines to get and set the major variables.

```
YY_EXTRA_TYPE yyget_extra (yyscan_t yyscanner ); yyextra
void yyset_extra (YY_EXTRA_TYPE user_defined ,yyscan_t yyscanner );

FILE *yyget_in (yyscan_t yyscanner ); yyin
void yyset_in  (FILE * in_str ,yyscan_t yyscanner );

FILE *yyget_out (yyscan_t yyscanner ); yyout
void yyset_out  (FILE * out_str ,yyscan_t yyscanner );

int yyget_lineno (yyscan_t yyscanner );
void yyset_lineno (int line_number ,yyscan_t yyscanner );

int yyget_leng(yyscan_t yyscanner); yyleng, read only

char *yyget_text(yyscan_t yyscanner); yytext
```

These functions are all available in nonreentrant scanners, without the `yyscanner` argument, although there's little need for them since they are entirely equivalent to reading or setting the appropriate variable.

Reentrant Scanners, Nested Files, and Multiple Scanners

Reentrant scanners, nested files, and multiple scanners all address sort of the same issue, more than one scanner in the same program, but they each do very different things.

- Reentrant scanners allow you to have multiple instances simultaneously active that are applying the same set of patterns to separate input sources.
- Nested files (using `yy_switch_to_buffer` and related routines) allow you to have a single scanner that reads from one file, then another, and then perhaps returns to the first file.
- Multiple scanners allow you to apply different sets of patterns to different input sources.

You can combine all three of these as needed. For example, you can call `yy_switch_to_buffer` in a reentrant scanner to change the input source for a particular instance. You can say `%option reentrant prefix="foo"` to create a scanner that can be invoked multiple times (call `foolex_init` to get started) and can be linked into the same program with other reentrant or nonreentrant scanners.

Using Reentrant Scanners with Bison

Bison has its own option to create a reentrant parser, known as `pure-parser`, which can, with some effort, be used along with a reentrant scanner. A pure parser normally gets tokens by calling `yylex`, with a pointer to the place to put the token value, but without the `yyscan_t` value that flex needs. (Flex and bison developers don't always talk to each other.) Flex provides `%option reentrant bison-bridge`, which changes the declaration of `yylex` to be

```
   int yylex (YYSTYPE * yylval_param ,yyscan_t yyscanner);
```

and also sets `yylval` automatically from the argument. One important difference from normal scanners is that `yylval` is now a pointer to a union rather than a union, so a reference that was `yylval.member` is now `yylval->member`. See Pure Scanners and Parsers for an example of a pure bison parser calling a reentrant scanner.

Regular Expression Syntax

Flex patterns are an extended version of the regular expressions used by editors and utilities such as grep. Regular expressions are composed of normal characters, which represent themselves, and metacharacters, which have special meaning in a pattern. All characters other than those listed in the following section are regular characters. Whitespace (spaces and tabs) separate the pattern from the action and so must be quoted to include them in a pattern.

Metacharacters

The metacharacters in a flex expression include the following:

`.`
: Matches any single character except the newline character `\n`.

`[]`
: Match any one of the characters within the brackets. A range of characters is indicated with the `-` (dash), for example, `[0-9]` for any of the 10 digits. If the first character after the open bracket is a dash or close bracket, it is not interpreted as a metacharacter. If the first character is a circumflex `^`, it changes the meaning to match any character except those within the brackets. (Such a character class will match a newline unless you explicitly exclude it.) Other metacharacters have no special meaning within square brackets except that C escape sequences starting with `\` are recognized. POSIX added more special square bracket patterns for internationalization. See the last few items in this list for details.

`[a-z]{-}[jv] [a-f]{+}[0-9]`
: Set difference or union of character classes. The pattern matches characters that are in the first class, with the characters in the second class omitted (for `{-}`) or added (for `{+}`).

`*`
: Matches zero or more of the preceding expression. For example, the pattern `a.*z` matches any string that starts with *a* and ends with *z*, such as *`az`*, *`abz`* or *`alcatraz`*.

`+`
: Matches one or more occurrence of the preceding regular expression. For example, `x+` matches *`x`*, *`xxx`*, or *`xxxxxx`*, but not an empty string, and `(ab)+` matches *`ab`*, *`abab`*, *`ababab`*, and so forth.

`?`
: Matches zero or one occurrence of the preceding regular expression. For example, `-?[0-9]+` indicates a number with an optional leading unary minus.

{}

Mean different things depending on what is inside. A single number `{n}` means `n` repetitions of the preceding pattern; for example, `[A-Z]{3}` matches any three uppercase letters. If the braces contain two numbers separated by a comma, `{n,m}`, they are the minimum and maximum numbers of repetitions of the preceding pattern. For example, `A{1,3}` matches one to three occurrences of the letter *A*. If the second number is missing, it is taken to be infinite, so *{1,}* means the same as +, and *{0,}* is the same as *. If the braces contain a name, it refers to the substitution by that name.

\

If the following character is a lowercase letter, then it is a C escape sequence such as `\t` for tab. Some implementations also allow octal and hex characters in the form `\123` and `\x3f`. Otherwise, `\` quotes the following character, so `\*` matches an asterisk.

()

Group a series of regular expressions together. Each of `*`, `+`, and `[]` affects only the expression immediately to its left, and `|` normally affects everything to its left and right. Parentheses can change this; for example, `(ab|cd)?ef` matches *abef*, *cdef*, or just *ef*.

|

Matches either the preceding regular expression or the subsequent regular expression. For example, `twelve|12` matches either *twelve* or *12*.

“...”

Match everything within the quotation marks literally. Metacharacters other than `\` lose their meaning. For example, `"/*"` matches the two characters `/*`.

/

Matches the preceding regular expression but only if followed by the following regular expression. For example, `0/1` matches “0” in the string “01” but does not match anything in the strings “0” or “02”. Only one slash is permitted per pattern, and a pattern cannot contain both a slash and a trailing `$`.

^

As the first character of a regular expression, it matches the beginning of a line; it is also used for negation within square brackets. Otherwise, it’s not special.

$

As the last character of a regular expression, it matches the end of a line—otherwise not special. This has the same meaning as `/\n` at the end of an expression.

<>

A name or list of names in angle brackets at the beginning of a pattern makes that pattern apply only in the given start states.

<<EOF>>

The special pattern `<<EOF>>` matches the end of file.

(?# comment)

Perl-style expression comments.

(?a:pattern) or (?a-x:pattern)

Perl-style pattern modifiers. Interpret the pattern using modifier `a`, but without modifier `x`. The modifiers are `i` for case insensitive, `s` to match as though everything is a single line (in particular, make the `.` match a `\n` character), and `x` to ignore whitespace and C-style comments. The pattern can extend over more than one line.

```
(?i:xyz) [Xx][Yy][Zz]

(?i:x(?-i:y)z) [Xx]y[Zz]

(?s:a.b) a(.|\n)b

(?x:a /* hi */ b) ab
```

Flex also allows a limited set of POSIX character classes inside character class expressions:

```
[:alnum:] [:alpha:] [:blank:] [:cntrl:] [:digit:]
[:graph:] [:lower:] [:print:] [:punct:] [:space:]
[:upper:] [:xdigit:]
```

A character class expression stands for any character of a named type handled by the ctype macros, with the types being `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`. The class name is enclosed in square brackets and colons and must itself be inside a character class. For example, `[[:digit:]]` would be equivalent to `[0123456789]`, and `[x[:xdigit:]]`, equivalent to `[x0123456789AaBbCcDdEeFf]`.

REJECT

Usually lex separates the input into nonoverlapping tokens. But sometimes you want all occurrences of a token even if it overlaps with other tokens. The special action `REJECT` lets you do this. If an action executes `REJECT`, flex conceptually puts back the text matched by the pattern and finds the next best match for it. The example finds all occurrences of the words *pink*, *pin*, and *ink* in a file, even when they overlap:

```
...
%%
pink    { npink++; REJECT;  }
ink     { nink++; REJECT; }
pin     { npin++; REJECT; }
.      |
\n     ;    /* discard other characters */
```

If the input contains the word *pink*, all three patterns will match. Without the `REJECT` statements, only *pink* would match.

Scanners that use `REJECT` are much larger and slower than those that don't, since they need considerable extra information to allow backtracking and re-lexing.

Returning Values from yylex()

The C code executed when a pattern matches a token can contain a return statement that returns a value from `yylex()` to its caller, typically a parser generated by yacc. The next time `yylex()` is called, the scanner picks up where it left off. When a scanner matches a token of interest to the parser (e.g., a keyword, variable name, or operator), it returns to pass the token back to the parser. When it matches a token not of interest to the parser (e.g., whitespace or a comment), it does not return, and the scanner immediately proceeds to match another token.

This means that you cannot restart a lexer just by calling `yylex()`. You have to reset it into the default state using `BEGIN INITIAL` and reset the input state so that the next call to `input()` will start reading the new input. A call to `yyrestart(file)`, where `file` is a standard I/O file pointer, arranges to start reading from that file.

Start States

You can declare start states, also called *start conditions* or *start rules*, in the definition section. Start states are used to limit the scope of certain rules or to change the way the lexer treats part of the file. Start states come in two versions, inclusive declared with `%s` and exclusive declared with `%x`. For example, suppose we want to scan the following C preprocessor directive:

```
#include <somefile.h>
```

Normally, the angle brackets and the filename would be scanned as the five tokens `<`, `somefile`, `.`, `h`, and `>`, but after `#include`, they are a single filename token. You can use a start state to apply a set of rules only at certain times. Be warned that those rules that do not have start states can apply in any inclusive state! The `BEGIN` statement (see BEGIN) in an action sets the current start state. For example:

```
%s INCLMODE
%%
^"#include" { BEGIN INCLMODE; }
<INCLMODE>"<" [^>\n]+">" { ... do something with the name ...  }
<INCLMODE>\n      { BEGIN INITIAL;  /* return to normal */ }
```

You declare a start state with a `%s` or `%x` line. For example:

```
%s  PREPROC
```

The previous code creates the inclusive start state `PREPROC`. In the rules section, then, a rule that has `<PREPROC>` prepended to it will apply only in state `PREPROC`. The standard state in which lex starts is state zero, also known as `INITIAL`. The current start state can be found in `YY_START` (also called `YYSTATE` for compatibility with older versions of lex), which is useful when you have a state that wants to switch back to whatever the previous state was, for example:

```
%s A B C X
 int savevar;
%%
<A,B,C>start { save = YY_START; BEGIN X; }
<X>end       { BEGIN save;
```

Flex also has exclusive start states declared with `%x`. The difference between regular and exclusive start states is that a rule with no start state is not matched when an exclusive state is active. In practice, exclusive states are a lot more useful than regular states, and you will probably want to use them.

Exclusive start states make it easy to do things like recognize C language comments:

```
%x COMMENT
%%
"/*"             { BEGIN COMMENT;  /* switch to comment mode */ }
<COMMENT>.       |
<COMMENT>\n ;                   /* throw away comment text */
<COMMENT>"*/"    { BEGIN INITIAL;  /* return to regular mode */ }
```

This wouldn't work using regular start states since all of the regular token patterns would still be active in `COMMENT` state.

unput()

The macro `unput(c)` returns the character `c` to the input stream. Unlike the analogous stdio routine `unputc()`, you can call `unput()` several times in a row to put several characters back in the input. The limit of data "pushed back" by `unput()` varies, but it is always at least as great as the longest token the lexer recognizes.

For example, when expanding macros such as C's `#define`, you need to insert the text of the macro in place of the macro call. One way to do this is to call `unput()` to push back the text, for example:

```
... in lexer action code...
char *p = macro_contents () ;
char *q = p + strlen(p);

while(q > p)
    unput(*--q);    /* push back right-to-left */
```

yyinput() yyunput()

In C++ scanners, the macros `input()` and `unput()` are called `yyinput()` and `yyunput()` to avoid colliding with C++ library names.

yyleng

Whenever a scanner matches a token, the text of the token is stored in the null-terminated string `yytext`, and its length is in `yyleng`. The length in `yyleng` is the same as the value that would be returned by `strlen(yytext)`.

yyless()

You can call `yyless(n)` from the code associated with a rule to "push back" all but the first `n` characters of the token. This can be useful when the rule to determine the boundary between tokens is inconvenient to express as a regular expression. For example, consider a pattern to match quoted strings, but where a quotation mark within a string can be escaped with a backslash:

```
\" [^"]*\"    { /* is the char before close quote a \ ? */
                   if(yytext[yyleng-2]   ==  '\\') {
                       yyless(yyleng-1); /* return last quote */
                       yymore();         /* append next string */
                     } else {
                     ...  /* process string */
                     }
                }
```

If the quoted string ends with a backslash before the closing quotation mark, it uses `yyless()` to push back the closing quote, and it uses `yymore()` to tell lex to append the next token to this one (see yymore()). The next token will be the rest of the quoted string starting with the pushed back quote, so the entire string will end up in `yytext`.

A call to `yyless()` has the same effect as calling `unput()` with the characters to be pushed back, but `yyless()` is often much faster because it can take advantage of the fact that the characters pushed back are the same ones just fetched from the input.

Another use of `yyless()` is to reprocess a token using rules for a different start state:

```
sometoken  { BEGIN OTHER_STATE; yyless(0); }
```

`BEGIN` tells lex to use another start state, and the call to `yyless()` pushes back all of the token's characters so they can be reread using the new start state. If the new start state doesn't enable different patterns that take precedence over the current one, `yyless(0)` will cause an infinite loop in the scanner because the same token is repeatedly recognized and pushed back. (This is similar to the function of `REJECT` but is considerably faster.) Unlike `REJECT`, it will loop and match the same pattern unless you use `BEGIN` to change what patterns are active.

yylex() and YY_DECL

Normally, `yylex` is called with no arguments and interacts with the rest of the program primarily through global variables. The macro `YY_DECL` declares its calling sequence, and you can redefine it to add whatever arguments you want.

```
%{
#define YY_DECL int yylex(int *fruitp)
%}

%%

apple|orange   { (*fruitp)++; }
```

Note that there is no semicolon in `YY_DECL`, since it is expanded immediately before the open brace at the beginning of the body of `yylex`.

When using a reentrant or bison-bridge parser, you can still redefine `YY_DECL`, but you must be sure to include the parameters that the reentrant code in the lexer is expecting. In a bison-bridge scanner, the normal definition is as follows:

```
#define YY_DECL int yylex (YYSTYPE * yylval_param , yyscan_t yyscanner)
```

In a plain reentrant scanner there's no `yylval_param`.

yymore()

You can call `yymore()` from the code associated with a rule to tell lex to append the next token to this one. For example:

```
%%
hyper yymore ();
text   printf("Token is %s\n", yytext);
```

If the input string is `hypertext`, the output will be “Token is hypertext.”

Using `yymore()` is most often useful where it is inconvenient or impractical to define token boundaries with regular expressions. See yyless() for an example.

yyrestart()

You can call `yyrestart(f)` to make your scanner read from open stdio file `f`. See Stdio File Chaining.

yy_scan_string and yy_scan_buffer

These functions prepare to take the scanner's input from a string or buffer in memory. See Input from Strings.

YY_USER_ACTION

This macro is expanded just before the code for each scanner action, after `yytext` and `yyleng` are set up. Its most common use is to set bison token locations. See Chapter 8.

yywrap()

When a lexer encounters an end of file, it optionally calls the routine `yywrap()` to find out what to do next. If `yywrap()` returns 0, the scanner continues scanning, while if it returns 1, the scanner returns a zero token to report the end-of-file. If your lexer doesn't use `yywrap()` to switch files, the option `%option noyywrap` removes the calls to `yywrap()`. The special token `<<EOF>>` is usually a better way to handle end-of-file situations.

The standard version of `yywrap()` in the flex library always returns 1, but if you use `yywrap()`, you should replace it with one of your own. If `yywrap()` returns 0 to indicate that there is more input, it needs first to adjust `yyin` to point to a new file, probably using `fopen()`.

---

[19] According to the guy who wrote it.