NAME

flex − fast lexical analyzer generator

SYNOPSIS

flex [−bcdfhilnpstvwBFILTV78+? −C[aefFmr] −ooutput −Pprefix −Sskeleton] [−−help −−version] *[filename ...]*

OVERVIEW

This manual describes *flex,* a tool for generating programs that perform pattern-matching on text. The manual includes both tutorial and reference sections:

Description
a brief overview of the tool

Some Simple Examples

Format Of The Input File

Patterns
the extended regular expressions used by flex

How The Input Is Matched
the rules for determining what has been matched

Actions
how to specify what to do when a pattern is matched

The Generated Scanner
details regarding the scanner that flex produces;
how to control the input source

Start Conditions
introducing context into your scanners, and
managing "mini-scanners"

Multiple Input Buffers
how to manipulate multiple input sources; how to
scan from strings instead of files

End-of-file Rules
special rules for matching the end of the input

Miscellaneous Macros
a summary of macros available to the actions

Values Available To The User
a summary of values available to the actions

Interfacing With Yacc
connecting flex scanners together with yacc parsers

Options
flex command-line options, and the "%option"
directive

DESCRIPTION

flex is a tool for generating *scanners:* programs which recognized lexical patterns in text. *flex* reads the given input files, or its standard input if no file names are given, for a description of a 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().** This file is compiled and linked with the **−lfl** library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

SOME SIMPLE EXAMPLES

First some simple examples to get the flavor of how one uses *flex.* 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.

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).

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 will be explained in the following sections.

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-white-space 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 %{}'s removed). The %{}'s 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 (but not in the rules section), an unindented comment (i.e., a line beginning with "/*") is also copied verbatim to the output up to the next "*/".

PATTERNS

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

```
x          match the character 'x'
.          any character (byte) 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 '*')
\0         a NUL character (ASCII code 0)
\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 text matched by s is included when determining whether this rule is the "longest match", but is then returned to the input before the action is executed. So the action only sees the text matched by r. This type of pattern is called trailing context". (There are some combinations of r/s that flex cannot match correctly; see notes in the Deficiencies / Bugs section below regarding "dangerous trailing context".)

ˆr an r, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).

r$ an r, but only at the end of a line (i.e., just before a newline). Equivalent to "r/\n".

Note that flex’s notion of "newline" is exactly whatever the C compiler used to compile flex interprets ’\n’ as; in particular, on some DOS systems you must either filter out \r’s in the input yourself, or explicitly use r/\r\n for "r$".

<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

<*>r an r in any start condition, even an exclusive one.

<<EOF>> an end-of-file

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

Note that inside of a character class, all regular expression operators lose their special meaning except escape (’\’) and the character class operators, ’-’, ’]’, and, at the beginning of the class, ’ˆ’.

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 r's. To match "foo" or zero-or-more "bar"'s, use:

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

and to match zero-or-more "foo"'s-or-"bar"'s:

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

In addition to characters and ranges of characters, character classes can also contain character class *expressions.* These are expressions enclosed inside **[:** and **:]** delimiters (which themselves must appear between the '[' and ']' of the character class; other elements may occur inside the character class, too). The valid expressions are:

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

These expressions all designate a set of characters equivalent to the corresponding standard C **isXXX** function. For example, **[:alnum:]** designates those characters for which **isalnum()** returns true - i.e., any alphabetic or numeric. Some systems don't provide **isblank(),** so flex defines **[:blank:]** as a blank or a tab.

For example, the following character classes are all equivalent:

```
[[:alnum:]]
[[:alpha:][:digit:]]
[[:alpha:]0-9]
[a-zA-Z0-9]
```

If your scanner is case-insensitive (the **−i** flag), then **[:upper:]** and **[:lower:]** are equivalent to **[:alpha:].**

Some notes on patterns:

- A negated character class such as the example "[ˆA-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., "[ˆA-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 [ˆ"]* can match the entire input unless there's another quote in the input.
- A rule can have at most one instance of trailing context (the '/' operator or the '$' operator). The start condition, 'ˆ', and "<<EOF>>" patterns can only occur at the beginning of a pattern, and, as well as with '/' and '$', cannot be grouped inside parentheses. A 'ˆ' 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
  ```

  Note that the first of these, can be written "foo/bar\n".

  The following will result in '$' or 'ˆ' being treated as a normal character:

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

If what's wanted is a "foo" or a bar-followed-by-a-newline, the following could be used (the special '|' action is explained below):

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

A similar trick will work for matching a foo or a bar-at-the-beginning-of-a-line.

HOW THE INPUT IS MATCHED

When the generated scanner is run, 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.

Note that **yytext** can be defined in two different ways: either as a character *pointer* or as a character *array.* You can control which definition *flex* uses by including one of the special directives **%pointer** or **%array** in the first (definitions) section of your flex input. The default is **%pointer,** unless you use the **-l** lex compatibility option, in which case **yytext** will be an array. The advantage of using **%pointer** is substantially faster scanning and no buffer overflow when matching very large tokens (unless you run out of dynamic memory). The disadvantage is that you are restricted in how your actions can modify **yytext** (see the next section), and calls to the **unput()** function destroys the present contents of **yytext,** which can be a considerable porting headache when moving between different *lex* versions.

The advantage of **%array** is that you can then modify **yytext** to your heart's content, and calls to **unput()** do not destroy **yytext** (see below). Furthermore, existing *lex* programs sometimes access **yytext** externally using declarations of the form:

```
extern char yytext[];
```

This definition is erroneous when used with **%pointer,** but correct for **%array.**

%array defines **yytext** to be an array of **YYLMAX** characters, which defaults to a fairly large value. You can change the size by simply #define'ing **YYLMAX** to a different value in the first section of your *flex* input. As mentioned above, with **%pointer** yytext grows dynamically to accommodate large tokens. While this means your **%pointer** scanner can accommodate very large tokens (such as matching entire blocks of comments), bear in mind that each time the scanner must resize **yytext** it also must rescan the entire token from the beginning, so matching such tokens can prove slow. **yytext** presently does *not* dynamically grow if a call to **unput()** results in too much text being pushed back; instead, a run-time error results.

Also note that you cannot use **%array** with C++ scanner classes (the **c++** option; see below).

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.

Actions are free to modify **yytext** except for lengthening it (adding characters to its end--these will overwrite later characters in the input stream). This however does not apply when using **%array** (see above); in that case, **yytext** may be freely modified in any way.

Actions are free to modify **yyleng** except they should not do so if the action also includes use of **yymore()** (see below).

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 "How the Input is Matched", 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"'s in the input would not be counted as words, since the scanner normally executes only one action per token. Multiple **REJECT's** 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 performance; 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 *-Cf* or *-CF* 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".

Two notes regarding use of **yymore().** First, **yymore()** depends on the value of *yyleng* correctly reflecting the size of the current token, so you must not modify *yyleng* if you are using **yymore().** Second, 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;
```

An argument of 0 to **yyless** 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.

Note that **yyless** is a macro and can only be used in the flex input file, not from other source files.

- **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;
/* Copy yytext because unput() trashes yytext */
char *yycopy = strdup( yytext );
```

```
unput( ')' );
for ( i = yyleng - 1; i >= 0; --i )
    unput( yycopy[i] );
unput( '(' );
free( yycopy );
}
```

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.

An important potential problem when using **unput()** is that if you are using **%pointer** (the default), a call to **unput()** *destroys* the contents of *yytext*, starting with its rightmost character and devouring one character to the left with each call. If you need the value of yytext preserved after a call to **unput()** (as in the above example), you must either first copy it elsewhere, or build your scanner using **%array** instead (see How The Input Is Matched).

Finally, note that you cannot put back **EOF** to attempt to mark the input stream with an end-of-file.

- **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 by the name of *input.)*

- **YY_FLUSH_BUFFER** flushes the scanner's internal buffer so that the next time the scanner attempts to match a token, it will first refill the buffer using **YY_INPUT** (see The Generated Scanner, below). This action is a special case of the more general **yy_flush_buffer()** function, described below in the section Multiple Input Buffers.

- **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". By default, **yyterminate()** is also called when an end-of-file is encountered. It is a macro and may be redefined.

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 defining the "YY_DECL" macro. For example, you could use:

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

to give the scanning routine the name *lexscan,* returning a float, and taking two floats 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 semi-colon (;).

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.

If the scanner reaches an end-of-file, subsequent calls are undefined unless either *yyin* is pointed at a new input file (in which case scanning continues from that file), or **yyrestart()** is called. **yyrestart()** takes one argument, a **FILE** * pointer (which can be nil, if you've set up **YY_INPUT** to scan from a source other than *yyin),* and initializes *yyin* for scanning from that file. Essentially there is no difference between just assigning *yyin* to a new input file or using **yyrestart()** to do so; the latter is available for compatibility with previous versions of *flex,* and because it can be used to switch input files in the middle of scanning. It can also be used to throw away the current input buffer, by calling it with an argument of *yyin;* but better is to use **YY_FLUSH_BUFFER** (see above). Note that **yyrestart()** does *not* reset the start condition to **INITIAL** (see Start Conditions, below).

If **yylex()** stops scanning due to executing a *return* statement in one of the actions, the scanner may then be called again and it will resume scanning where it left off.

By default (and for purposes of efficiency), the scanner uses block-reads rather than simple *getc()* calls to read characters from *yyin.* The nature of how it gets its input can be controlled by defining 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 definition of YY_INPUT (in the definitions section of the input file):

```
%{
#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.

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. Note that in either case, the start condition remains unchanged; it does *not* revert to **INITIAL.**

If you do not supply your own version of **yywrap(),** then you must either use **%option noyywrap** (in which case the scanner behaves as though **yywrap()** returned 1), or you must link with **−lfl** to obtain the default version of the routine, which always returns 1.

Three routines are available for scanning from in-memory buffers rather than files: **yy_scan_string(), yy_scan_bytes(),** and **yy_scan_buffer().** See the discussion of them below in the section Multiple Input Buffers.

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.

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 "mini-scanners" 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();

bar            something_else();
```

is equivalent to

```
%x example
```

```
%%

<example>foo   do_something();

<INITIAL,example>bar    something_else();
```

Without the <**INITIAL,example**> qualifier, the *bar* pattern in the second example wouldn't be active (i.e., couldn't match) when in start condition **example.** If we just used <**example**> to qualify *bar*, though, then it would only be active in **example** and not in **INITIAL,** while in the first example it's active in both, because in the first example the **example** startion condition is an *inclusive* **(%s)** start condition.

Also note that the special start-condition specifier <*> matches every start condition. Thus, the above example could also have been written;

```
%x example
%%

<example>foo   do_something();

<*>bar    something_else();
```

The default rule (to **ECHO** any unmatched character) remains active in start conditions. It is equivalent to:

```
<*>.|\n     ECHO;
```

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 it will treat it 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);
```

This scanner goes to a bit of trouble to match as much text as possible with each rule. In general, when attempting to write a high-speed scanner try to match as much possible in each rule, as it's a big win.

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);
```

Furthermore, you can access the current start condition using the integer-valued **YY_START** macro. For example, the above assignments to *comment_caller* could instead be written

```
comment_caller = YY_START;
```

Flex provides **YYSTATE** as an alias for **YY_START** (since that is what's used by AT&T *lex).*

Note that start conditions do not have their own name-space; %s's and %x's declare names in the same fashion as #define's.

Finally, here's an example of how to match C-style quoted strings using exclusive start conditions, including expanded escape sequences (but not including checking for a string that's too long):

```
%x str

%%
        char string_buf[MAX_STR_CONST];
        char *string_buf_ptr;


\"      string_buf_ptr = string_buf; BEGIN(str);

<str>\"        { /* saw closing quote - all done */
        BEGIN(INITIAL);
        *string_buf_ptr = '\0';
        /* return string constant token type and
         * value to parser
         */
        }

<str>\n        {
        /* error - unterminated string constant */
        /* generate error message */
        }

<str>\\[0-7]{1,3} {
        /* octal escape sequence */
        int result;

        (void) sscanf( yytext + 1, "%o", &result );
```

```
            if ( result > 0xff )
                    /* error, constant is out-of-bounds */

            *string_buf_ptr++ = result;
            }

    <str>\\[0-9]+ {
            /* generate error - bad escape sequence; something
             * like '\48' or '\0777777'
             */
            }

    <str>\\n  *string_buf_ptr++ = '\n';
    <str>\\t  *string_buf_ptr++ = '\t';
    <str>\\r  *string_buf_ptr++ = '\r';
    <str>\\b  *string_buf_ptr++ = '\b';
    <str>\\f  *string_buf_ptr++ = '\f';

    <str>\\(.|\n)  *string_buf_ptr++ = yytext[1];

    <str>[^\\\n\"]+        {
            char *yptr = yytext;

            while ( *yptr )
                    *string_buf_ptr++ = *yptr++;
            }
```

Often, such as in some of the examples above, you wind up writing a whole bunch of rules all preceded by the same start condition(s). Flex makes this a little easier and cleaner by introducing a notion of start condition *scope.* A start condition scope is begun with:

```
    <SCs>{
```

where *SCs* is a list of one or more start conditions. Inside the start condition scope, every rule automatically has the prefix <*SCs*> applied to it, until a '}' which matches the initial '{'. So, for example,

```
    <ESC>{
        "\\n"   return '\n';
        "\\r"   return '\r';
        "\\f"   return '\f';
        "\\0"   return '\0';
    }
```

is equivalent to:

```
    <ESC>"\\n"  return '\n';
    <ESC>"\\r"  return '\r';
    <ESC>"\\f"  return '\f';
    <ESC>"\\0"  return '\0';
```

Start condition scopes may be nested.

Three routines are available for manipulating stacks of start conditions:

void yy_push_state(int new_state)
pushes the current start condition onto the top of the start condition stack and switches to *new_state* as though you had used **BEGIN new_state** (recall that start condition names are also integers).

void yy_pop_state()
pops the top of the stack and switches to it via **BEGIN.**

int yy_top_state()
returns the top of the stack without altering the stack's contents.

The start condition stack grows dynamically and so has no built-in size limitation. If memory is exhausted, program execution aborts.

To use start condition stacks, your scanner must include a **%option stack** directive (see Options below).

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 (see below). The **YY_BUFFER_STATE** type is a pointer to an opaque **struct yy_buffer_state** structure, so you may safely initialize YY_BUFFER_STATE variables to **((YY_BUFFER_STATE) 0)** if you wish, and also refer to the opaque structure in order to correctly declare input buffers in source files other than that of your scanner. Note that the *FILE* pointer in the call to **yy_create_buffer** is only used as the value of *yyin* seen by **YY_INPUT;** if you redefine **YY_INPUT** so it no longer uses *yyin,* then you can safely pass a nil *FILE* pointer to **yy_create_buffer.** You select a particular buffer to scan from using:

```
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 set things up for continued scanning, instead of opening a new file and pointing *yyin* at it. Note also that switching input sources via either **yy_switch_to_buffer()** or **yywrap()** does *not* change the start condition.

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

is used to reclaim the storage associated with a buffer. (**buffer** can be nil, in which case the routine does nothing.) You can also clear the current contents of a buffer using:

```
void yy_flush_buffer( YY_BUFFER_STATE buffer )
```

This function discards the buffer's contents, so the next time the scanner attempts to match a token from the buffer, it will first fill the buffer anew using **YY_INPUT.**

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_delete_buffer( YY_CURRENT_BUFFER );
        yy_switch_to_buffer(
             include_stack[include_stack_ptr] );
        }
    }
```

Three routines are available for setting up input buffers for scanning in-memory strings instead of files. All of them create a new input buffer for scanning the string, and return a corresponding **YY_BUFFER_STATE** handle (which you should delete with **yy_delete_buffer()** when done with it). They also switch to the new buffer using **yy_switch_to_buffer(),** so the next call to **yylex()** will start scanning the string.

yy_scan_string(const char *str)
: scans a NUL-terminated string.

yy_scan_bytes(const char *bytes, int len)
: scans *len* bytes (including possibly NUL's) starting at location *bytes.*

Note that both of these functions create and scan a *copy* of the string or bytes. (This may be desirable, since **yylex()** modifies the contents of the buffer it is scanning.) You can avoid the copy by using:

yy_scan_buffer(char *base, yy_size_t size)
: which scans in place the buffer starting at *base*, consisting of *size* bytes, the last two bytes of which *must* be **YY_END_OF_BUFFER_CHAR** (ASCII NUL). These last two bytes are not scanned; thus, scanning consists of **base[0]** through **base[size-2],** inclusive.

: If you fail to set up *base* in this manner (i.e., forget the final two **YY_END_OF_BUFFER_CHAR** bytes), then **yy_scan_buffer()** returns a nil pointer instead of creating a new input buffer.

: The type **yy_size_t** is an integral type to which you can cast an integer expression reflecting the size of the buffer.

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:

- assigning *yyin* to a new input file (in previous versions of flex, after doing the assignment you had to call the special action **YY_NEW_FILE;** this is no longer necessary);
- executing a *return* statement;
- executing 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" );
         else
            yyterminate();
         }
```

MISCELLANEOUS MACROS

The macro **YY_USER_ACTION** can be defined 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. When **YY_USER_ACTION** is invoked, the variable *yy_act* gives the number of the matched rule (rules are numbered starting with 1). Suppose you want to profile how often each of your rules is matched. The following would do the trick:

```
#define YY_USER_ACTION ++ctr[yy_act]
```

where *ctr* is an array to hold the counts for the different rules. Note that the macro **YY_NUM_RULES** gives the total number of rules (including the default rule, even if you use **−s),** so a correct declaration for *ctr* is:

```
int ctr[YY_NUM_RULES];
```

The macro **YY_USER_INIT** may be defined 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.

The macro **yy_set_interactive(is_interactive)** can be used to control whether the current buffer is considered *interactive.* An interactive buffer is processed more slowly, but must be used when the scanner's input source is indeed interactive to avoid problems due to waiting to fill buffers (see the discussion of the **−I** flag below). A non-zero value in the macro invocation marks the buffer as interactive, a zero value as non-interactive. Note that use of this macro overrides **%option always-interactive** or **%option never-interactive** (see Options below). **yy_set_interactive()** must be invoked prior to beginning to scan the buffer that is (or is not) to be considered interactive.

The macro **yy_set_bol(at_bol)** can be used to control whether the current buffer's scanning context for the next token match is done as though at the beginning of a line. A non-zero macro argument makes rules anchored with

The macro **YY_AT_BOL()** returns true if the next token scanned from the current buffer will have '^' rules active, false otherwise.

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.

VALUES AVAILABLE TO THE USER

This section summarizes the various values available to the user in the rule actions.

- **char *yytext** holds the text of the current token. It may be modified but not lengthened (you cannot append characters to the end).

If the special directive **%array** appears in the first section of the scanner description, then **yytext** is instead declared **char yytext[YYLMAX],** where **YYLMAX** is a macro definition that you can redefine in the first section if you don't like the default value (generally 8KB). Using **%array** results in somewhat slower scanners, but the value of **yytext** becomes immune to calls to *input()* and *unput(),* which potentially destroy its value when **yytext** is a character pointer. The opposite of **%array** is **%pointer,** which is the default.

You cannot use **%array** when generating C++ scanner classes (the **−+** flag).

- **int yyleng** holds the length of the current token.
- **FILE *yyin** is the file which by default *flex* reads from. It may be redefined but doing so only makes sense before scanning begins or after an EOF has been encountered. Changing it in the midst of scanning will have unexpected results since *flex* buffers its input; use **yyrestart()** instead. Once scanning terminates because an end-of-file has been seen, you can assign *yyin* at the new input file and then call the scanner again to continue scanning.
- **void yyrestart(FILE *new_file)** may be called to point *yyin* at the new input file. The switch-over to the new file is immediate (any previously buffered-up input is lost). Note that calling **yyrestart()** with *yyin* as an argument thus throws away the current input buffer and continues scanning the same input file.
- **FILE *yyout** is the file to which **ECHO** actions are done. It can be reassigned by the user.
- **YY_CURRENT_BUFFER** returns a **YY_BUFFER_STATE** handle to the current buffer.
- **YY_START** returns an integer value corresponding to the current start condition. You can subsequently use this value with **BEGIN** to return to that start condition.

INTERFACING WITH YACC

One of the main uses of *flex* is as a companion to the *yacc* parser-generator. *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,* one specifies the **−d** option to *yacc* to instruct it to generate the file **y.tab.h** containing definitions of all the **%tokens** appearing in the *yacc* input. This file is then included 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;
```

OPTIONS

flex has the following options:

−b Generate backing-up information to *lex.backup.* This is a list of scanner states which require backing up and the input characters on which they do so. By adding rules one can remove backing-up states. If *all* backing-up states are eliminated and **−Cf** or **−CF** 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 the section on Performance Considerations below.)

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

−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 backs up, 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 *fast scanner.* No table compression is done and stdio is bypassed. The result is large but fast. This option is equivalent to **−Cfr** (see below).

−h generates a "help" summary of *flex's* options to *stdout* and then exits. **−?** and **−−help** are synonyms for **−h.**

−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).

−l turns on maximum compatibility with the original AT&T *lex* implementation. Note that this does not mean *full* compatibility. Use of this option costs a considerable amount of performance, and it cannot be used with the **−+, -f, -F, -Cf,** or **-CF** options. For details on the compatibilities it provides, see the section "Incompatibilities With Lex And POSIX" below. This option also results in the name **YY_FLEX_LEX_COMPAT** being #define'd in the generated scanner.

−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 serious loss of performance in the resulting scanner. If you give the flag twice, you will also get comments regarding features that lead to minor performance losses.

Note that the use of **REJECT, %option yylineno,** and variable trailing context (see the Deficiencies / Bugs section below) entails a substantial performance penalty; use of *yymore(),* the ˆ 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* (same as reported by **−V),** and the next line the flags used when generating the scanner, including those that are on by default.

−w suppresses warning messages.

−B instructs *flex* to generate a *batch* scanner, the opposite of *interactive* scanners generated by **−I** (see below). In general, you use **−B** when you are *certain* that your scanner will never be used interactively, and you want to squeeze a *little* more performance out of it. If your goal is instead to squeeze out a *lot* more performance, you should be using the **−Cf** or **−CF** options (discussed below), which turn on **−B** automatically anyway.

−F specifies that the *fast* scanner table representation should be used (and stdio bypassed). 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 **−CFr** (see below). It cannot be used with **−+.**

−I instructs *flex* to generate an *interactive* scanner. An interactive scanner is one that only looks ahead to decide what token has been matched if it absolutely must. It turns out that always looking one extra character ahead, even if the scanner has already seen enough text to disambiguate the current token, is a bit faster than only looking ahead when necessary. But scanners that always look ahead give dreadful interactive performance; for example, when a user types a newline, it is not recognized as a newline token until they enter *another* token, which often means typing in another whole line.

Flex scanners default to *interactive* unless you use the **−Cf** or **−CF** table-compression options (see below). That's because if you're looking for high-performance you should be using one of these options, so if you didn't, *flex* assumes you'd rather trade off a bit of run-time performance for intuitive interactive behavior. Note also that you *cannot* use **−I** in conjunction with **−Cf** or **−CF.** Thus, this option is not really needed; it is on by default for all those cases in which it is allowed.

You can force a scanner to *not* be interactive by using **−B** (see above).

−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 either the original *flex* input file (if the errors are due to code in the input file), or **lex.yy.c** (if the errors are *flex's* fault -- you should report these sorts of errors to the email address given below).

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

−V prints the version number to *stdout* and exits. **−−version** is a synonym for **−V.**

−7 instructs *flex* to generate a 7-bit scanner, i.e., one which can only recognized 7-bit characters in its input. The advantage of using **−7** is that the scanner's tables can be up to half the size of those generated using the **−8** option (see below). The disadvantage is that such scanners often hang or crash if their input contains an 8-bit character.

Note, however, that unless you generate your scanner using the **−Cf** or **−CF** table compression options, use of **−7** will save only a small amount of table space, and make your scanner considerably less portable. *Flex's* default behavior is to generate an 8-bit scanner unless you use the **−Cf** or **−CF,** in which case *flex* defaults to generating 7-bit scanners unless your site was always configured to generate 8-bit scanners (as will often be the case with non-USA sites). You can tell whether flex generated a 7-bit or an 8-bit scanner by inspecting the flag summary in the **−v** output as described above.

Note that if you use **−Cfe** or **−CFe** (those table compression options, but also using equivalence classes as discussed see below), flex still defaults to generating an 8-bit scanner, since usually with these compression options full 8-bit tables are not much more expensive than 7-bit tables.

−8 instructs *flex* to generate an 8-bit scanner, i.e., one which can recognize 8-bit characters. This flag is only needed for scanners generated using **−Cf** or **−CF,** as otherwise flex defaults to generating an 8-bit scanner anyway.

See the discussion of **−7** above for flex's default behavior and the tradeoffs between 7-bit and 8-bit scanners.

−+ specifies that you want flex to generate a C++ scanner class. See the section on Generating C++ Scanners below for details.

−C[aefFmr]
controls the degree of table compression and, more generally, trade-offs between small scanners and fast scanners.

−Ca ("align") instructs flex to trade off larger tables in the generated scanner for faster performance because the elements of the tables are better aligned for memory access and computation. On some RISC architectures, fetching and manipulating longwords is more efficient than with smaller-sized units such as shortwords. This option can double the size of the tables used by your scanner.

−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* should 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. This option cannot be used with **−+.**

−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).

−Cr causes the generated scanner to *bypass* use of the standard I/O library (stdio) for input. Instead of calling **fread()** or **getc(),** the scanner will use the **read()** system call, resulting in a performance gain which varies from system to system, but in general is probably negligible unless you are also using **−Cf** or **−CF.** Using **−Cr** can cause strange behavior if, for example, you read from *yyin* using stdio prior to calling the scanner (because the scanner will miss whatever text your previous reads left in the stdio input buffer).

−Cr has no effect if you define **YY_INPUT** (see The Generated Scanner above).

A lone **−C** specifies that the scanner tables should be compressed but neither equivalence classes nor meta-equivalence classes should be used.

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 being compressed. Otherwise the options may be freely mixed, and are cumulative.

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 & smallest
      -Cem
      -Cm
```

```
        -Ce
        -C
        -C{f,F}e
        -C{f,F}
        -C{f,F}a
    fastest & 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.

−ooutput
directs flex to write the scanner to the file **output** instead of **lex.yy.c.** If you combine **−o** with the **−t** option, then the scanner is written to *stdout* but its **#line** directives (see the **−L** option above) refer to the file **output.**

−Pprefix
changes the default *yy* prefix used by *flex* for all globally-visible variable and function names to instead be *prefix.* For example, **−Pfoo** changes the name of **yytext** to **footext.** It also changes the name of the default output file from **lex.yy.c** to **lex.foo.c.** Here are all of the names affected:

```
yy_create_buffer
yy_delete_buffer
yy_flex_debug
yy_init_buffer
yy_flush_buffer
yy_load_buffer_state
yy_switch_to_buffer
yyin
yyleng
yylex
yylineno
yyout
yyrestart
yytext
yywrap
```

(If you are using a C++ scanner, then only **yywrap** and **yyFlexLexer** are affected.) Within your scanner itself, you can still refer to the global variables and functions using either version of their name; but externally, they have the modified name.

This option lets you easily link together multiple *flex* programs into the same executable. Note, though, that using this option also renames **yywrap(),** so you now *must* either provide your own (appropriately-named) version of the routine for your scanner, or use **%option noyywrap,** as linking with **−lfl** no longer provides one for you by default.

−Sskeleton_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.

flex also provides a mechanism for controlling options within the scanner specification itself, rather than from the flex command-line. This is done by including **%option** directives in the first section of the scanner specification. You can specify multiple options with a single **%option** directive, and multiple directives in the first section of your flex input file.

Most options are given simply as names, optionally preceded by the word "no" (with no intervening whitespace) to negate their meaning. A number are equivalent to flex flags or their negation:

```
7bit            -7 option
8bit            -8 option
align           -Ca option
backup          -b option
batch           -B option
c++             -+ option

caseful or
case-sensitive  opposite of -i (default)

case-insensitive or
caseless        -i option

debug           -d option
default         opposite of -s option
ecs             -Ce option
fast            -F option
full            -f option
interactive     -I option
lex-compat      -l option
meta-ecs        -Cm option
perf-report     -p option
read            -Cr option
stdout          -t option
verbose         -v option
warn            opposite of -w option
                (use "%option nowarn" for -w)

array           equivalent to "%array"
pointer         equivalent to "%pointer" (default)
```

Some **%option's** provide features otherwise not available:

always-interactive
instructs flex to generate a scanner which always considers its input "interactive". Normally, on each new input file the scanner calls **isatty()** in an attempt to determine whether the scanner's input source is interactive and thus should be read a character at a time. When this option is used, however, then no such call is made.

main directs flex to provide a default **main()** program for the scanner, which simply calls **yylex().** This option implies **noyywrap** (see below).

never-interactive
instructs flex to generate a scanner which never considers its input "interactive" (again, no call made to **isatty()).** This is the opposite of **always-interactive.**

stack enables the use of start condition stacks (see Start Conditions above).

stdinit if set (i.e., **%option stdinit)** initializes *yyin* and *yyout* to *stdin* and *stdout,* instead of the default of *nil.* Some existing *lex* programs depend on this behavior, even though it is not compliant with ANSI C, which does not require *stdin* and *stdout* to be compile-time constant.

yylineno
directs *flex* to generate a scanner that maintains the number of the current line read from its

input in the global variable **yylineno.** This option is implied by **%option lex-compat.**

yywrap if unset (i.e., **%option noyywrap),** makes the scanner not call **yywrap()** upon an end-of-file, but simply assume that there are no more files to scan (until the user points *yyin* at a new file and calls **yylex()** again).

flex scans your rule actions to determine whether you use the **REJECT** or **yymore()** features. The **reject** and **yymore** options are available to override its decision as to whether you use the options, either by setting them (e.g., **%option reject)** to indicate the feature is indeed used, or unsetting them to indicate it actually is not used (e.g., **%option noyymore).**

Three options take string-delimited values, offset with '=':

```
%option outfile="ABC"
```

is equivalent to **-oABC,** and

```
%option prefix="XYZ"
```

is equivalent to **-PXYZ.** Finally,

```
%option yyclass="foo"
```

only applies when generating a C++ scanner (**−+** option). It informs *flex* that you have derived **foo** as a subclass of **yyFlexLexer,** so *flex* will place your actions in the member function **foo::yylex()** instead of **yyFlexLexer::yylex().** It also generates a **yyFlexLexer::yylex()** member function that emits a run-time error (by invoking **yyFlexLexer::LexerError())** if called. See Generating C++ Scanners, below, for additional information.

A number of options are available for lint purists who want to suppress the appearance of unneeded routines in the generated scanner. Each of the following, if unset (e.g., **%option nounput**), results in the corresponding routine not appearing in the generated scanner:

```
input, unput
yy_push_state, yy_pop_state, yy_top_state
yy_scan_buffer, yy_scan_bytes, yy_scan_string
```

(though **yy_push_state()** and friends won't appear anyway unless you use **%option stack).**

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 on scanner speed of the table compression **−C** options outlined above, there are a number of options/actions which degrade performance. These are, from most expensive to least:

```
REJECT
%option yylineno
arbitrary trailing context

pattern sets that require backing up
%array
%option interactive
%option always-interactive

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

with the first three all being quite expensive and the last two being quite cheap. Note also that **unput()** is implemented as a routine call that potentially does quite a bit of work, while **yyless()** is a quite-cheap macro; so if just putting back some excess text you scanned, use **yyless().**

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

Getting rid of backing up is messy and often may be an enormous amount of work for a complicated scanner. In principal, one begins by using the **−b** flag to generate a *lex.backup* 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 back up.
```

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 back up 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 an 'a', the scanner will have to back up to accept "foo". Similarly, the comment for State #9 concerns when "fooba" has been scanned and an 'r' does not follow.

The final comment reminds us that there's no point going to all the trouble of removing backing up from the rules unless we're using **−Cf** or **−CF,** since there's no performance gain doing so with compressed scanners.

The way to remove the backing up 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 backing up 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.

Backing up 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 backing up (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 backing up).

It's important to keep in mind that you gain the benefits of eliminating backing up only if you eliminate *every* instance of backing up. Leaving just one means you gain nothing.

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 Deficiencies / Bugs below).

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 backing up 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 to back up 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 backing up, 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.

Another final note regarding performance: as mentioned above in the section How the Input is Matched, dynamically resizing **yytext** to accommodate huge tokens is a slow process because it presently requires that the (huge) token be rescanned from the beginning. Thus if performance is vital, you should attempt to match "large" quantities of text but not "huge" quantities, where the cutoff between the two is at about 8K characters/token.

GENERATING C++ SCANNERS

flex provides two different ways to generate scanners for use with C++. The first way is to simply compile a scanner generated by *flex* using a C++ compiler instead of a C compiler. You should not encounter any compilations errors (please report any you find to the email address given in the Author section below). You can then use C++ code in your rule actions instead of C code. Note that the default input source for your scanner remains *yyin,* and default echoing is still done to *yyout.* Both of these remain *FILE ∗* variables and not C++ *streams.*

You can also use *flex* to generate a C++ scanner class, using the **−+** option (or, equivalently, **%option c++),** which is automatically specified if the name of the flex executable ends in a '+', such as *flex++.* When using this option, flex defaults to generating the scanner to the file **lex.yy.cc** instead of **lex.yy.c.** The generated scanner includes the header file *FlexLexer.h,* which defines the interface to two C++ classes.

The first class, **FlexLexer,** provides an abstract base class defining the general scanner class interface. It provides the following member functions:

const char∗ YYText()
: returns the text of the most recently matched token, the equivalent of **yytext.**

int YYLeng()
: returns the length of the most recently matched token, the equivalent of **yyleng.**

int lineno() const
: returns the current input line number (see **%option yylineno),** or **1** if **%option yylineno** was not used.

void set_debug(int flag)
: sets the debugging flag for the scanner, equivalent to assigning to **yy_flex_debug** (see the Options section above). Note that you must build the scanner using **%option debug** to include debugging information in it.

int debug() const
: returns the current setting of the debugging flag.

Also provided are member functions equivalent to **yy_switch_to_buffer(), yy_create_buffer()** (though the first argument is an **istream∗** object pointer and not a **FILE∗), yy_flush_buffer(), yy_delete_buffer(),** and **yyrestart()** (again, the first argument is a **istream∗** object pointer).

The second class defined in *FlexLexer.h* is **yyFlexLexer,** which is derived from **FlexLexer.** It defines the following additional member functions:

yyFlexLexer(istream∗ arg_yyin = 0, ostream∗ arg_yyout = 0)
: constructs a **yyFlexLexer** object using the given streams for input and output. If not specified, the streams default to **cin** and **cout,** respectively.

virtual int yylex()
: performs the same role is **yylex()** does for ordinary flex scanners: it scans the input stream, consuming tokens, until a rule's action returns a value. If you derive a subclass **S** from **yyFlexLexer** and want to access the member functions and variables of **S** inside **yylex(),** then you need to use **%option yyclass="S"** to inform *flex* that you will be using that subclass instead of **yyFlexLexer.** In this case, rather than generating **yyFlexLexer::yylex(),** *flex* generates **S::yylex()** (and also generates a dummy **yyFlexLexer::yylex()** that calls **yyFlexLexer::LexerError()** if called).

virtual void switch_streams(istream∗ new_in = 0,
: **ostream∗ new_out = 0)** reassigns **yyin** to **new_in** (if non-nil) and **yyout** to **new_out** (ditto), deleting the previous input buffer if **yyin** is reassigned.

int yylex(istream∗ new_in, ostream∗ new_out = 0)
: first switches the input streams via **switch_streams(new_in, new_out)** and then returns the value of **yylex().**

In addition, **yyFlexLexer** defines the following protected virtual functions which you can redefine in derived classes to tailor the scanner:

virtual int LexerInput(char∗ buf, int max_size)

reads up to **max_size** characters into **buf** and returns the number of characters read. To indicate end-of-input, return 0 characters. Note that "interactive" scanners (see the **−B** and **−I** flags) define the macro **YY_INTERACTIVE.** If you redefine **LexerInput()** and need to take different actions depending on whether or not the scanner might be scanning an interactive input source, you can test for the presence of this name via **#ifdef.**

virtual void LexerOutput(const char* buf, int size)
writes out **size** characters from the buffer **buf,** which, while NUL-terminated, may also contain "internal" NUL's if the scanner's rules can match text with NUL's in them.

virtual void LexerError(const char* msg)
reports a fatal error message. The default version of this function writes the message to the stream **cerr** and exits.

Note that a **yyFlexLexer** object contains its *entire* scanning state. Thus you can use such objects to create reentrant scanners. You can instantiate multiple instances of the same **yyFlexLexer** class, and you can also combine multiple C++ scanner classes together in the same program using the **−P** option discussed above.

Finally, note that the **%array** feature is not available to C++ scanner classes; you must use **%pointer** (the default).

Here is an example of a simple C++ scanner:

```
    // An example of using the flex C++ scanner class.

%{
int mylineno = 0;
%}

string  \"[^\n"]+\"

ws      [ \t]+

alpha   [A-Za-z]
dig     [0-9]
name    ({alpha}|{dig}|\$)({alpha}|{dig}|[_.\-/$])*
num1    [-+]?{dig}+\.?([eE][-+]?{dig}+)?
num2    [-+]?{dig}*\.{dig}+([eE][-+]?{dig}+)?
number  {num1}|{num2}

%%

{ws}    /* skip blanks and tabs */

"/*"    {
        int c;

        while((c = yyinput()) != 0)
            {
            if(c == '\n')
                ++mylineno;

            else if(c == '*')
                {
                if((c = yyinput()) == '/')
```

```
                break;
            else
                unput(c);
            }
        }
    }

{number}  cout << "number " << YYText() << '\n';

\n        mylineno++;

{name}    cout << "name " << YYText() << '\n';

{string}  cout << "string " << YYText() << '\n';

%%

int main( int /* argc */, char** /* argv */ )
    {
    FlexLexer* lexer = new yyFlexLexer;
    while(lexer->yylex() != 0)
        ;
    return 0;
    }
```

If you want to create multiple (different) lexer classes, you use the **−P** flag (or the **prefix=** option) to rename each **yyFlexLexer** to some other **xxFlexLexer.** You then can include <**FlexLexer.h**> in your other sources once per lexer class, first renaming **yyFlexLexer** as follows:

```
#undef yyFlexLexer
#define yyFlexLexer xxFlexLexer
#include <FlexLexer.h>

#undef yyFlexLexer
#define yyFlexLexer zzFlexLexer
#include <FlexLexer.h>
```

if, for example, you used **%option prefix="xx"** for one of your scanners and **%option prefix="zz"** for the other.

IMPORTANT: the present form of the scanning class is *experimental* and may change considerably between major releases.

INCOMPATIBILITIES WITH LEX AND POSIX

flex is a rewrite of the AT&T Unix *lex* tool (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. Flex is fully compliant with the POSIX *lex* specification, except that when using **%pointer** (the default), a call to **unput()** destroys the contents of **yytext,** which is counter to the POSIX specification.

In this section we discuss all of the known areas of incompatibility between flex, AT&T lex, and the POSIX specification.

flex's **−l** option turns on maximum compatibility with the original AT&T *lex* implementation, at the cost of a major loss in the generated scanner's performance. We note below which incompatibilities can be overcome using the **−l** option.

flex is fully compatible with *lex* with the following exceptions:

- The undocumented *lex* scanner internal variable **yylineno** is not supported unless **−l** or **%option yylineno** is used.

  yylineno should be maintained on a per-buffer basis, rather than a per-scanner (single global variable) basis.

  yylineno is not part of the POSIX specification.

- 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 defining the **YY_INPUT** macro.

  The *flex* restriction that **input()** cannot be redefined is in accordance with the POSIX specification, which simply does not specify any way of controlling the scanner's input other than by making an initial assignment to *yyin.*

- The **unput()** routine is not redefinable. This restriction is in accordance with POSIX.

- *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 );
  ```

  Note that this call will throw away any buffered input; usually this isn't a problem with an interactive scanner.

  Also note that flex C++ scanner classes *are* reentrant, so if using C++ is an option for you, you should use them instead. See "Generating C++ Scanners" above for details.

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

  output() is not part of the POSIX specification.

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

- 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 if the definition begins with ^ or ends with **$** then it is *not* expanded with parentheses, to allow these operators to appear in definitions without losing their special meanings. But the <**s**>**, /,** and <<**EOF**>> operators cannot be used in a *flex* definition.

Using **−l** results in the *lex* behavior of no parentheses around the definition.

The POSIX specification is that the definition be enclosed in parentheses.

- Some implementations of *lex* allow a rule's action to begin on a separate line, if the rule's pattern has trailing whitespace:

```
%%
foo|bar<space here>
  { foobar_action(); }
```

  flex does not support this feature.

- The *lex* **%r** (generate a Ratfor scanner) option is not supported. It is not part of the POSIX specification.
- After a call to **unput(),** *yytext* is undefined until the next token is matched, unless the scanner was built using **%array.** This is not the case with *lex* or the POSIX specification. The **−l** option does away with this incompatibility.
- 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 POSIX specification.
- 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 POSIX specification.
- 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.* Scanners also include **YY_FLEX_MAJOR_VERSION** and **YY_FLEX_MINOR_VERSION** indicating which version of *flex* generated the scanner (for example, for the 2.5 release, these defines would be 2 and 5 respectively).

The following *flex* features are not included in *lex* or the POSIX specification:

```
C++ scanners
%option
start condition scopes
start condition stacks
interactive/non-interactive scanners
yy_scan_string() and friends
yyterminate()
yy_set_interactive()
yy_set_bol()
YY_AT_BOL()
<<EOF>>
<*>
YY_DECL
YY_START
YY_USER_ACTION
YY_USER_INIT
#line directives
%{}'s around actions
multiple actions on a line
```

plus almost all of the flex flags. The last feature in the list refers to the fact that with *flex* you can put multiple actions on the same line, separated with semi-colons, 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.

DIAGNOSTICS

warning, rule cannot be matched indicates that the given rule cannot be matched because it follows other rules that will always match the same text as it. For example, in the following "foo" cannot be matched because it comes after an identifier "catch-all" rule:

```
[a-z]+    got_identifier();
foo       got_foo();
```

Using **REJECT** in a scanner suppresses this warning.

warning, **−s** *option given but default rule can be matched* means that it is possible (perhaps only in a particular start condition) that the default rule (match any single character) is the only one that will match a particular input. Since **−s** was given, presumably this is not intended.

reject_used_but_not_detected undefined or *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). Use **%option reject** or **%option yymore** to indicate to flex that you really do use these features.

flex scanner jammed - a scanner compiled with **−s** has encountered an input string which wasn't matched by any of its rules. This error can also occur due to internal problems.

token too large, exceeds YYLMAX - your scanner uses **%array** and one of its rules matched a string longer than the **YYLMAX** constant (8K bytes by default). You can increase the value by #define'ing **YYLMAX** in the definitions section of your *flex* input.

scanner requires −8 flag to use the character 'x' - Your scanner specification includes recognizing the 8-bit character *'x'* and you did not specify the −8 flag, and your scanner defaulted to 7-bit because you used the **−Cf** or **−CF** table compression options. See the discussion of the **−7** flag for details.

flex scanner push-back overflow - you used **unput()** to push back so much text that the scanner's buffer could not hold both the pushed-back text and the current token in **yytext.** Ideally the scanner should dynamically resize the buffer in this case, but at present it does not.

input buffer overflow, can't enlarge buffer because scanner uses REJECT - the scanner was working on matching an extremely large token and needed to expand the input buffer. This doesn't work with scanners that use **REJECT.**

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 );
```

or, as noted above, switch to using the C++ scanner class.

too many start conditions in <> you listed more start conditions in a <> construct than exist (so you must have listed at least one of them twice).

FILES

−lfl library with which scanners must be linked.

lex.yy.c generated scanner (called *lexyy.c* on some systems).

lex.yy.cc
generated C++ scanner class, when using **-+.**

<FlexLexer.h>
header file defining the C++ scanner base class, **FlexLexer,** and its derived class, **yyFlexLexer.**

flex.skl skeleton scanner. This file is only used when building flex, not when flex executes.

lex.backup
backing-up information for **−b** flag (called *lex.bck* on some systems).

DEFICIENCIES / 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, in the following:

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

Use of **unput()** invalidates yytext and yyleng, unless the **%array** directive or the **−l** option has been used.

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

Dynamic resizing of the input buffer is slow, as it entails rescanning all the text matched so far by the current (generally huge) token.

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.

The *flex* internal algorithms need documentation.

SEE ALSO

lex(1), yacc(1), sed(1), awk(1).

John Levine, Tony Mason, and Doug Brown, *Lex & Yacc,* O'Reilly and Associates. Be sure to get the 2nd edition.

M. E. Lesk and E. Schmidt, *LEX − Lexical Analyzer Generator*

Alfred Aho, Ravi Sethi and Jeffrey Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley (1986). Describes the pattern-matching techniques used by *flex* (deterministic finite automata).

AUTHOR

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 Francois Pinard, Casey Leedom, Robert Abramovitz, Stan Adermann, Terry Allen, David Barker-Plummer, John Basrai, Neal Becker, Nelson H.F. Beebe, benson@odi.com, Karl Berry, Peter A. Bigot, Simon Blanchard, Keith Bostic, Frederic Brehm, Ian Brockbank, Kin Cho, Nick Christopher, Brian Clapper, J.T. Conklin, Jason Coughlin, Bill Cox, Nick Cropper, Dave Curtis, Scott David Daniels, Chris G. Demetriou, Theo Deraadt, Mike Donahue, Chuck Doucette, Tom Epperly, Leo Eskin, Chris Faylor, Chris Flatters, Jon Forrest, Jeffrey Friedl, Joe Gayda, Kaveh R. Ghazi, Wolfgang Glunz, Eric Goldman, Christopher M. Gould, Ulrich Grepel, Peer Griebel, Jan Hajic, Charles Hemphill, NORO Hideo, Jarkko Hietaniemi, Scott Hofmann, Jeff Honig, Dana Hudes, Eric Hughes, John Interrante, Ceriel Jacobs, Michal Jaegermann, Sakari Jalovaara, Jeffrey R. Jones, Henry Juengst, Klaus Kaempf, Jonathan I. Kamens, Terrence O Kane, Amir Katz, ken@ken.hilco.com, Kevin B. Kenny, Steve Kirsch, Winfried Koenig, Marq Kole, Ronald Lamprecht, Greg Lee, Rohan Lenard, Craig Leres, John Levine, Steve Liddle, David Loffredo, Mike Long, Mohamed el Lozy, Brian Madsen, Malte, Joe Marshall, Bengt Martensson, Chris Metcalf, Luke Mewburn, Jim Meyering, R. Alexander Milowski, Erik Naggum, G.T. Nicol, Landon Noll, James Nordby, Marc Nozell, Richard Ohnemus, Karsten Pahnke, Sven Panne, Roland Pesch, Walter Pelissero, Gaumond Pierre, Esmond Pitt, Jef Poskanzer, Joe Rahmeh, Jarmo Raiha, Frederic Raimbault, Pat Rankin, Rick Richardson, Kevin Rodgers, Kai Uwe Rommel, Jim Roskind, Alberto Santini, Andreas Scherer, Darrell Schiebel, Raf Schietekat, Doug Schmidt, Philippe Schnoebelen, Andreas Schwab, Larry Schwimmer, Alex Siegel, Eckehard Stolz, Jan-Erik Strvmquist, Mike Stump, Paul Stuart, Dave Tallman, Ian Lance Taylor, Chris Thewalt, Richard M. Timoney, Jodi Tsai, Paul Tuinenga, Gary Weik, Frank Whaley, Gerhard Wilhelms, Kent Williams, Ken Yap, Ron Zellar, Nathan Zelle, David Zuhn, and those whose names have slipped my marginal mail-archiving skills but whose contributions are appreciated all the same.

Thanks to Keith Bostic, Jon Forrest, Noah Friedman, John Gilmore, Craig Leres, John Levine, Bob Mulcahy, G.T. Nicol, Francois Pinard, 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 Kent Williams and Tom Epperly for C++ class support; to Ove Ewerlid for support of NUL's; and to Eric Hughes for support of multiple buffers.

This work was primarily done when I was with 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@ee.lbl.gov.