Using `flex`

A Fast Lexical Analyzer Generator

26 May 1990—Version 2.3

This product includes software developed by the University of California, Berkeley and its contributors.

Vern Paxson

Using flex
Revision: 1.5
TEXinfo 2024-02-10.22
Original text: vern@ee.lbl.gov
Texinfo conversion:pesch@cygnus.com

Copyright (c) 1990 The Regents of the University of California. All rights reserved.

This code is derived from software contributed to Berkeley by Vern Paxson.

The United States Government has rights in this work pursuant to contract no. DE-AC03-76SF00098 between the United States Department of Energy and the University of California.

Redistribution and use in source and binary forms are permitted provided that: (1) source distributions retain this entire copyright notice and comment, and (2) distributions including binaries display the following acknowledgement: "This product includes software developed by the University of California, Berkeley and its contributors" in the documentation or other materials provided with the distribution and in all advertising materials mentioning features or use of this software. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

1 An Overview of flex, with Examples

flex is a tool for generating scanners: programs which recognize lexical patterns in text. flex reads the given input files (or its standard input if no file names are given) for a description of the scanner to generate. The description is in the form of pairs of regular expressions and C code, called *rules*. flex generates as output a C source file, lex.yy.c, which defines a routine yylex. Compile and link this file with the '-lfl' library to produce an executable. When the executable runs, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Some simple examples follow, to give you the flavor of using flex.

1.1 Text-Substitution Scanner

The following flex input specifies a scanner which, whenever it encounters the string 'username', will replace it with the user's login name:

```
%%
username    printf( "%s", getlogin() );
```

By default, any text not matched by a flex scanner is copied to the output, so the net effect of this scanner is to copy its input file to its output with each occurrence of 'username' expanded. In this input, there is just one rule. 'username' is the pattern and the printf is the action. The '%%' marks the beginning of the rules.

1.2 A Scanner to Count Lines and Characters

Here's another simple example:

```
    int num_lines = 0, num_chars = 0;

%%
\n    ++num_lines; ++num_chars;
.     ++num_chars;

%%
main()
    {
    yylex();
    printf( "# of lines = %d, # of chars = %d\n",
            num_lines, num_chars );
    }
```

This scanner counts the number of characters and the number of lines in its input (it produces no output other than the final report on the counts). The first line declares two globals, num_lines and num_chars, which are accessible both inside yylex and in the main routine declared after the second '%%'. There are two rules, one which matches a newline ('\n') and increments both the line count and the character count, and one which matches any character other than a newline (indicated by the '.' regular expression).

1.3 Simplified Pascal-like Language Scanner

A somewhat more complicated example:

```
/* scanner for a toy Pascal-like language */

%{
/* need this for the call to atof() below */
#include <math.h>
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+    {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }

{DIGIT}+"."{DIGIT}*        {
            printf( "A float: %s (%g)\n", yytext,
                    atof( yytext ) );
            }

if|then|begin|end|procedure|function        {
            printf( "A keyword: %s\n", yytext );
            }

{ID}        printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"   printf( "An operator: %s\n", yytext );

"{"[^}\n]*"}"     /* eat up one-line comments */

[ \t\n]+          /* eat up whitespace */

.           printf( "Unrecognized character: %s\n", yytext );

%%

main( argc, argv )
int argc;
char **argv;
    {
    ++argv, --argc;  /* skip over program name */
    if ( argc > 0 )
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;

    yylex();
    }
```

This is the beginnings of a simple scanner for a language like Pascal. It identifies different types of tokens and reports on what it has seen.

The details of this example are explained in the following chapters.

2 Input and Output Files

`flex`'s actions are specified by definitions (which may include embedded C code) in one or more input files. The primary output file is `lex.yy.c`. You can also use some of the command-line options to get diagnostic output (see Chapter 3 [Command-line options], page 17). This chapter gives the details of how to structure your input to define the scanner you need.

2.1 Format of the Input File

The `flex` input file consists of three sections, separated by a line with just '`%%`' in it:

```
definitions
%%
rules
%%
user code
```

The *definitions* section contains declarations of simple name definitions to simplify the scanner specification, and declarations of start conditions, which are explained in a later section.

Name definitions have the form:

```
name definition
```

The *name* is a word beginning with a letter or an underscore ('`_`') followed by zero or more letters, digits, '`_`', or '`-`' (dash). The definition is taken to begin at the first non-whitespace character following the name, and continuing to the end of the line. The definition can subsequently be referred to using '`{name}`', which will expand to '`(definition)`'. For example,

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

defines '`DIGIT`' to be a regular expression which matches a single digit, and '`ID`' to be a regular expression which matches a letter followed by zero or more letters or digits. A subsequent reference to

```
{DIGIT}+"."{DIGIT}*
```

is identical to

```
([0-9])+"."([0-9])*
```

and matches one or more digits followed by a '`.`' followed by zero or more digits.

The rules section of the `flex` input contains a series of rules of the form:

```
pattern   action
```

where the *pattern* must be unindented and the *action* must begin on the same line.

See below for a further description of patterns and actions.

Finally, the user code section is simply copied to `lex.yy.c` verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second '`%%`' in the input file may be skipped, too.

In the definitions and rules sections, any indented text or text enclosed in '%{' and '%}' is copied verbatim to the output (with the '%{}' removed). The '%{}' must appear unindented on lines by themselves.

In the rules section, any indented or '%{}' text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or '%{}' text in the rule section is still copied to the output, but its meaning is not well defined and it may well cause compile-time errors (this feature is present for POSIX compliance; see below for other such features).

In the definitions section, an unindented comment (i.e., a line beginning with '/*') is also copied verbatim to the output up to the next '*/'. Also, any line in the definitions section beginning with '#' is ignored, though this style of comment is deprecated and may go away in the future.

2.1.1 Patterns in the Input

The patterns in the input are written using an extended set of regular expressions. These are:

x	match the character '*x*'
.	any character except newline
[xyz]	a "character class"; in this case, the pattern matches either an 'x', a 'y', or a 'z'
[abj-oZ]	a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'
[^A-Z]	a "negated character class", i.e., any character but those in the class. In this case, any character *except* an uppercase letter.
[^A-Z\n]	any character *except* an uppercase letter or a newline
*r**	zero or more *r*'s, where *r* is any regular expression
r+	one or more *r*'s
r? zero or one *r*'s (that is, ``an optional *r*'')	
r{2,5}	anywhere from two to five *r*'s
r{2,}	two or more *r*'s
r{4}	exactly 4 *r*'s
{*name*}	the expansion of the *name* definition (see above)
"[xyz]\"foo"	the literal string: '[xyz]"foo'
X	if *X* is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI C interpretation of '*X*'. Otherwise, a literal '*X*' (used to escape operators such as '*')
\123	the character with octal value 123
\x2a	the character with hexadecimal value 2a

`(r)` match an *r*; parentheses are used to override precedence (see below)

`rs` the regular expression *r* followed by the regular expression *s*; called "concatenation"

`r|s` either an *r* or an *s*

`r/s` an *r* but only if it is followed by an *s*. The *s* is not part of the matched text. This type of pattern is called *trailing context*.

`^r` an *r*, but only at the beginning of a line

`r$` an *r*, but only at the end of a line. Equivalent to '`r/\n`'.

`<s>r` an *r*, but only in start condition *s* (see below for discussion of start conditions)

`<s1,s2,s3>r`
same, but in any of start conditions *s1*, *s2*, or *s3*

`<<EOF>>` an end-of-file

`<s1,s2><<EOF>>`
an end-of-file when in start condition *s1* or *s2*

The regular expressions listed above are grouped according to precedence, from highest precedence at the top to lowest at the bottom. Those grouped together have equal precedence. For example,

```
foo|bar*
```

is the same as

```
(foo)|(ba(r*))
```

since the '`*`' operator has higher precedence than concatenation, and concatenation higher than alternation ('`|`'). This pattern therefore matches either the string '`foo`' or the string '`ba`' followed by zero or more instances of '`r`'. To match '`foo`' or zero or more instances of '`bar`', use:

```
foo|(bar)*
```

and to match zero or more instances of either '`foo`' or '`bar`':

```
(foo|bar)*
```

Some notes on patterns:

- A negated character class such as the example '`[^A-Z]`' above will match a newline unless '`\n`' (or an equivalent escape sequence) is one of the characters explicitly present in the negated character class (e.g., '`[^A-Z\n]`'). This is unlike how many other regular expression tools treat negated character classes, but unfortunately the inconsistency is historically entrenched. Matching newlines means that a pattern like '`[^"]*`' can match an entire input (overflowing the scanner's input buffer) unless there's another quote in the input.
- A rule can have at most one instance of trailing context (the '`/`' operator or the '`$`' operator). The start condition, '`^`', and '`<<EOF>>`' patterns can only occur at the beginning of a pattern, and, as well as with '`/`' and '`$`', cannot be grouped inside parentheses. A '`^`' which does not occur at the beginning of a rule or a '`$`' which does not occur at the end of a rule loses its special properties and is treated as a normal character.

The following are illegal:

```
foo/bar$
<sc1>foo<sc2>bar
```

You can write the first of these instead as '`foo/bar\n`'.

In the following examples, '`$`' and '`^`' are treated as normal characters:

```
foo|(bar$)
foo|^bar
```

If what you want to specify is "either '`foo`', or '`bar`' followed by a newline" you can use the following (the special '`|`' action is explained below):

```
foo      |
bar$     /* action goes here */
```

A similar trick will work for matching "either '`foo`', or '`bar`' at the beginning of a line."

2.1.2 How the Input is Matched

When the generated scanner runs, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the `flex` input file is chosen.

Once the match is determined, the text corresponding to the match (called the *token*) is made available in the global character pointer `yytext`, and its length in the global integer `yyleng`. The action corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.

If no match is found, then the default rule is executed: the next character in the input is considered matched and copied to the standard output. Thus, the simplest legal `flex` input is:

```
%%
```

which generates a scanner that simply copies its input (one character at a time) to its output.

2.1.3 Actions

Each pattern in a rule has a corresponding action, which can be any arbitrary C statement. The pattern ends at the first non-escaped whitespace character; the remainder of the line is its action. If the action is empty, then when the pattern is matched the input token is simply discarded. For example, here is the specification for a program which deletes all occurrences of '`zap me`' from its input:

```
%%
"zap me"
```

(It will copy all other characters in the input to the output since they will be matched by the default rule.)

Here is a program which compresses multiple blanks and tabs down to a single blank, and throws away whitespace found at the end of a line:

```
%%
```

```
[ \t]+        putchar( ' ' );
[ \t]+$       /* ignore this token */
```

If the action contains a '{', then the action spans till the balancing '}' is found, and the action may cross multiple lines. `flex` knows about C strings and comments and won't be fooled by braces found within them, but also allows actions to begin with '`%{`' and will consider the action to be all the text up to the next '`%}`' (regardless of ordinary braces inside the action).

An action consisting solely of a vertical bar ('`|`') means "same as the action for the next rule." See below for an illustration.

Actions can include arbitrary C code, including return statements to return a value to whatever routine called `yylex`. Each time `yylex` is called it continues processing tokens from where it last left off until it either reaches the end of the file or executes a return. Once it reaches an end-of-file, however, then any subsequent call to `yylex` will simply immediately return, unless `yyrestart` is first called (see below).

Actions are not allowed to modify '`yytext`' or '`yyleng`'.

There are a number of special directives which can be included within an action:

`ECHO` copies `yytext` to the scanner's output.

`BEGIN` followed by the name of a start condition places the scanner in the corresponding start condition (see below).

`REJECT` directs the scanner to proceed on to the "second best" rule which matched the input (or a prefix of the input). The rule is chosen as described above in Section 2.1.2 [How the Input is Matched], page 6, and `yytext` and `yyleng` set up appropriately. It may either be one which matched as much text as the originally chosen rule but came later in the `flex` input file, or one which matched less text. For example, the following will both count the words in the input and call the routine `special` whenever '`frob`' is seen:

```
            int word_count = 0;
%%

frob        special(); REJECT;
[^ \t\n]+   ++word_count;
```

Without the `REJECT`, any '`frob`' in the input would not be counted as a word, since the scanner normally executes only one action per token. Multiple `REJECT` actions are allowed, each one finding the next best choice to the currently active rule. For example, when the following scanner scans the token '`abcd`', it will write '`abcdabcaba`' to the output:

```
%%
a        |
ab       |
abc      |
abcd     ECHO; REJECT;
.|\n     /* eat up any unmatched character */
```

(The first three rules share the fourth's action, since they use the special '`|`' action.) `REJECT` is a particularly expensive feature in terms of scanner perfor-

mance; if it is used in any of the scanner's actions, it will slow down all of the scanner's matching. Furthermore, REJECT cannot be used with the '-f' or '-F' options (see below).

Note also that unlike the other special actions, REJECT is a branch; code immediately following it in the action will not be executed.

yymore() tells the scanner that the next time it matches a rule, the corresponding token should be appended onto the current value of yytext rather than replacing it. For example, given the input 'mega-kludge' the following will write 'mega-mega-kludge' to the output:

```
%%
mega-    ECHO; yymore();
kludge   ECHO;
```

First 'mega-' is matched and echoed to the output. Then 'kludge' is matched, but the previous 'mega-' is still hanging around at the beginning of yytext so the ECHO for the 'kludge' rule will actually write 'mega-kludge'. The presence of yymore in the scanner's action entails a minor performance penalty in the scanner's matching speed.

yyless(*n*)

returns all but the first *n* characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. yytext and yyleng are adjusted appropriately (e.g., yyleng will now be equal to *n*). For example, on the input 'foobar' the following will write out 'foobarbar':

```
%%
foobar    ECHO; yyless(3);
[a-z]+    ECHO;
```

'yyless(0)' will cause the entire current input string to be scanned again. Unless you've changed how the scanner will subsequently process its input (using BEGIN, for example), this will result in an endless loop.

unput(*c*) puts the character *c* back onto the input stream. It will be the next character scanned. The following action will take the current token and cause it to be rescanned enclosed in parentheses.

```
{
int i;
unput( ')' );
for ( i = yyleng - 1; i >= 0; --i )
    unput( yytext[i] );
unput( '(' );
}
```

Note that since each unput puts the given character back at the beginning of the input stream, pushing back strings must be done back-to-front.

input() reads the next character from the input stream. For example, the following is one way to eat up C comments:

```
%%
```

```
"/*"        {
            register int c;

            for ( ; ; )
                {
                while ( (c = input()) != '*' &&
                        c != EOF )
                    ; /* eat up text of comment */

                if ( c == '*' )
                    {
                    while ( (c = input()) == '*' )
                        ;
                    if ( c == '/' )
                        break; /* found the end */
                    }

                if ( c == EOF )
                    {
                    error( "EOF in comment" );
                    break;
                    }
                }
            }
```

(Note that if the scanner is compiled using C++, then `input` is instead referred to as `yyinput`, in order to avoid a name clash with the C++ stream named `input`.)

`yyterminate()`
: can be used in lieu of a `return` statement in an action. It terminates the scanner and returns a 0 to the scanner's caller, indicating '`all done`'. Subsequent calls to the scanner will immediately return unless preceded by a call to `yyrestart` (see below). By default, `yyterminate` is also called when an end-of-file is encountered. It is a macro and may be redefined.

2.2 The Generated Scanner

The output of `flex` is the file `lex.yy.c`, which contains the scanning routine `yylex`, a number of tables used by it for matching tokens, and a number of auxiliary routines and macros. By default, `yylex` is declared as follows:

```
int yylex()
    {
    ... various definitions and the actions in here ...
    }
```

(If your environment supports function prototypes, then it will be '`int yylex( void )`'.) This definition may be changed by redefining the `YY_DECL` macro. For example, you could use:

```
#undef YY_DECL
#define YY_DECL float lexscan( a, b ) float a, b;
```

to give the scanning routine the name `lexscan`, returning a `float`, and taking two `float` values as arguments. Note that if you give arguments to the scanning routine using a

K&R-style/non-prototyped function declaration, you must terminate the definition with a semicolon (‘;’).

Whenever yylex is called, it scans tokens from the global input file yyin (which defaults to stdin). It continues until it either reaches an end-of-file (at which point it returns the value 0) or one of its actions executes a return statement. In the former case, when called again the scanner will immediately return unless yyrestart is called to point yyin at the new input file. (yyrestart takes one argument, a ‘FILE *’ pointer.) In the latter case (i.e., when an action executes a return), the scanner may then be called again and it will resume scanning where it left off.

By default (and for efficiency), the scanner uses block-reads rather than simple getc calls to read characters from yyin. You can control how it gets input by redefining the YY_INPUT macro. YY_INPUT’s calling sequence is ‘YY_INPUT(*buf*,*result*,*max_size*)’. Its action is to place up to *max_size* characters in the character array *buf* and return in the integer variable result either the number of characters read or the constant YY_NULL (0 on Unix systems) to indicate EOF. The default YY_INPUT reads from the global file-pointer yyin.

A sample redefinition of YY_INPUT (in the definitions section of the input file):

```
%{
#undef YY_INPUT
#define YY_INPUT(buf,result,max_size) \
    { \
    int c = getchar(); \
    result = (c == EOF) ? YY_NULL : (buf[0] = c, 1); \
    }
%}
```

This definition will change the input processing to occur one character at a time.

You also can add in things like keeping track of the input line number this way; but don’t expect your scanner to go very fast.

When the scanner receives an end-of-file indication from YY_INPUT, it then checks the yywrap function. If yywrap returns false (zero), then it is assumed that the function has gone ahead and set up yyin to point to another input file, and scanning continues. If it returns true (non-zero), then the scanner terminates, returning 0 to its caller.

The default yywrap always returns 1. At present, to redefine it you must first ‘#undef yywrap’, as it is currently implemented as a macro. As indicated by the hedging in the previous sentence, it may be changed to a true function in the near future.

The scanner writes its ECHO output to the yyout global (default, stdout), which may be redefined by the user simply by assigning it to some other FILE pointer.

2.3 Start Conditions

flex provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with ‘<*sc*>’ will only be active when the scanner is in the start condition named *sc*. For example,

```
<STRING>[^"]*        { /* eat up the string body ... */
             ...
```

```
                }
```

will be active only when the scanner is in the 'STRING' start condition, and

```
<INITIAL,STRING,QUOTE>\.        { /* handle an escape ... */
            ...
            }
```

will be active only when the current start condition is either 'INITIAL', 'STRING', or 'QUOTE'.

Start conditions are declared in the definitions (first) section of the input using unindented lines beginning with either '%s' or '%x' followed by a list of names. The former declares *inclusive* start conditions, the latter *exclusive* start conditions. A start condition is activated using the BEGIN action. Until the next BEGIN action is executed, rules with the given start condition will be active and rules with other start conditions will be inactive. If the start condition is inclusive, then rules with no start conditions at all will also be active. If it is exclusive, then only rules qualified with the start condition will be active. A set of rules contingent on the same exclusive start condition describe a scanner which is independent of any of the other rules in the flex input. Because of this, exclusive start conditions make it easy to specify "miniscanners" which scan portions of the input that are syntactically different from the rest (e.g., comments).

If the distinction between inclusive and exclusive start conditions is still a little vague, here's a simple example illustrating the connection between the two. The set of rules:

```
%s example
%%
<example>foo            /* do something */
```

is equivalent to

```
%x example
%%
<INITIAL,example>foo    /* do something */
```

The default rule (to ECHO any unmatched character) remains active in start conditions.

'BEGIN(0)' returns to the original state where only the rules with no start conditions are active. This state can also be referred to as the start-condition 'INITIAL', so 'BEGIN(INITIAL)' is equivalent to 'BEGIN(0)'. (The parentheses around the start condition name are not required but are considered good style.)

BEGIN actions can also be given as indented code at the beginning of the rules section. For example, the following will cause the scanner to enter the 'SPECIAL' start condition whenever yylex is called and the global variable enter_special is true:

```
            int enter_special;

%x SPECIAL
%%
            if ( enter_special )
                BEGIN(SPECIAL);

<SPECIAL>blahblahblah
... more rules follow ...
```

To illustrate the uses of start conditions, here is a scanner which provides two different interpretations of a string like '123.456'. By default this scanner will treat the string

as three tokens: the integer '123', a dot '.', and the integer '456'. But if the string is preceded earlier in the line by the string 'expect-floats' it will treat it as a single token, the floating-point number 123.456:

```
%{
#include <math.h>
%}
%s expect

%%
expect-floats        BEGIN(expect);

<expect>[0-9]+"."[0-9]+      {
             printf( "found a float, = %f\n",
                     atof( yytext ) );
             }
<expect>\n           {
             /* that's the end of the line, so
              * we need another "expect-number"
              * before we'll recognize any more
              * numbers
              */
             BEGIN(INITIAL);
             }

[0-9]+       {
             printf( "found an integer, = %d\n",
                     atoi( yytext ) );
             }

"."          printf( "found a dot\n" );
```

Here is a scanner which recognizes (and discards) C comments while maintaining a count of the current input line.

```
%x comment
%%
        int line_num = 1;

"/*"         BEGIN(comment);

<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

Note that start-conditions names are really integer values and can be stored as such. Thus, the above could be extended in the following fashion:

```
%x comment foo
%%
        int line_num = 1;
        int comment_caller;
```

```
"/*"         {
             comment_caller = INITIAL;
             BEGIN(comment);
             }

...

<foo>"/*"    {
             comment_caller = foo;
             BEGIN(comment);
             }

<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(comment_caller);
```

One can then implement a "stack" of start conditions using an array of integers. (It is likely that such stacks will become a full-fledged `flex` feature in the future.) Note, though, that start conditions do not have their own namespace; '`%s`' and '`%x`' declare names in the same fashion as `#define`.

2.4 Multiple Input Buffers

Some scanners (such as those which support "include" files) require reading from several input streams. As `flex` scanners do a large amount of buffering, one cannot control where the next input will be read from by simply writing a `YY_INPUT` which is sensitive to the scanning context. `YY_INPUT` is only called when the scanner reaches the end of its buffer, which may be a long time after scanning a statement such as an "include" which requires switching the input source.

To negotiate these sorts of problems, `flex` provides a mechanism for creating and switching between multiple input buffers. An input buffer is created by using:

```
YY_BUFFER_STATE yy_create_buffer( FILE *file, int size )
```

which takes a `FILE` pointer and a size and creates a buffer associated with the given file and large enough to hold *size* characters (when in doubt, use `YY_BUF_SIZE` for the size). It returns a `YY_BUFFER_STATE` handle, which may then be passed to other routines:

```
void yy_switch_to_buffer( YY_BUFFER_STATE new_buffer )
```

switches the scanner's input buffer so subsequent tokens will come from *new_buffer*. Note that `yy_switch_to_buffer` may be used by `yywrap` to sets things up for continued scanning, instead of opening a new file and pointing `yyin` at it.

```
void yy_delete_buffer( YY_BUFFER_STATE buffer )
```

is used to reclaim the storage associated with a buffer.

`yy_new_buffer` is an alias for `yy_create_buffer`, provided for compatibility with the C++ use of `new` and `delete` for creating and destroying dynamic objects.

Finally, the `YY_CURRENT_BUFFER` macro returns a `YY_BUFFER_STATE` handle to the current buffer.

Here is an example of using these features for writing a scanner which expands include files (the '`<<EOF>>`' feature is discussed below):

```
/* the "incl" state is used for picking up the name
```

```
 * of an include file
 */
%x incl

%{
#define MAX_INCLUDE_DEPTH 10
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];
int include_stack_ptr = 0;
%}

%%
include             BEGIN(incl);

[a-z]+              ECHO;
[^a-z\n]*\n?        ECHO;

<incl>[ \t]*      /* eat the whitespace */
<incl>[^ \t\n]+   { /* got the include file name */
        if ( include_stack_ptr >= MAX_INCLUDE_DEPTH )
            {
            fprintf( stderr, "Includes nested too deeply" );
            exit( 1 );
            }

        include_stack[include_stack_ptr++] =
            YY_CURRENT_BUFFER;

        yyin = fopen( yytext, "r" );

        if ( ! yyin )
            error( ... );

        yy_switch_to_buffer(
            yy_create_buffer( yyin, YY_BUF_SIZE ) );

        BEGIN(INITIAL);
        }

<<EOF>> {
        if ( --include_stack_ptr < 0 )
            {
            yyterminate();
            }

        else
            yy_switch_to_buffer(
                 include_stack[include_stack_ptr] );
        }
```

2.5 End-of-File Rules

The special rule '`<<EOF>>`' indicates actions which are to be taken when an end-of-file is encountered and `yywrap` returns non-zero (i.e., indicates no further files to process). The action must finish by doing one of four things:

- the special `YY_NEW_FILE` action, if `yyin` has been pointed at a new file to process;
- a return statement;

- the special `yyterminate` action;
- or switching to a new buffer using `yy_switch_to_buffer` as shown in the example above.

'`<<EOF>>`' rules may not be used with other patterns; they may only be qualified with a list of start conditions. If an unqualified '`<<EOF>>`' rule is given, it applies to all start conditions which do not already have '`<<EOF>>`' actions. To specify an '`<<EOF>>`' rule for only the initial start condition, use

```
<INITIAL><<EOF>>
```

These rules are useful for catching things like unclosed comments. An example:

```
%x quote
%%

... other rules for dealing with quotes ...

<quote><<EOF>>   {
         error( "unterminated quote" );
         yyterminate();
         }
<<EOF>>  {
         if ( *++filelist )
             {
             yyin = fopen( *filelist, "r" );
             YY_NEW_FILE;
             }
         else
            yyterminate();
         }
```

2.6 Miscellaneous Macros

The macro `YY_USER_ACTION` can be redefined to provide an action which is always executed prior to the matched rule's action. For example, it could be `#define`d to call a routine to convert `yytext` to lower-case.

The macro `YY_USER_INIT` may be redefined to provide an action which is always executed before the first scan (and before the scanner's internal initializations are done). For example, it could be used to call a routine to read in a data table or open a logging file.

In the generated scanner, the actions are all gathered in one large switch statement and separated using `YY_BREAK`, which may be redefined. By default, it is simply a `break`, to separate each rule's action from the following rule's. Redefining `YY_BREAK` allows, for example, C++ users to '`#define YY_BREAK`' to do nothing (while being very careful that every rule ends with a `break` or a `return`!) to avoid suffering from unreachable statement warnings where because a rule's action ends with `return`, the `YY_BREAK` is inaccessible.

2.7 Interfacing with Parser Generators

One of the main uses of `flex` is as a companion to parser generators like `yacc`. `yacc` parsers expect to call a routine named `yylex` to find the next input token. The routine is supposed to return the type of the next token as well as putting any associated value in the global `yylval`. To use `flex` with `yacc`, specify the '`-d`' option to `yacc` to instruct it to generate the file `y.tab.h` containing definitions of all the `%token`s appearing in the `yacc` input. Then include this file in the `flex` scanner. For example, if one of the tokens is '`TOK_NUMBER`', part of the scanner might look like:

```
%{
#include "y.tab.h"
%}

%%

[0-9]+        yylval = atoi( yytext ); return TOK_NUMBER;
```

2.8 Translation Table

In the name of POSIX compliance, `flex` supports a translation table for mapping input characters into groups. The table is specified in the first section, and its format looks like:

```
%t
1        abcd
2        ABCDEFGHIJKLMNOPQRSTUVWXYZ
52       0123456789
6        \t\ \n
%t
```

This example specifies that the characters '`a`', '`b`', '`c`', and '`d`' are to all be lumped into group #1, upper-case letters in group #2, digits in group #52, tabs, blanks, and newlines into group #6, and no other characters will appear in the patterns. The group numbers are actually disregarded by `flex`; `%t` serves, though, to lump characters together. Given the above table, for example, the pattern '`a(AA)*5`' is equivalent to '`d(ZQ)*0`'. They both say, "match any character in group #1, followed by zero or more pairs of characters from group #2, followed by a character from group #52." Thus '`%t`' provides a crude way for introducing equivalence classes into the scanner specification.

Note that the '`-i`' option (see below) coupled with the equivalence classes which `flex` automatically generates take care of virtually all the instances when one might consider using '`%t`'. But what the hell, it's there if you want it.

3 Command-line Options

You can call `flex` with the following command-line options:

`-b` Generate backtracking information to `lex.backtrack`. This is a list of scanner states which require backtracking and the input characters on which they do so. By adding rules one can remove backtracking states. If all backtracking states are eliminated and '`-f`' or '`-F`' is used, the generated scanner will run faster (see the '`-p`' flag). Only users who wish to squeeze every last cycle out of their scanners need worry about this option. (See Chapter 4 [Performance Considerations], page 21.)

`-c` is a do-nothing, deprecated option included for POSIX compliance.

Note: in previous releases of `flex`, you could use `-c` to specify table-compression options. This functionality is now given by the '`-C`' flag. To ease the the impact of this change, when `flex` encounters '`-c`', it currently issues a warning message and assumes that '`-C`' was desired instead. In the future this "promotion" of '`-c`' to '`-C`' will go away in the name of full POSIX compliance (unless the POSIX meaning is removed first).

`-d` makes the generated scanner run in debug mode. Whenever a pattern is recognized and the global `yy_flex_debug` is non-zero (which is the default), the scanner will write to `stderr` a line of the form:

```
--accepting rule at line 53 ("the matched text")
```

The line number refers to the location of the rule in the file defining the scanner (i.e., the file that was fed to `flex`). Messages are also generated when the scanner backtracks, accepts the default rule, reaches the end of its input buffer (or encounters a `NUL`; at this point, the two look the same as far as the scanner's concerned), or reaches an end-of-file.

`-f` specifies (take your pick) full table or fast scanner. No table compression is done. The result is large but fast. This option is equivalent to '`-Cf`' (see below).

`-i` instructs `flex` to generate a case-insensitive scanner. The case of letters given in the `flex` input patterns will be ignored, and tokens in the input will be matched regardless of case. The matched text given in `yytext` will have the preserved case (i.e., it will not be folded).

`-n` is another do-nothing, deprecated option included only for POSIX compliance.

`-p` generates a performance report to `stderr`. The report consists of comments regarding features of the `flex` input file which will cause a loss of performance in the resulting scanner. Note that the use of `REJECT` and variable trailing context (see Chapter 7 [Deficiencies and Bugs], page 33) entails a substantial performance penalty; use of `yymore`, the '`^`' operator, and the '`-I`' flag entail minor performance penalties.

`-s` causes the default rule (that unmatched scanner input is echoed to `stdout`) to be suppressed. If the scanner encounters input that does not match any of its rules, it aborts with an error. This option is useful for finding holes in a scanner's rule set.

`-t` instructs `flex` to write the scanner it generates to standard output instead of `lex.yy.c`.

`-v` specifies that `flex` should write to `stderr` a summary of statistics regarding the scanner it generates. Most of the statistics are meaningless to the casual `flex` user, but the first line identifies the version of `flex`, which is useful for figuring out where you stand with respect to patches and new releases, and the next two lines give the date when the scanner was created and a summary of the flags which were in effect.

`-F` specifies that the fast scanner table representation should be used. This representation is about as fast as the full table representation (‘`-f`’), and for some sets of patterns will be considerably smaller (and for others, larger). In general, if the pattern set contains both “keywords” and a catch-all, “identifier” rule, such as in the set:

```
"case"    return TOK_CASE;
"switch"  return TOK_SWITCH;
...
"default" return TOK_DEFAULT;
[a-z]+    return TOK_ID;
```

then you’re better off using the full table representation. If only the “identifier” rule is present and you then use a hash table or some such to detect the keywords, you’re better off using ‘`-F`’.

This option is equivalent to ‘`-CF`’ (see below).

`-I` instructs `flex` to generate an interactive scanner. Normally, scanners generated by `flex` always look ahead one character before deciding that a rule has been matched. At the cost of some scanning overhead, `flex` will generate a scanner which only looks ahead when needed. Such scanners are called interactive because if you want to write a scanner for an interactive system such as a command shell, you will probably want the user’s input to be terminated with a newline, and without ‘`-I`’ the user will have to type a character in addition to the newline in order to have the newline recognized. This leads to dreadful interactive performance.

If all this seems too confusing, here’s the general rule: if a human will be typing in input to your scanner, use ‘`-I`’, otherwise don’t; if you don’t care about squeezing the utmost performance from your scanner and you don’t want to make any assumptions about the input to your scanner, use ‘`-I`’.

Note: ‘`-I`’ cannot be used in conjunction with full or fast tables, i.e., the ‘`-f`’, ‘`-F`’, ‘`-Cf`’, or ‘`-CF`’ flags.

`-L` instructs `flex` not to generate `#line` directives. Without this option, `flex` peppers the generated scanner with `#line` directives so error messages in the actions will be correctly located with respect to the original `flex` input file, and not to the fairly meaningless line numbers of `lex.yy.c`. (Unfortunately `flex` does not presently generate the necessary directives to “retarget” the line numbers for those parts of `lex.yy.c` which it generated. So if there is an error in the generated code, a meaningless line number is reported.)

`-T` makes `flex` run in trace mode. It will generate a lot of messages to `stdout` concerning the form of the input and the resultant non-deterministic and deterministic finite automata. This option is mostly for use in maintaining `flex`.

`-8` instructs `flex` to generate an 8-bit scanner, i.e., one which can recognize 8-bit characters. On some sites, `flex` is installed with this option as the default. On others, the default is 7-bit characters. To see which is the case, check the verbose (‘`-v`’) output for ‘`equivalence classes created`’. If the denominator of the number shown is 128, then by default `flex` is generating 7-bit characters. If it is 256, then the default is 8-bit characters and the ‘`-8`’ flag is not required (but may be a good idea to keep the scanner specification portable). Feeding a 7-bit scanner 8-bit characters will result in infinite loops, bus errors, or other such fireworks, so when in doubt, use the flag. Note that if equivalence classes are used, 8-bit scanners take only slightly more table space than 7-bit scanners (128 bytes, to be exact); if equivalence classes are not used, however, then the tables may grow up to twice their 7-bit size.

`-C`[`efmF`] controls the degree of table compression.

‘`-Ce`’ directs `flex` to construct equivalence classes, i.e., sets of characters which have identical lexical properties (for example, if the only appearance of digits in the `flex` input is in the character class ‘`[0-9]`’ then the digits ‘`0`’, ‘`1`’, ..., ‘`9`’ will all be put in the same equivalence class). Equivalence classes usually give dramatic reductions in the final table/object file sizes (typically a factor of 2–5) and are pretty cheap performance-wise (one array look-up per character scanned).

‘`-Cf`’ specifies that the full scanner tables should be generated; `flex` will not compress the tables by taking advantages of similar transition functions for different states.

‘`-CF`’ specifies that the alternate fast scanner representation (described above under the ‘`-F`’ flag) should be used.

‘`-Cm`’ directs `flex` to construct meta-equivalence classes, which are sets of equivalence classes (or characters, if equivalence classes are not being used) that are commonly used together. Meta-equivalence classes are often a big win when using compressed tables, but they have a moderate performance impact (one or two `if` tests and one array look-up per character scanned).

A lone ‘`-C`’ specifies that the scanner tables should be compressed, but `flex` is not to use either equivalence classes nor meta-equivalence classes.

The options ‘`-Cf`’ or ‘`-CF`’ and ‘`-Cm`’ do not make sense together. There is no opportunity for meta-equivalence classes if the table is not compressed. Otherwise the options may be freely mixed.

The default setting is ‘`-Cem`’, which specifies that `flex` should generate equivalence classes and meta-equivalence classes. This setting provides the highest degree of table compression. You can trade off faster-executing scanners at the cost of larger tables with the following generally being true:

slowest and smallest
`-Cem`
`-Cm`
`-Ce`
`-C`
`-C{f,F}e`
`-C{f,F}`
fastest and largest

Note that scanners with the smallest tables are usually generated and compiled the quickest, so during development you will usually want to use the default, maximal compression.

'`-Cfe`' is often a good compromise between speed and size for production scanners.

'`-C`' options are not cumulative; whenever the flag is encountered, the previous '`-C`' settings are forgotten.

`-S`*`skeleton_file`*
overrides the default skeleton file from which `flex` constructs its scanners. You'll never need this option unless you are doing `flex` maintenance or development.

4 Performance Considerations

The main design goal of `flex` is that it generate high performance scanners. It has been optimized for dealing well with large sets of rules. Aside from the effects of table compression on scanner speed outlined above, there are a number of options/actions which degrade performance. These are, from most expensive to least:

`REJECT`

pattern sets that require backtracking
arbitrary trailing context

'`^`' beginning-of-line operator
`yymore`

with the first three all being quite expensive and the last two being quite cheap.

`REJECT` should be avoided at all costs when performance is important. It is a particularly expensive option.

Getting rid of backtracking is messy and often may be an enormous amount of work for a complicated scanner. In principle, one begins by using the '`-b`' flag to generate a `lex.backtrack` file. For example, on the input

```
%%
foo        return TOK_KEYWORD;
foobar     return TOK_KEYWORD;
```

the file looks like:

```
State #6 is non-accepting -
 associated rule line numbers:
       2       3
 out-transitions: [ o ]
 jam-transitions: EOF [ \001-n  p-\177 ]

State #8 is non-accepting -
 associated rule line numbers:
       3
 out-transitions: [ a ]
 jam-transitions: EOF [ \001-`  b-\177 ]

State #9 is non-accepting -
 associated rule line numbers:
       3
 out-transitions: [ r ]
 jam-transitions: EOF [ \001-q  s-\177 ]

Compressed tables always backtrack.
```

The first few lines tell us that there's a scanner state in which it can make a transition on an '`o`' but not on any other character, and that in that state the currently scanned text does not match any rule. The state occurs when trying to match the rules found at lines 2 and 3

in the input file. If the scanner is in that state and then reads something other than an 'o', it will have to backtrack to find a rule which is matched. With a bit of headscratching one can see that this must be the state it's in when it has seen 'fo'. When this has happened, if anything other than another 'o' is seen, the scanner will have to back up to simply match the 'f' (by the default rule).

The comment regarding State #8 indicates there's a problem when 'foob' has been scanned. Indeed, on any character other than a 'b', the scanner will have to back up to accept 'foo'. Similarly, the comment for State #9 concerns when 'fooba' has been scanned.

The final comment reminds us that there's no point going to all the trouble of removing backtracking from the rules unless we're using '-f' or '-F', since there's no performance gain doing so with compressed scanners.

The way to remove the backtracking is to add "error" rules:

```
%%
foo         return TOK_KEYWORD;
foobar      return TOK_KEYWORD;

fooba       |
foob        |
fo          {
            /* false alarm, not really a keyword */
            return TOK_ID;
            }
```

Eliminating backtracking among a list of keywords can also be done using a "catch-all" rule:

```
%%
foo         return TOK_KEYWORD;
foobar      return TOK_KEYWORD;

[a-z]+      return TOK_ID;
```

This is usually the best solution when appropriate.

Backtracking messages tend to cascade. With a complicated set of rules it's not uncommon to get hundreds of messages. If one can decipher them, though, it often only takes a dozen or so rules to eliminate the backtracking (though it's easy to make a mistake and have an error rule accidentally match a valid token. A possible future flex feature will be to automatically add rules to eliminate backtracking).

Variable trailing context (where both the leading and trailing parts do not have a fixed length) entails almost the same performance loss as REJECT (i.e., substantial). So when possible a rule like:

```
%%
mouse|rat/(cat|dog)   run();
```

is better written:

```
%%
mouse/cat|dog         run();
rat/cat|dog           run();
```

or as

```
%%
mouse|rat/cat         run();
mouse|rat/dog         run();
```

Note that here the special '`|`' action does not provide any savings, and can even make things worse (see Chapter 7 [Deficiencies and Bugs], page 33).

Another area where the user can increase a scanner's performance (and one that's easier to implement) arises from the fact that the longer the tokens matched, the faster the scanner will run. This is because with long tokens the processing of most input characters takes place in the (short) inner scanning loop, and does not often have to go through the additional work of setting up the scanning environment (e.g., `yytext`) for the action. Recall the scanner for C comments:

```
%x comment
%%
        int line_num = 1;

"/*"         BEGIN(comment);

<comment>[^*\n]*
<comment>"*"+[^*/\n]*
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

This could be sped up by writing it as:

```
%x comment
%%
        int line_num = 1;

"/*"         BEGIN(comment);

<comment>[^*\n]*
<comment>[^*\n]*\n      ++line_num;
<comment>"*"+[^*/\n]*
<comment>"*"+[^*/\n]*\n ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

Now instead of each newline requiring the processing of another action, recognizing the newlines is "distributed" over the other rules to keep the matched text as long as possible. Note that adding rules does not slow down the scanner! The speed of the scanner is independent of the number of rules or (modulo the considerations given at the beginning of this section) how complicated the rules are with regard to operators such as '`*`' and '`|`'.

A final example in speeding up a scanner: suppose you want to scan through a file containing identifiers and keywords, one per line and with no other extraneous characters, and recognize all the keywords. A natural first approach is:

```
%%
asm      |
auto     |
```

```
break    |
... etc ...
volatile |
while    /* it's a keyword */

.|\n     /* it's not a keyword */
```

To eliminate the back-tracking, introduce a catch-all rule:

```
%%
asm      |
auto     |
break    |
... etc ...
volatile |
while    /* it's a keyword */

[a-z]+   |
.|\n     /* it's not a keyword */
```

Now, if it's guaranteed that there's exactly one word per line, then we can reduce the total number of matches by a half by merging in the recognition of newlines with that of the other tokens:

```
%%
asm\n    |
auto\n   |
break\n  |
... etc ...
volatile\n |
while\n  /* it's a keyword */

[a-z]+\n |
.|\n     /* it's not a keyword */
```

One has to be careful here, as we have now reintroduced backtracking into the scanner. In particular, while we know that there will never be any characters in the input stream other than letters or newlines, `flex` can't figure this out, and it will plan for possibly needing backtracking when it has scanned a token like '`auto`' and then the next character is something other than a newline or a letter. Previously it would then just match the '`auto`' rule and be done, but now it has no '`auto`' rule, only a '`auto\n`' rule. To eliminate the possibility of backtracking, we could either duplicate all rules but without final newlines, or, since we never expect to encounter such an input and therefore don't how it's classified, we can introduce one more catch-all rule, this one which doesn't include a newline:

```
%%
asm\n    |
auto\n   |
break\n  |
... etc ...
volatile\n |
```

```
while\n  /* it's a keyword */
[a-z]+\n |
[a-z]+   |
.|\n     /* it's not a keyword */
```

Compiled with '`-Cf`', this is about as fast as one can get a `flex` scanner to go for this particular problem.

A final note: `flex` is slow when matching `NUL`'s, particularly when a token contains multiple `NUL`'s. It's best to write rules which match short amounts of text if it's anticipated that the text will often include `NUL`'s.

5 Incompatibilities with lex and POSIX

flex is a rewrite of the Unix tool lex (the two implementations do not share any code, though), with some extensions and incompatibilities, both of which are of concern to those who wish to write scanners acceptable to either implementation. At present, the POSIX lex draft is very close to the original lex implementation, so some of these incompatibilities are also in conflict with the POSIX draft. But the intent is that except as noted below, flex as it presently stands will ultimately be POSIX conformant (i.e., that those areas of conflict with the POSIX draft will be resolved in flex's favor). Please bear in mind that all the comments which follow are with regard to the POSIX draft standard of Summer 1989, and not the final document (or subsequent drafts); they are included so flex users can be aware of the standardization issues and those areas where flex may in the near future undergo changes incompatible with its current definition.

flex is fully compatible with lex with the following exceptions:

- The undocumented lex scanner internal variable yylineno is not supported. It is difficult to support this option efficiently, since it requires examining every character scanned and reexamining the characters when the scanner backs up. Things get more complicated when the end of buffer or file is reached or a NUL is scanned (since the scan must then be restarted with the proper line number count), or the user uses the yyless, unput, or REJECT actions, or the multiple input buffer functions.

  The fix is to add rules which, upon seeing a newline, increment yylineno. This is usually an easy process, though it can be a drag if some of the patterns can match multiple newlines along with other characters.

  yylineno is not part of the POSIX draft.

- The input routine is not redefinable, though it may be called to read characters following whatever has been matched by a rule. If input encounters an end-of-file the normal yywrap processing is done. A "real" end-of-file is returned by input as EOF.

  Input is instead controlled by redefining the YY_INPUT macro.

  The flex restriction that input cannot be redefined is in accordance with the POSIX draft, but YY_INPUT has not yet been accepted into the draft (and probably won't; it looks like the draft will simply not specify any way of controlling the scanner's input other than by making an initial assignment to yyin).

- flex scanners do not use stdio for input. Because of this, when writing an interactive scanner one must explicitly call fflush on the stream associated with the terminal after writing out a prompt. With lex such writes are automatically flushed since lex scanners use getchar for their input. Also, when writing interactive scanners with flex, the '-I' flag must be used.

- flex scanners are not as reentrant as lex scanners. In particular, if you have an interactive scanner and an interrupt handler which long-jumps out of the scanner, and the scanner is subsequently called again, you may get the following message:

  ```
  fatal flex scanner internal error--end of buffer missed
  ```

  To reenter the scanner, first use

  ```
  yyrestart( yyin );
  ```

- output is not supported. Output from the ECHO macro is done to the file-pointer yyout (default stdout).

  The POSIX draft mentions that an output routine exists but currently gives no details as to what it does.

- lex does not support exclusive start conditions ('%x'), though they are in the current POSIX draft.

- When definitions are expanded, flex encloses them in parentheses. With lex, the following:

```
NAME    [A-Z][A-Z0-9]*
%%
foo{NAME}?      printf( "Found it\n" );
%%
```

  will not match the string 'foo' because, when the macro is expanded, the rule is equivalent to 'foo[A-Z][A-Z0-9]*?' and the precedence is such that the '?' is associated with '[A-Z0-9]*'. With flex, the rule will be expanded to 'foo([A-Z][A-Z0-9]*)?' and so the string 'foo' will match. Note that because of this, the '^', '$', '<s>', '/', and '<<EOF>>' operators cannot be used in a flex definition.

  The POSIX draft interpretation is the same as in flex.

- To specify a character class which matches anything but a left bracket (']'), in lex one can use '[^]]' but with flex one must use '[^\]]'. The latter works with lex, too.

- The lex '%r' (generate a Ratfor scanner) option is not supported. It is not part of the POSIX draft.

- If you are providing your own yywrap routine, you must include a '#undef yywrap' in the definitions section (section 1). Note that the '#undef' will have to be enclosed in '%{}'.

  The POSIX draft specifies that yywrap is a function, and this is very unlikely to change; so flex users are warned that yywrap is likely to be changed to a function in the near future.

- After a call to unput, yytext and yyleng are undefined until the next token is matched. This is not the case with lex or the present POSIX draft.

- The precedence of the '{}' (numeric range) operator is different. lex interprets 'abc{1,3}' as "match one, two, or three occurrences of 'abc'," whereas flex interprets it as "match 'ab' followed by one, two, or three occurrences of 'c'." The latter is in agreement with the current POSIX draft.

- The precedence of the '^' operator is different. lex interprets '^foo|bar' as "match either 'foo' at the beginning of a line, or 'bar' anywhere", whereas flex interprets it as "match either 'foo' or 'bar' if they come at the beginning of a line". The latter is in agreement with the current POSIX draft.

- To refer to yytext outside of the scanner source file, the correct definition with flex is 'extern char *yytext' rather than 'extern char yytext[]'. This is contrary to the current POSIX draft but a point on which flex will not be changing, as the array representation entails a serious performance penalty. It is hoped that the POSIX draft will be amended to support the flex variety of declaration (as this is a fairly painless change to require of lex users).

- yyin is initialized by lex to be stdin; flex, on the other hand, initializes yyin to NULL and then assigns it to stdin the first time the scanner is called, providing yyin has not already been assigned to a non-NULL value. The difference is subtle, but the net effect is that with flex scanners, yyin does not have a valid value until the scanner has been called.
- The special table-size declarations such as '%a' supported by lex are not required by flex scanners; flex ignores them.
- The name FLEX_SCANNER is #define'd so scanners may be written for use with either flex or lex.

The following flex features are not included in lex or the POSIX draft standard:

yyterminate()
<<EOF>>
YY_DECL
#line directives
'%{}' around actions
yyrestart()
comments beginning with '#' (deprecated)
multiple actions on a line

This last feature refers to the fact that with flex you can put multiple actions on the same line, separated with semicolons, while with lex, the following

```
foo    handle_foo(); ++num_foos_seen;
```

is (rather surprisingly) truncated to

```
foo    handle_foo();
```

flex does not truncate the action. Actions that are not enclosed in braces are simply terminated at the end of the line.

6 Diagnostic Messages

`reject_used_but_not_detected undefined`
`yymore_used_but_not_detected undefined`
: These errors can occur at compile time. They indicate that the scanner uses `REJECT` or `yymore` but that `flex` failed to notice the fact, meaning that `flex` scanned the first two sections looking for occurrences of these actions and failed to find any, but somehow you snuck some in (via a `#include` file, for example). Make an explicit reference to the action in your `flex` input file. (Note that previously `flex` supported a `%used`/`%unused` mechanism for dealing with this problem; this feature is still supported but now deprecated, and will go away soon unless the author hears from people who can argue compellingly that they need it.)

`flex scanner jammed`
: A scanner compiled with '`-s`' has encountered an input string which wasn't matched by any of its rules.

`flex input buffer overflowed`
: A scanner rule matched a string long enough to overflow the scanner's internal input buffer (16K bytes by default—controlled by `YY_BUF_SIZE` in `flex.skel`. Note that to redefine this macro, you must first `#undefine` it).

`scanner requires -8 flag`
: Your scanner specification includes recognizing 8-bit characters and you did not specify the '`-8`' flag (and your site has not installed `flex` with '`-8`' as the default).

`fatal flex scanner internal error--end of buffer missed`
: This can occur in an scanner which is reentered after a long-jump has jumped out (or over) the scanner's activation frame. Before reentering the scanner, use:

```
yyrestart( yyin );
```

`too many %t classes!`
: You managed to put every single character into its own `%t` class. `flex` requires that at least one of the classes share characters.

7 Deficiencies and Bugs

Some trailing context patterns cannot be properly matched and generate warning messages ('`Dangerous trailing context`'). These are patterns where the ending of the first part of the rule matches the beginning of the second part, such as '`zx*/xy*`', where the '`x*`' matches the '`x`' at the beginning of the trailing context. (Note that the POSIX draft states that the text matched by such patterns is undefined.)

For some trailing context rules, parts which are actually fixed-length are not recognized as such, leading to the abovementioned performance loss. In particular, parts using '`|`' or {n} (such as '`foo{3}`') are always considered variable-length.

Combining trailing context with the special '`|`' action can result in fixed trailing context being turned into the more expensive variable trailing context. For example, this happens in the following example:

```
%%
abc      |
xyz/def
```

Use of `unput` invalidates `yytext` and `yyleng`.

Use of `unput` to push back more text than was matched can result in the pushed-back text matching a beginning-of-line ('`^`') rule even though it didn't come at the beginning of the line (though this is rare!).

Pattern-matching of `NUL`'s is substantially slower than matching other characters.

`flex` does not generate correct `#line` directives for code internal to the scanner; thus, bugs in `flex.skel` yield bogus line numbers.

Due to both buffering of input and read-ahead, you cannot intermix calls to `stdio.h` routines, such as, for example, `getchar`, with `flex` rules and expect it to work. Call `input` instead.

The total table entries listed by the '`-v`' flag excludes the number of table entries needed to determine what rule has been matched. The number of entries is equal to the number of DFA states if the scanner does not use `REJECT`, and somewhat greater than the number of states if it does.

`REJECT` cannot be used with the '`-f`' or '`-F`' options.

Some of the macros, such as `yywrap`, may in the future become functions which live in the `-lfl` library. This will doubtless break a lot of code, but may be required for POSIX compliance.

The `flex` internal algorithms need documentation.

8 Contributors to `flex`

The author of `flex` is Vern Paxson, with the help of many ideas and much inspiration from Van Jacobson. Original version by Jef Poskanzer. The fast table representation is a partial implementation of a design done by Van Jacobson. The implementation was done by Kevin Gong and Vern Paxson.

Thanks to the many `flex` beta-testers, feedbackers, and contributors, especially Casey Leedom, `benson@odi.com`, Keith Bostic, Frederic Brehm, Nick Christopher, Jason Coughlin, Scott David Daniels, Leo Eskin, Chris Faylor, Eric Goldman, Eric Hughes, Jeffrey R. Jones, Kevin B. Kenny, Ronald Lamprecht, Greg Lee, Craig Leres, Mohamed el Lozy, Jim Meyering, Marc Nozell, Esmond Pitt, Jef Poskanzer, Jim Roskind, Dave Tallman, Frank Whaley, Ken Yap, and those whose names have slipped my marginal mail-archiving skills but whose contributions are appreciated all the same.

Thanks to Keith Bostic, John Gilmore, Craig Leres, Bob Mulcahy, Rich Salz, and Richard Stallman for help with various distribution headaches.

Thanks to Esmond Pitt and Earle Horton for 8-bit character support; to Benson Margulies and Fred Burke for C++ support; to Ove Ewerlid for the basics of support for `NUL`'s; and to Eric Hughes for the basics of support for multiple buffers.

Work is being done on extending `flex` to generate scanners in which the state machine is directly represented in C code rather than tables. These scanners may well be substantially faster than those generated using '`-f`' or '`-F`'. If you are working in this area and are interested in comparing notes and seeing whether redundant work can be avoided, contact Ove Ewerlid (`ewerlid@mizar.DoCS.UU.SE`).

This work was primarily done when I was at the Real Time Systems Group at the Lawrence Berkeley Laboratory in Berkeley, CA. Many thanks to all there for the support I received.

Send comments to:

Vern Paxson
Computer Systems Engineering
Bldg. 46A, Room 1123
Lawrence Berkeley Laboratory
Berkeley, CA 94720

vern@ee.lbl.gov

Table of Contents