Flex, version 2.5.31

A fast scanner generator
Edition 2.5.31, 27 March 2003

Vern Paxson
W. L. Estes
John Millaway

Table of Contents

flex

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

This edition of *The flex Manual* documents `flex` version 2.5.31. It was last updated on 27 March 2003.

1 Copyright

2 Reporting Bugs

If you have problems with `flex` or think you have found a bug, please send mail detailing your problem to `lex-help@lists.sourceforge.net`. Patches are always welcome.

3 Introduction

`flex` is a tool for generating *scanners*. A scanner is a program which recognizes lexical patterns in text. The `flex` program 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`' by default, which defines a routine `yylex()`. This file can be compiled and linked with the flex runtime 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.

4 Some Simple Examples

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

The following `flex` input specifies a scanner which, when 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 '`%%`' symbol 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 character and line 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.

5 Format of the Input File

The `flex` input file consists of three sections, separated by a line containing only '`%%`'.

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

5.1 Format of the Definitions Section

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

Name definitions have the form:

```
name definition
```

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

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

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

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

is identical to

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

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

An unindented comment (i.e., a line beginning with '`/*`') is copied verbatim to the output up to the next '`*/`'.

Any *indented* text or text enclosed in '`%{`' and '`%}`' is also copied verbatim to the output (with the %{ and %} symbols removed). The %{ and %} symbols must appear unindented on lines by themselves.

A `%top` block is similar to a '`%{`' ... '`%}`' block, except that the code in a `%top` block is relocated to the *top* of the generated file, before any flex definitions[1]. The `%top` block is useful when you want certain preprocessor macros to be defined or certain files to be

[1] Actually, `yyIN_HEADER` is defined before the '`%top`' block.

included before the generated code. The single characters, '`{`' and '`}`' are used to delimit the `%top` block, as show in the example below:

```
%top{
    /* This code goes at the "top" of the generated file. */
    #include <stdint.h>
    #include <inttypes.h>
}
```

Multiple `%top` blocks are allowed, and their order is preserved.

5.2 Format of the Rules Section

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 ⟨undefined⟩ [Patterns], page ⟨undefined⟩, for a further description of patterns and actions.

In the rules section, any indented or %{ %} enclosed 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 ⟨undefined⟩ [Lex and Posix], page ⟨undefined⟩, for other such features).

Any *indented* text or text enclosed in '`%{`' and '`%}`' is copied verbatim to the output (with the %{ and %} symbols removed). The %{ and %} symbols must appear unindented on lines by themselves.

5.3 Format of the User Code Section

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.

5.4 Comments in the Input

Flex supports C-style comments, that is, anything between /* and */ is considered a comment. Whenever flex encounters a comment, it copies the entire comment verbatim to the generated source code. Comments may appear just about anywhere, but with the following exceptions:

Comments may not appear in the Rules Section wherever flex is expecting a regular expression. This means comments may not appear at the beginning of a line, or immediately following a list of scanner states.

Comments may not appear on an '`%option`' line in the Definitions Section.

If you want to follow a simple rule, then always begin a comment on a new line, with one or more whitespace characters before the initial '`/*`'). This rule will work anywhere in the input file.

All the comments in the following example are valid:

```
%{
/* code block */
%}

/* Definitions Section */
%x STATE_X

%%
    /* Rules Section */
ruleA   /* after regex */ { /* code block */ } /* after code block */
        /* Rules Section (indented) */
STATE_X>{
ruleC   ECHO;
ruleD   ECHO;
%{
/* code block */
%}
}
%%
/* User Code Section */
```

6 Patterns

The patterns in the input (see ⟨undefined⟩ [Rules Section], page ⟨undefined⟩) 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 ⟨undefined⟩ [Format], page ⟨undefined⟩).

‘`"[xyz]\"foo"`’
the literal string: ‘`[xyz]"foo`’

‘`\X`’ if X is ‘`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 ⟨undefined⟩ [Limitations], page ⟨undefined⟩, regarding dangerous trailing context.)

‘`^r`’ an ‘`r`’, but only at the beginning of a line (i.e., when 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 ⟨undefined⟩ [Start Conditions], page ⟨undefined⟩ 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, ‘`^`’.

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 (see special note on the precedence of the repeat operator, ‘`{}`’, under the documentation for the ‘`--posix`’ POSIX compliance option). 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 repetitions of the string ‘`bar`’, use:

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

And to match a sequence of zero or more repetitions of ‘`foo`’ and ‘`bar`’:

```
(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 character. 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]
```

Some notes on patterns are in order.

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

Character classes with ranges, such as '`[a-Z]`', should be used with caution in a case-insensitive scanner if the range spans upper or lowercase characters. Flex does not know if you want to fold all upper and lowercase characters together, or if you want the literal numeric range specified (with no case folding). When in doubt, flex will assume that you meant the literal numeric range, and will issue a warning. The exception to this rule is a character range such as '`[a-z]`' or '`[S-W]`' where it is obvious that you want case-folding to occur. Here are some examples with the '`-i`' flag enabled:

Range	Result	Literal Range	Alternate Range
'`[a-t]`'	ok	'`[a-tA-T]`'	
'`[A-T]`'	ok	'`[a-tA-T]`'	
'`[A-t]`'	ambiguous	'`[A-Z\[\\\]_‘a-t]`'	'`[a-tA-T]`'
'`[_-{]`'	ambiguous	'`[_‘a-z{]`'	'`[_‘a-zA-Z{]`'
'`[@-C]`'	ambiguous	'`[@ABC]`'	'`[@A-Z\[\\\]_‘abc]`'

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

The following are invalid:

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

Note that the first of these can be written '`foo/bar\n`'.

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

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

If the desired meaning is a '`foo`' or a '`bar`'-followed-by-a-newline, the following could be used (the special `|` action is explained below, see ⟨undefined⟩ [Actions], page ⟨undefined⟩):

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

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

7 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 (see ⟨undefined⟩ [Actions], page ⟨undefined⟩), 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 valid `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 ⟨undefined⟩ [Actions], page ⟨undefined⟩), 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 ⟨undefined⟩ [Actions], page ⟨undefined⟩). 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`.

The `%array` declaration 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 (see ⟨undefined⟩ [Cxx], page ⟨undefined⟩).

8 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"
```

This example 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 ⟨undefined⟩ [Matching], page ⟨undefined⟩). 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 ⟨undefined⟩ [Matching], page ⟨undefined⟩, 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 occurences of '`frob`' in the input would not be counted as words, since the scanner normally executes only one action per token. Multiple uses of `REJECT` 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 ⟨undefined⟩ [Scanner Options], page ⟨undefined⟩).

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 ⟨undefined⟩ [Matching], page ⟨undefined⟩).

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 ⟨undefined⟩ [Generated Scanner], page ⟨undefined⟩). This action is a special case of the more general `yy_flush_buffer()` function, described below (see ⟨undefined⟩ [Multiple Input Buffers], page ⟨undefined⟩)

`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.

9 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 (;).

`flex` generates '`C99`' function definitions by default. However flex does have the ability to generate obsolete, er, '`traditional`', function definitions. This is to support bootstrapping gcc on old systems. Unfortunately, traditional definitions prevent us from using any standard data types smaller than int (such as short, char, or bool) as function arguments. For this reason, future versions of `flex` may generate standard C99 code only, leaving K&R-style functions to the historians. Currently, if you do **not** want '`C99`' definitions, then you must use `%option noansi-definitions`.

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 NULL, 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 it would be better to use `YY_FLUSH_BUFFER` (see ⟨undefined⟩ [Actions], page ⟨undefined⟩). Note that `yyrestart()` does *not* reset the start condition to `INITIAL` (see ⟨undefined⟩ [Start Conditions], page ⟨undefined⟩).

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. The calling sequence for `YY_INPUT()` 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`'.

Here is 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.

For scanning from in-memory buffers (e.g., scanning strings), see ⟨undefined⟩ [Scanning Strings], page ⟨undefined⟩. See ⟨undefined⟩ [Multiple Input Buffers], page ⟨undefined⟩.

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.

10 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 start 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]+@samp{.}[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`).

For historical reasons, start conditions do not have their own name-space within the generated scanner. The start condition names are unmodified in the generated scanner and generated header. See ⟨undefined⟩ [option-header], page ⟨undefined⟩. See ⟨undefined⟩ [option-prefix], page ⟨undefined⟩.

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.

The following routines are available for manipulating stacks of start conditions:

`void` **yy_push_state** (`int new_state`) Function

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** () Function

pops the top of the stack and switches to it via `BEGIN`.

`int` **yy_top_state** () Function

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 ⟨undefined⟩ [Scanner Options], page ⟨undefined⟩).

11 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` statement 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`) Function

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 NULL `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`) Function

The above function 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. If you are looking for a stack of input buffers, then you want to use `yypush_buffer_state()` instead of this function. 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`) Function

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

`void` **yypush_buffer_state** (`YY_BUFFER_STATE buffer`) Function

This function pushes the new buffer state onto an internal stack. The pushed state becomes the new current state. The stack is maintained by flex and will grow as required. This function is intended to be used instead of `yy_switch_to_buffer`, when you want to change states, but preserve the current state for later use.

`void` **yypop_buffer_state** ( ) Function

This function removes the current state from the top of the stack, and deletes it by calling `yy_delete_buffer`. The next state on the stack, if any, becomes the new current state.

`void` **yy_flush_buffer** (`YY_BUFFER_STATE buffer`) Function

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_BUFFER_STATE` **yy_new_buffer** (`FILE *file, int size`) Function

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

`YY_CURRENT_BUFFER` macro returns a `YY_BUFFER_STATE` handle to the current buffer. It should not be used as an lvalue.

Here are two examples of using these features for writing a scanner which expands include files (the `<<EOF>>` feature is discussed below).

This first example uses yypush_buffer_state and yypop_buffer_state. Flex maintains the stack internally.

```
/* the "incl" state is used for picking up the name
 * of an include file
 */
%x incl
%%
include             BEGIN(incl);

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

<incl>[ \t]*      /* eat the whitespace */
<incl>[^ \t\n]+   { /* got the include file name */
        yyin = fopen( yytext, "r" );

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

                    yypush_buffer_state(yy_create_buffer( yyin, YY_BUF_SIZE ));

        BEGIN(INITIAL);
        }

<<EOF>> {
                    yypop_buffer_state();

        if ( !YY_CURRENT_BUFFER )
            {
            yyterminate();
            }
        }
```

The second example, below, does the same thing as the previous example did, but manages its own input buffer stack manually (instead of letting flex do it).

```
/* 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] );
        }
    }
```

The following 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_BUFFER_STATE` **yy_scan_string** (`const char *str`) Function
scans a NUL-terminated string.

`YY_BUFFER_STATE` **yy_scan_bytes** (`const char *bytes, int len`) Function
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_BUFFER_STATE` **yy_scan_buffer** (`char *base, yy_size_t size`) Function
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 NULL pointer instead of creating a new input buffer.

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

12 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 the following 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();
         }
```

13 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 in ⟨undefined⟩ [Scanner Options], page ⟨undefined⟩). 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 ⟨undefined⟩ [Scanner Options], page ⟨undefined⟩). `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 '`^`' active, while a zero argument makes '`^`' rules inactive.

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.

14 Values Available To the User

This chapter 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 `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.

15 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;
```

16 Scanner Options

The various `flex` options are categorized by function in the following menu. If you want to lookup a particular option by name, See ⟨undefined⟩ [Index of Scanner Options], page ⟨undefined⟩.

Even though there are many scanner options, a typical scanner might only specify the following options:

```
%option   8bit reentrant bison-bridge
%option   warn nodefault
%option   yylineno
%option   outfile="scanner.c" header-file="scanner.h"
```

The first line specifies the general type of scanner we want. The second line specifies that we are being careful. The third line asks flex to track line numbers. The last line tells flex what to name the files. (The options can be specified in any order. We just dividied them.)

`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. The names are the same as their long-option equivalents (but without the leading '`--`').

`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)`.

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

yyget_extra, yyset_extra, yyget_leng, yyget_text,
yyget_lineno, yyset_lineno, yyget_in, yyset_in,
yyget_out, yyset_out, yyget_lval, yyset_lval,
yyget_lloc, yyset_lloc, yyget_debug, yyset_debug
```

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

16.1 Options for Specifing Filenames

‘`--header-file=FILE, %option header-file="FILE"`’
: instructs flex to write a C header to ‘`FILE`’. This file contains function prototypes, extern variables, and types used by the scanner. Only the external API is exported by the header file. Many macros that are usable from within scanner actions are not exported to the header file. This is due to namespace problems and the goal of a clean external API.

  While in the header, the macro `yyIN_HEADER` is defined, where ‘`yy`’ is substituted with the appropriate prefix.

  The ‘`--header-file`’ option is not compatible with the ‘`--c++`’ option, since the C++ scanner provides its own header in ‘`yyFlexLexer.h`’.

‘`-oFILE, --outfile=FILE, %option outfile="FILE"`’
: directs flex to write the scanner to the file ‘`FILE`’ instead of ‘`lex.yy.c`’. If you combine ‘`--outfile`’ with the ‘`--stdout`’ option, then the scanner is written to ‘`stdout`’ but its `#line` directives (see the ‘`-l`’ option above) refer to the file ‘`FILE`’.

‘`-t, --stdout, %option stdout`’
: instructs `flex` to write the scanner it generates to standard output instead of ‘`lex.yy.c`’.

‘`-SFILE, --skel=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.

‘`--tables-file=FILE`’
: Write serialized scanner dfa tables to FILE. The generated scanner will not contain the tables, and requires them to be loaded at runtime. See ⟨undefined⟩ [serialization], page ⟨undefined⟩.

‘`--tables-verify`’
: This option is for flex development. We document it here in case you stumble upon it by accident or in case you suspect some inconsistency in the serialized tables. Flex will serialize the scanner dfa tables but will also generate the in-code tables as it normally does. At runtime, the scanner will verify that the serialized tables match the in-code tables, instead of loading them.

16.2 Options Affecting Scanner Behavior

‘`-i, --case-insensitive, %option case-insensitive`’
: 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). For tricky behavior, see ⟨undefined⟩ [case and character ranges], page ⟨undefined⟩.

‘`-l, --lex-compat, %option lex-compat`’
: 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 '`--c++`', '`--full`', '`--fast`', '`-Cf`', or '`-CF`' options. For details on the compatibilities it provides, see ⟨undefined⟩ [Lex and Posix], page ⟨undefined⟩. This option also results in the name `YY_FLEX_LEX_COMPAT` being `#define`'d in the generated scanner.

'`-B, --batch, %option batch`'
instructs `flex` to generate a *batch* scanner, the opposite of *interactive* scanners generated by '`--interactive`' (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, which turn on '`--batch`' automatically anyway.

'`-I, --interactive, %option interactive`'
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 ⟨undefined⟩ [Performance], page ⟨undefined⟩). 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 '`--interactive`' 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 '`--batch`'

'`-7, --7bit, %option 7bit`'
instructs `flex` to generate a 7-bit scanner, i.e., one which can only recognize 7-bit characters in its input. The advantage of using '`--7bit`' is that the scanner's tables can be up to half the size of those generated using the '`--8bit`'. 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 '`--7bit`' 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 '`--verbose`' output as described above.

Note that if you use '`-Cfe`' or '`-CFe`' `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, --8bit, %option 8bit`'
: 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 '`--7bit`' above for `flex`'s default behavior and the tradeoffs between 7-bit and 8-bit scanners.

'`--default, %option default`'
: generate the default rule.

'`--always-interactive, %option 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.

'`--never-interactive, --never-interactive`'
: instructs flex to generate a scanner which never considers its input interactive. This is the opposite of `always-interactive`.

'`-X, --posix, %option posix`'
: turns on maximum compatibility with the POSIX 1003.2-1992 definition of `lex`. Since `flex` was originally designed to implement the POSIX definition of `lex` this generally involves very few changes in behavior. At the current writing the known differences between `flex` and the POSIX standard are:

  In POSIX and AT&T `lex`, the repeat operator, '`{}`', has lower precedence than concatenation (thus '`ab{3}`' yields '`ababab`'). Most POSIX utilities use an Extended Regular Expression (ERE) precedence that has the precedence of the repeat operator higher than concatenation (which causes '`ab{3}`' to yield '`abbb`'). By default, `flex` places the precedence of the repeat operator higher than concatenation which matches the ERE processing of other POSIX utilities. When either '`--posix`' or '`-l`' are specified, `flex` will use the traditional AT&T and POSIX-compliant precedence for the repeat operator where concatenation has higher precedence than the repeat operator.

'`--stack, %option stack`'
: enables the use of start condition stacks (see ⟨undefined⟩ [Start Conditions], page ⟨undefined⟩).

'`--stdinit, %option stdinit`'
: if set (i.e., **%option stdinit)** initializes `yyin` and `yyout` to '`stdin`' and '`stdout`', instead of the default of '`NULL`'. 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. In a reentrant scanner,

however, this is not a problem since initialization is performed in `yylex_init` at runtime.

'`--yylineno, %option 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`. In a reentrant C scanner, the macro `yylineno` is accessible regardless of the value of `%option yylineno`, however, its value is not modified by `flex` unless `%option yylineno` is enabled.

'`--yywrap, %option yywrap`'
if unset (i.e., `--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).

16.3 Code-Level And API Options

'`--ansi-definitions, %option ansi-definitions`'
instruct flex to generate ANSI C99 definitions for functions. This option is enabled by default. If `%option noansi-definitions` is specified, then the obsolete style is generated.

'`--ansi-prototypes, %option ansi-prototypes`'
instructs flex to generate ANSI C99 prototypes for functions. This option is enabled by default. If `noansi-prototypes` is specified, then prototypes will have empty parameter lists.

'`--bison-bridge, %option bison-bridge`'
instructs flex to generate a C scanner that is meant to be called by a `GNU bison` parser. The scanner has minor API changes for `bison` compatibility. In particular, the declaration of `yylex` is modified to take an additional parameter, `yylval`. See ⟨undefined⟩ [Bison Bridge], page ⟨undefined⟩.

'`--bison-locations, %option bison-locations`'
instruct flex that `GNU bison %locations` are being used. This means `yylex` will be passed an additional parameter, `yylloc`. This option implies `%option bison-bridge`. See ⟨undefined⟩ [Bison Bridge], page ⟨undefined⟩.

'`-L, --noline, %option noline`'
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 in ⟨undefined⟩ [Reporting Bugs], page ⟨undefined⟩).

'`-R, --reentrant, %option reentrant`'
instructs flex to generate a reentrant C scanner. The generated scanner may safely be used in a multi-threaded environment. The API for a reentrant scanner is different than for a non-reentrant scanner see ⟨undefined⟩ [Reentrant],

page ⟨undefined⟩). Because of the API difference between reentrant and non-reentrant `flex` scanners, non-reentrant flex code must be modified before it is suitable for use with this option. This option is not compatible with the '`--c++`' option.

The option '`--reentrant`' does not affect the performance of the scanner.

'`-+, --c++, %option c++`'
specifies that you want flex to generate a C++ scanner class. See ⟨undefined⟩ [Cxx], page ⟨undefined⟩, for details.

'`--array, %option array`'
specifies that you want yytext to be an array instead of a char*

'`--pointer, %option pointer`'
specify that `yytext` should be a `char *`, not an array. This default is `char *`.

'`-PPREFIX, --prefix=PREFIX, %option prefix="PREFIX"`'
changes the default '`yy`' prefix used by `flex` for all globally-visible variable and function names to instead be '`PREFIX`'. For example, '`--prefix=foo`' 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 is a partial list 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
yyalloc
yyrealloc
yyfree
```

(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.

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

'`--nounistd, %option nounistd`'
: suppresses inclusion of the non-ANSI header file '`unistd.h`'. This option is meant to target environments in which '`unistd.h`' does not exist. Be aware that certain options may cause flex to generate code that relies on functions normally found in '`unistd.h`', (e.g. `isatty()`, `read()`.) If you wish to use these functions, you will have to inform your compiler where to find them. See ⟨undefined⟩ [option-always-interactive], page ⟨undefined⟩. See ⟨undefined⟩ [option-read], page ⟨undefined⟩.

'`--yyclass, %option yyclass="NAME"`'
: only applies when generating a C++ scanner (the '`--c++`' 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 ⟨undefined⟩ [Cxx], page ⟨undefined⟩.

16.4 Options for Scanner Speed and Size

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

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

  '`-Ca, --align, %option align`'
  : ("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 quadruple the size of the tables used by your scanner.

  '`-Ce, --ecs, %option ecs`'
  : 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 '`--fast`' flag) should be used. This option cannot be used with '`--c++`'.

'`-Cm, --meta-ecs, %option meta-ecs`'
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, --read, %option read`'
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 ⟨undefined⟩ [Generated Scanner], page ⟨undefined⟩).

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.

'`-f, --full, %option full`'
specifies *fast scanner*. No table compression is done and `stdio` is bypassed. The result is large but fast. This option is equivalent to '`--Cfr`'

'`-F, --fast, %option fast`'
specifies that the *fast* scanner table representation should be used (and `stdio` bypassed). This representation is about as fast as the full table representation '`--full`', 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 '`--fast`'.

This option is equivalent to '`-CFr`' (see below). It cannot be used with '`--c++`'.

16.5 Debugging Options

'`-b, --backup, %option backup`'
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 '`--perf-report`' flag). Only users who wish to squeeze every last cycle out of their scanners need worry about this option. (see ⟨undefined⟩ [Performance], page ⟨undefined⟩).

'`-d, --debug, %option debug`'
makes the generated scanner run in *debug* mode. Whenever a pattern is recognized and the global variable `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.

‘`-p, --perf-report, %option perf-report`’
: 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`, and variable trailing context (see ⟨undefined⟩ [Limitations], page ⟨undefined⟩) entails a substantial performance penalty; use of `yymore()`, the ‘`^`’ operator, and the ‘`--interactive`’ flag entail minor performance penalties.

‘`-s, --nodefault, %option nodefault`’
: 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, --trace, %option trace`’
: 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`.

‘`-w, --nowarn, %option nowarn`’
: suppresses warning messages.

‘`-v, --verbose, %option verbose`’
: 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 ‘`--version`’), and the next line the flags used when generating the scanner, including those that are on by default.

‘`--warn, %option warn`’
: warn about certain things. In particular, if the default rule can be matched but no defualt rule has been given, the flex will warn you. We recommend using this option always.

16.6 Miscellaneous Options

‘`-c`’
: is a do-nothing option included for POSIX compliance.

  generates

‘`-h, -?, --help`’
: generates a “help” summary of `flex`'s options to ‘`stdout`’ and then exits.

‘`-n`’
: is another do-nothing option included only for POSIX compliance.

‘`-V, --version`’
: prints the version number to ‘`stdout`’ and exits.

17 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
arbitrary trailing context

pattern sets that require backing up
%option yylineno
%array

%option interactive
%option always-interactive

@samp{^} beginning-of-line operator
yymore()
```

with the first two 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 you are just putting back some excess text you scanned, use `ss()`.

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

There is one case when `%option yylineno` can be expensive. That is when your patterns match long tokens that could *possibly* contain a newline character. There is no performance penalty for rules that can not possibly match newlines, since flex does not need to check them for newlines. In general, you should avoid rules such as `[^f]+`, which match very long tokens, including newlines, and may possibly match your entire file! A better approach is to separate `[^f]+` into two rules:

```
%option yylineno
%%
    [^f\n]+
    \n+
```

The above scanner does not incur a performance penalty.

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 ⟨undefined⟩ [Limitations], page ⟨undefined⟩).

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 in ⟨undefined⟩ [Matching], page ⟨undefined⟩, 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 per token.

18 Generating C++ Scanners

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

`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 compilation errors (see ⟨undefined⟩ [Reporting Bugs], page ⟨undefined⟩). 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 ⟨undefined⟩ [Scanner Options], page ⟨undefined⟩). Note that you must build the scannerusing `%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-null) and `yyout` to `new_out` (if non-null), 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 in ⟨undefined⟩ [Scanner Options], page ⟨undefined⟩) 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` statements.

`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, but see also ⟨undefined⟩ [Reentrant], page ⟨undefined⟩. 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 == @samp{*})
                {
                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 */ )
    {
```

```
@code{flex}Lexer* 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 <FflexLexer.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.

19 Reentrant C Scanners

`flex` has the ability to generate a reentrant C scanner. This is accomplished by specifying `%option reentrant` (‘`-R`’) The generated scanner is both portable, and safe to use in one or more separate threads of control. The most common use for reentrant scanners is from within multi-threaded applications. Any thread may create and execute a reentrant `flex` scanner without the need for synchronization with other threads.

19.1 Uses for Reentrant Scanners

However, there are other uses for a reentrant scanner. For example, you could scan two or more files simultaneously to implement a `diff` at the token level (i.e., instead of at the character level):

```
/* Example of maintaining more than one active scanner. */

do {
    int tok1, tok2;

    tok1 = yylex( scanner_1 );
    tok2 = yylex( scanner_2 );

    if( tok1 != tok2 )
        printf("Files are different.");

} while ( tok1 && tok2 );
```

Another use for a reentrant scanner is recursion. (Note that a recursive scanner can also be created using a non-reentrant scanner and buffer states. See ⟨undefined⟩ [Multiple Input Buffers], page ⟨undefined⟩.)

The following crude scanner supports the ‘`eval`’ command by invoking another instance of itself.

```
/* Example of recursive invocation. */

%option reentrant

%%
"eval(".+")"  {
                    yyscan_t scanner;
                    YY_BUFFER_STATE buf;

                    yylex_init( &scanner );
                    yytext[yyleng-1] = ' ';

                    buf = yy_scan_string( yytext + 5, scanner );
                    yylex( scanner );
```

```
                    yy_delete_buffer(buf,scanner);
                    yylex_destroy( scanner );
                }
...
%%
```

19.2 An Overview of the Reentrant API

The API for reentrant scanners is different than for non-reentrant scanners. Here is a quick overview of the API:

`%option reentrant` must be specified.

All functions take one additional argument: `yyscanner`

All global variables are replaced by their macro equivalents. (We tell you this because it may be important to you during debugging.)

`yylex_init` and `yylex_destroy` must be called before and after `yylex`, respectively.

Accessor methods (get/set functions) provide access to common `flex` variables.

User-specific data can be stored in `yyextra`.

19.3 Reentrant Example

First, an example of a reentrant scanner:

```
/* This scanner prints "//" comments. */
%option reentrant stack
%x COMMENT
%%
"//"                yy_push_state( COMMENT, yyscanner);
.|\n
<COMMENT>\n         yy_pop_state( yyscanner );
<COMMENT>[^\n]+     fprintf( yyout, "%s\n", yytext);
%%
int main ( int argc, char * argv[] )
{
    yyscan_t scanner;

    yylex_init ( &scanner );
    yylex ( scanner );
    yylex_destroy ( scanner );
return 0;
}
```

19.4 The Reentrant API in Detail

Here are the things you need to do or know to use the reentrant C API of `flex`.

19.4.1 Declaring a Scanner As Reentrant

%option reentrant (–reentrant) must be specified.

Notice that `%option reentrant` is specified in the above example (see ⟨undefined⟩ [Reentrant Example], page ⟨undefined⟩. Had this option not been specified, `flex` would have happily generated a non-reentrant scanner without complaining. You may explicitly specify `%option noreentrant`, if you do *not* want a reentrant scanner, although it is not necessary. The default is to generate a non-reentrant scanner.

19.4.2 The Extra Argument

All functions take one additional argument: `yyscanner`.

Notice that the calls to `yy_push_state` and `yy_pop_state` both have an argument, `yyscanner` , that is not present in a non-reentrant scanner. Here are the declarations of `yy_push_state` and `yy_pop_state` in the generated scanner:

```
static void yy_push_state  ( int new_state , yyscan_t yyscanner ) ;
static void yy_pop_state  ( yyscan_t yyscanner  ) ;
```

Notice that the argument `yyscanner` appears in the declaration of both functions. In fact, all `flex` functions in a reentrant scanner have this additional argument. It is always the last argument in the argument list, it is always of type `yyscan_t` (which is typedef'd to `void *`) and it is always named `yyscanner`. As you may have guessed, `yyscanner` is a pointer to an opaque data structure encapsulating the current state of the scanner. For a list of function declarations, see ⟨undefined⟩ [Reentrant Functions], page ⟨undefined⟩. Note that preprocessor macros, such as `BEGIN`, `ECHO`, and `REJECT`, do not take this additional argument.

19.4.3 Global Variables Replaced By Macros

All global variables in traditional flex have been replaced by macro equivalents.

Note that in the above example, `yyout` and `yytext` are not plain variables. These are macros that will expand to their equivalent lvalue. All of the familiar `flex` globals have been replaced by their macro equivalents. In particular, `yytext`, `yyleng`, `yylineno`, `yyin`, `yyout`, `yyextra`, `yylval`, and `yylloc` are macros. You may safely use these macros in actions as if they were plain variables. We only tell you this so you don't expect to link to these variables externally. Currently, each macro expands to a member of an internal struct, e.g.,

```
#define yytext (((struct yyguts_t*)yyscanner)->yytext_r)
```

One important thing to remember about `yytext` and friends is that `yytext` is not a global variable in a reentrant scanner, you can not access it directly from outside an action or from other functions. You must use an accessor method, e.g., `yyget_text`, to accomplish this. (See below).

19.4.4 Init and Destroy Functions

`yylex_init` and `yylex_destroy` must be called before and after `yylex`, respectively.

```
int yylex_init ( yyscan_t * ptr_yy_globals ) ;
int yylex ( yyscan_t yyscanner ) ;
int yylex_destroy ( yyscan_t yyscanner ) ;
```

The function `yylex_init` must be called before calling any other function. The argument to `yylex_init` is the address of an uninitialized pointer to be filled in by `flex`. The contents of `ptr_yy_globals` need not be initialized, since `flex` will overwrite it anyway. The value stored in `ptr_yy_globals` should thereafter be passed to `yylex()` and **yylex_destroy()**. Flex does not save the argument passed to `yylex_init`, so it is safe to pass the address of a local pointer to `yylex_init`. The function `yylex` should be familiar to you by now. The reentrant version takes one argument, which is the value returned (via an argument) by `yylex_init`. Otherwise, it behaves the same as the non-reentrant version of `yylex`.

`yylex_init` returns 0 (zero) on success, or non-zero on failure, in which case, errno is set to one of the following values:

ENOMEM Memory allocation error. See ⟨undefined⟩ [memory-management], page ⟨undefined⟩.

EINVAL Invalid argument.

The function `yylex_destroy` should be called to free resources used by the scanner. After `yylex_destroy` is called, the contents of `yyscanner` should not be used. Of course, there is no need to destroy a scanner if you plan to reuse it. A `flex` scanner (both reentrant and non-reentrant) may be restarted by calling `yyrestart`.

Below is an example of a program that creates a scanner, uses it, then destroys it when done:

```
int main ()
{
    yyscan_t scanner;
    int tok;

    yylex_init(&scanner);

    while ((tok=yylex()) > 0)
        printf("tok=%d  yytext=%s\n", tok, yyget_text(scanner));

    yylex_destroy(scanner);
    return 0;
}
```

19.4.5 Accessing Variables with Reentrant Scanners

Accessor methods (get/set functions) provide access to common `flex` variables.

Many scanners that you build will be part of a larger project. Portions of your project will need access to `flex` values, such as `yytext`. In a non-reentrant scanner, these values

are global, so there is no problem accessing them. However, in a reentrant scanner, there are no global `flex` values. You can not access them directly. Instead, you must access `flex` values using accessor methods (get/set functions). Each accessor method is named `yyget_NAME` or `yyset_NAME`, where `NAME` is the name of the `flex` variable you want. For example:

```
/* Set the last character of yytext to NULL. */
void chop ( yyscan_t scanner )
{
    int len = yyget_leng( scanner );
    yyget_text( scanner )[len - 1] = '\0';
}
```

The above code may be called from within an action like this:

```
%%
.+\n    { chop( yyscanner );}
```

You may find that `%option header-file` is particularly useful for generating prototypes of all the accessor functions. See ⟨undefined⟩ [option-header], page ⟨undefined⟩.

19.4.6 Extra Data

User-specific data can be stored in `yyextra`.

In a reentrant scanner, it is unwise to use global variables to communicate with or maintain state between different pieces of your program. However, you may need access to external data or invoke external functions from within the scanner actions. Likewise, you may need to pass information to your scanner (e.g., open file descriptors, or database connections). In a non-reentrant scanner, the only way to do this would be through the use of global variables. `Flex` allows you to store arbitrary, "extra" data in a scanner. This data is accessible through the accessor methods `yyget_extra` and `yyset_extra` from outside the scanner, and through the shortcut macro `yyextra` from within the scanner itself. They are defined as follows:

```
#define YY_EXTRA_TYPE  void*
YY_EXTRA_TYPE  yyget_extra ( yyscan_t scanner );
void           yyset_extra ( YY_EXTRA_TYPE arbitrary_data , yyscan_t scanner);
```

By default, `YY_EXTRA_TYPE` is defined as type `void *`. You will have to cast `yyextra` and the return value from `yyget_extra` to the appropriate value each time you access the extra data. To avoid casting, you may override the default type by defining `YY_EXTRA_TYPE` in section 1 of your scanner:

```
/* An example of overriding YY_EXTRA_TYPE. */
%{
#include <sys/stat.h>
#include <unistd.h>
#define YY_EXTRA_TYPE  struct stat*
%}
```

```
%option reentrant
%%

__filesize__     printf( "%ld", yyextra->st_size  );
__lastmod__      printf( "%ld", yyextra->st_mtime );
%%
void scan_file( char* filename )
{
    yyscan_t scanner;
    struct stat buf;

    yylex_init ( &scanner );
    yyset_in( fopen(filename,"r"), scanner );

    stat( filename, &buf);
    yyset_extra( &buf, scanner );
    yylex ( scanner );
    yylex_destroy( scanner );
}
```

19.4.7 About yyscan_t

`yyscan_t` is defined as:

```
typedef void* yyscan_t;
```

It is initialized by `yylex_init()` to point to an internal structure. You should never access this value directly. In particular, you should never attempt to free it (use `yylex_destroy()` instead.)

19.5 Functions and Macros Available in Reentrant C Scanners

The following Functions are available in a reentrant scanner:

```
char *yyget_text ( yyscan_t scanner );
int yyget_leng ( yyscan_t scanner );
FILE *yyget_in ( yyscan_t scanner );
FILE *yyget_out ( yyscan_t scanner );
int yyget_lineno ( yyscan_t scanner );
YY_EXTRA_TYPE yyget_extra ( yyscan_t scanner );
int  yyget_debug ( yyscan_t scanner );

void yyset_debug ( int flag, yyscan_t scanner );
void yyset_in  ( FILE * in_str , yyscan_t scanner );
void yyset_out  ( FILE * out_str , yyscan_t scanner );
void yyset_lineno ( int line_number , yyscan_t scanner );
void yyset_extra ( YY_EXTRA_TYPE user_defined , yyscan_t scanner );
```

There are no “set” functions for yytext and yyleng. This is intentional.

The following Macro shortcuts are available in actions in a reentrant scanner:

```
yytext
yyleng
yyin
yyout
yylineno
yyextra
yy_flex_debug
```

In a reentrant C scanner, support for yylineno is always present (i.e., you may access yylineno), but the value is never modified by `flex` unless `%option yylineno` is enabled. This is to allow the user to maintain the line count independently of `flex`.

The following functions and macros are made available when `%option bison-bridge` (‘`--bison-bridge`’) is specified:

```
YYSTYPE * yyget_lval ( yyscan_t scanner );
void yyset_lval ( YYSTYPE * yylvalp , yyscan_t scanner );
yylval
```

The following functions and macros are made available when `%option bison-locations` (‘`--bison-locations`’) is specified:

```
YYLTYPE *yyget_lloc ( yyscan_t scanner );
void yyset_lloc ( YYLTYPE * yyllocp , yyscan_t scanner );
yylloc
```

Support for yylval assumes that `YYSTYPE` is a valid type. Support for yylloc assumes that `YYSLYPE` is a valid type. Typically, these types are generated by `bison`, and are included in section 1 of the `flex` input.

20 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 both implementations. `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 @code{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. See ⟨undefined⟩ [Reentrant], page ⟨undefined⟩, for `flex`'s reentrant API.

Also note that `flex` C++ scanner classes *are* reentrant, so if using C++ is an option for you, you should use them instead. See ⟨undefined⟩ [Cxx], page ⟨undefined⟩, and ⟨undefined⟩ [Reentrant], page ⟨undefined⟩ 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. The AT&T and POSIX specifications of `lex` interpret '`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 '`-l`' and '`--posix`' options do away with this incompatibility.

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`, `YY_FLEX_MINOR_VERSION` and `YY_FLEX_SUBMINOR_VERSION` indicating which version of `flex` generated the scanner. For example, for the 2.5.22 release, these defines would be 2, 5 and 22 respectively.

If the version of `flex` being used is a beta version, then the symbol `FLEX_BETA` is defined.

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

reentrant C API

multiple actions on a line

almost all of the `flex` command-line options

The feature "multiple actions on a line" 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.

21 Memory Management

This chapter describes how flex handles dynamic memory, and how you can override the default behavior.

21.1 The Default Memory Management

Flex allocates dynamic memory during initialization, and once in a while from within a call to yylex(). Initialization takes place during the first call to yylex(). Thereafter, flex may reallocate more memory if it needs to enlarge a buffer. As of version 2.5.9 Flex will clean up all memory when you call `yylex_destroy` See ⟨undefined⟩ [faq-memory-leak], page ⟨undefined⟩.

Flex allocates dynamic memory for four purposes, listed below[1]

16kB for the input buffer.
: Flex allocates memory for the character buffer used to perform pattern matching. Flex must read ahead from the input stream and store it in a large character buffer. This buffer is typically the largest chunk of dynamic memory flex consumes. This buffer will grow if necessary, doubling the size each time. Flex frees this memory when you call yylex_destroy(). The default size of this buffer (16384 bytes) is almost always too large. The ideal size for this buffer is the length of the longest token expected. Flex will allocate a few extra bytes for housekeeping.

16kb for the REJECT state. This will only be allocated if you use REJECT.
: The size is the same as the input buffer, so if you override the size of the input buffer, then you automatically override the size of this buffer as well.

100 bytes for the start condition stack.
: Flex allocates memory for the start condition stack. This is the stack used for pushing start states, i.e., with yy_push_state(). It will grow if necessary. Since the states are simply integers, this stack doesn't consume much memory. This stack is not present if `%option stack` is not specified. You will rarely need to tune this buffer. The ideal size for this stack is the maximum depth expected. The memory for this stack is automatically destroyed when you call yylex_destroy(). See ⟨undefined⟩ [option-stack], page ⟨undefined⟩.

40 bytes for each YY_BUFFER_STATE.
: Flex allocates memory for each YY_BUFFER_STATE. The buffer state itself is about 40 bytes, plus an additional large character buffer (described above.) The initial buffer state is created during initialization, and with each call to yy_create_buffer(). You can't tune the size of this, but you can tune the character buffer as described above. Any buffer state that you explicitly create by calling yy_create_buffer() is *NOT* destroyed automatically. You must call yy_delete_buffer() to free the memory. The exception to this rule is that flex will delete the current buffer automatically when you call yylex_destroy(). If

[1] The quantities given here are approximate, and may vary due to host architecture, compiler configuration, or due to future enhancements to flex.

you delete the current buffer, be sure to set it to NULL. That way, flex will not try to delete the buffer a second time (possibly crashing your program!) At the time of this writing, flex does not provide a growable stack for the buffer states. You have to manage that yourself. See ⟨undefined⟩ [Multiple Input Buffers], page ⟨undefined⟩.

84 bytes for the reentrant scanner guts
: Flex allocates about 84 bytes for the reentrant scanner structure when you call yylex_init(). It is destroyed when the user calls yylex_destroy().

21.2 Overriding The Default Memory Management

Flex calls the functions `yyalloc`, `yyrealloc`, and `yyfree` when it needs to allocate or free memory. By default, these functions are wrappers around the standard C functions, `malloc`, `realloc`, and `free`, respectively. You can override the default implementations by telling flex that you will provide your own implementations.

To override the default implementations, you must do two things:

1. Suppress the default implementations by specifying one or more of the following options:

   ```
   %option noyyalloc
   %option noyyrealloc
   %option noyyfree.
   ```

2. Provide your own implementation of the following functions:[2]

   ```
   // For a non-reentrant scanner
   void * yyalloc (size_t bytes);
   void * yyrealloc (void * ptr, size_t bytes);
   void   yyfree (void * ptr);

   // For a reentrant scanner
   void * yyalloc (size_t bytes, void * yyscanner);
   void * yyrealloc (void * ptr, size_t bytes, void * yyscanner);
   void   yyfree (void * ptr, void * yyscanner);
   ```

In the following example, we will override all three memory routines. We assume that there is a custom allocator with garbage collection. In order to make this example interesting, we will use a reentrant scanner, passing a pointer to the custom allocator through `yyextra`.

```
%{
#include "some_allocator.h"
%}

/* Suppress the default implementations. */
%option noyyalloc noyyrealloc noyyfree
```

[2] It is not necessary to override all (or any) of the memory management routines. You may, for example, override `yyrealloc`, but not `yyfree` or `yyalloc`.

```
%option reentrant

/* Initialize the allocator. */
#define YY_EXTRA_TYPE  struct allocator*
#define YY_USER_INIT  yyextra = allocator_create();

%%
.|\n   ;
%%

/* Provide our own implementations. */
void * yyalloc (size_t bytes, void* yyscanner) {
    return allocator_alloc (yyextra, bytes);
}

void * yyrealloc (void * ptr, size_t bytes, void* yyscanner) {
    return allocator_realloc (yyextra, bytes);
}

void yyfree (void * ptr, void * yyscanner) {
    /* Do nothing -- we leave it to the garbage collector. */
}
```

21.3 A Note About yytext And Memory

When flex finds a match, `yytext` points to the first character of the match in the input buffer. The string itself is part of the input buffer, and is *NOT* allocated separately. The value of yytext will be overwritten the next time yylex() is called. In short, the value of yytext is only valid from within the matched rule's action.

Often, you want the value of yytext to persist for later processing, i.e., by a parser with non-zero lookahead. In order to preserve yytext, you will have to copy it with strdup() or a similar function. But this introduces some headache because your parser is now responsible for freeing the copy of yytext. If you use a yacc or bison parser, (commonly used with flex), you will discover that the error recovery mechanisms can cause memory to be leaked.

To prevent memory leaks from strdup'd yytext, you will have to track the memory somehow. Our experience has shown that a garbage collection mechanism or a pooled memory mechanism will save you a lot of grief when writing parsers.

22 Serialized Tables

A `flex` scanner has the ability to save the DFA tables to a file, and load them at runtime when needed. The motivation for this feature is to reduce the runtime memory footprint. Traditionally, these tables have been compiled into the scanner as C arrays, and are sometimes quite large. Since the tables are compiled into the scanner, the memory used by the tables can never be freed. This is a waste of memory, especially if an application uses several scanners, but none of them at the same time.

The serialization feature allows the tables to be loaded at runtime, before scanning begins. The tables may be discarded when scanning is finished.

22.1 Creating Serialized Tables

You may create a scanner with serialized tables by specifying:

```
    %option tables-file=FILE
or
    --tables-file=FILE
```

These options instruct flex to save the DFA tables to the file *FILE*. The tables will *not* be embedded in the generated scanner. The scanner will not function on its own. The scanner will be dependent upon the serialized tables. You must load the tables from this file at runtime before you can scan anything.

If you do not specify a filename to `--tables-file`, the tables will be saved to '`lex.yy.tables`', where '`yy`' is the appropriate prefix.

If your project uses several different scanners, you can concatenate the serialized tables into one file, and flex will find the correct set of tables, using the scanner prefix as part of the lookup key. An example follows:

```
$ flex --tables-file --prefix=cpp cpp.l
$ flex --tables-file --prefix=c   c.l
$ cat lex.cpp.tables lex.c.tables  >  all.tables
```

The above example created two scanners, '`cpp`', and '`c`'. Since we did not specify a filename, the tables were serialized to '`lex.c.tables`' and '`lex.cpp.tables`', respectively. Then, we concatenated the two files together into '`all.tables`', which we will distribute with our project. At runtime, we will open the file and tell flex to load the tables from it. Flex will find the correct tables automatically. (See next section).

22.2 Loading and Unloading Serialized Tables

If you've built your scanner with `%option tables-file`, then you must load the scanner tables at runtime. This can be accomplished with the following function:

`int yytables_fload` (`FILE*` *fp* [, `yyscan_t` *scanner*]) Function

Locates scanner tables in the stream pointed to by *fp* and loads them. Memory for the tables is allocated via `yyalloc`. You must call this function before the first call to `yylex`. The argument *scanner* only appears in the reentrant scanner. This function returns '`0`' (zero) on success, or non-zero on error.

The loaded tables are **not** automatically destroyed (unloaded) when you call `yylex_destroy`. The reason is that you may create several scanners of the same type (in a reentrant scanner), each of which needs access to these tables. To avoid a nasty memory leak, you must call the following function:

`int yytables_destroy` ([`yyscan_t` *scanner*]) Function

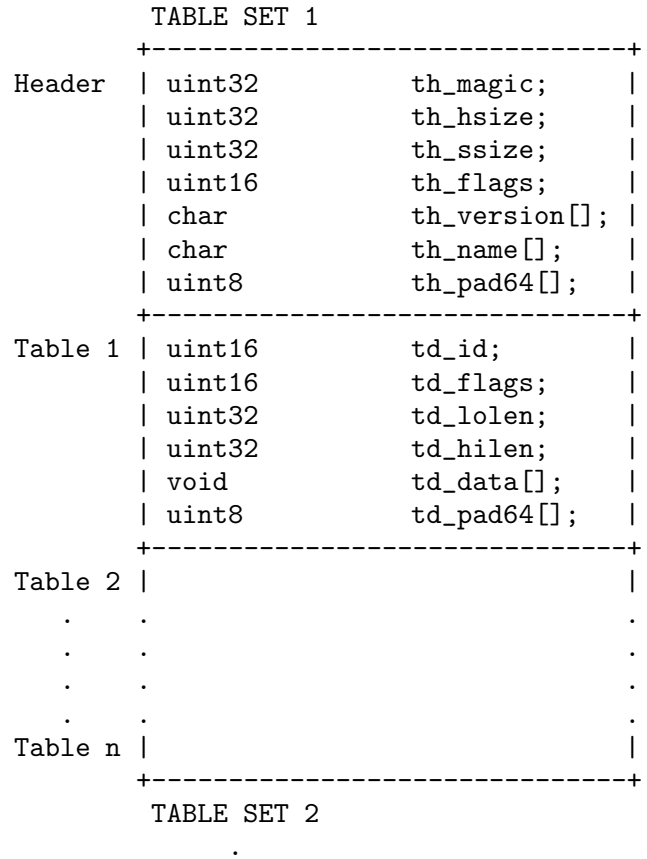
Unloads the scanner tables. The tables must be loaded again before you can scan any more data. The argument *scanner* only appears in the reentrant scanner. This function returns '`0`' (zero) on success, or non-zero on error.

The functions `yytables_fload` and `yytables_destroy` are not thread-safe. You must ensure that these functions are called exactly once (for each scanner type) in a threaded program, before any thread calls `yylex`. After the tables are loaded, they are never written to, and no thread protection is required thereafter – until you destroy them.

22.3 Tables File Format

This section defines the file format of serialized `flex` tables.

The tables format allows for one or more sets of tables to be specified, where each set corresponds to a given scanner. Scanners are indexed by name, as described below. The file format is as follows:



```
.
.
TABLE SET N
```

The above diagram shows that a complete set of tables consists of a header followed by multiple individual tables. Furthermore, multiple complete sets may be present in the same file, each set with its own header and tables. The sets are contiguous in the file. The only way to know if another set follows is to check the next four bytes for the magic number (or check for EOF). The header and tables sections are padded to 64-bit boundaries. Below we describe each field in detail. This format does not specify how the scanner will expand the given data, i.e., data may be serialized as int8, but expanded to an int32 array at runtime. This is to reduce the size of the serialized data where possible. Remember, *all integer values are in network byte order.*

Fields of a table header:

`th_magic` Magic number, always 0xF13C57B1.

`th_hsize` Size of this entire header, in bytes, including all fields plus any padding.

`th_ssize` Size of this entire set, in bytes, including the header, all tables, plus any padding.

`th_flags` Bit flags for this table set. Currently unused.

`th_version[]`
: Flex version in NULL-termninated string format. e.g., '`2.5.13a`'. This is the version of flex that was used to create the serialized tables.

`th_name[]`
: Contains the name of this table set. The default is '`yytables`', and is prefixed accordingly, e.g., '`footables`'. Must be NULL-terminated.

`th_pad64[]`
: Zero or more NULL bytes, padding the entire header to the next 64-bit boundary as calculated from the beginning of the header.

Fields of a table:

`td_id` Specifies the table identifier. Possible values are:

`YYTD_ID_ACCEPT (0x01)`
: `yy_accept`

`YYTD_ID_BASE (0x02)`
: `yy_base`

`YYTD_ID_CHK (0x03)`
: `yy_chk`

`YYTD_ID_DEF (0x04)`
: `yy_def`

`YYTD_ID_EC (0x05)`
: `yy_ec`

`YYTD_ID_META (0x06)`
: `yy_meta`

`YYTD_ID_NUL_TRANS (0x07)`
: `yy_NUL_trans`

`YYTD_ID_NXT (0x08)`
: `yy_nxt`. This array may be two dimensional. See the `td_hilen` field below.

`YYTD_ID_RULE_CAN_MATCH_EOL (0x09)`
: `yy_rule_can_match_eol`

`YYTD_ID_START_STATE_LIST (0x0A)`
: `yy_start_state_list`. This array is handled specially because it is an array of pointers to structs. See the `td_flags` field below.

`YYTD_ID_TRANSITION (0x0B)`
: `yy_transition`. This array is handled specially because it is an array of structs. See the `td_lolen` field below.

`YYTD_ID_ACCLIST (0x0C)`
: `yy_acclist`

`td_flags` Bit flags describing how to interpret the data in `td_data`. The data arrays are one-dimensional by default, but may be two dimensional as specified in the `td_hilen` field.

`YYTD_DATA8 (0x01)`
: The data is serialized as an array of type int8.

`YYTD_DATA16 (0x02)`
: The data is serialized as an array of type int16.

`YYTD_DATA32 (0x04)`
: The data is serialized as an array of type int32.

`YYTD_PTRANS (0x08)`
: The data is a list of indexes of entries in the expanded `yy_transition` array. Each index should be expanded to a pointer to the corresponding entry in the `yy_transition` array. We count on the fact that the `yy_transition` array has already been seen.

`YYTD_STRUCT (0x10)`
: The data is a list of yy_trans_info structs, each of which consists of two integers. There is no padding between struct elements or between structs. The type of each member is determined by the `YYTD_DATA*` bits.

`td_lolen` Specifies the number of elements in the lowest dimension array. If this is a one-dimensional array, then it is simply the number of elements in this array. The element size is determined by the `td_flags` field.

`td_hilen` If `td_hilen` is non-zero, then the data is a two-dimensional array. Otherwise, the data is a one-dimensional array. `td_hilen` contains the number of elements in the higher dimensional array, and `td_lolen` contains the number of elements in the lowest dimension.

Conceptually, `td_data` is either `sometype td_data[td_lolen]`, or `sometype td_data[td_hilen][td_lolen]`, where `sometype` is specified by the `td_flags` field. It is possible for both `td_lolen` and `td_hilen` to be zero, in which case `td_data` is a zero length array, and no data is loaded, i.e., this table is simply skipped. Flex does not currently generate tables of zero length.

`td_data[]`

The table data. This array may be a one- or two-dimensional array, of type `int8`, `int16`, `int32`, `struct yy_trans_info`, or `struct yy_trans_info*`, depending upon the values in the `td_flags`, `td_lolen`, and `td_hilen` fields.

`td_pad64[]`

Zero or more NULL bytes, padding the entire table to the next 64-bit boundary as calculated from the beginning of this table.

23 Diagnostics

The following is a list of `flex` diagnostic messages:

'`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, ⟨undefined⟩ [Scanner Options], page ⟨undefined⟩, 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 a 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 <> construct!`' you listed more start conditions in a `<>` construct than exist (so you must have listed at least one of them twice).

24 Limitations

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, **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.

25 Additional Reading

You may wish to read more about the following programs:

lex

yacc

sed

awk

The following books may contain material of interest:

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).

FAQ

From time to time, the `flex` maintainer receives certain questions. Rather than repeat answers to well-understood problems, we publish them here.

When was flex born?

Vern Paxson took over the *Software Tools* lex project from Jef Poskanzer in 1982. At that point it was written in Ratfor. Around 1987 or so, Paxson translated it into C, and a legend was born :-).

How do I expand \ escape sequences in C-style quoted strings?

A key point when scanning quoted strings is that you cannot (easily) write a single rule that will precisely match the string if you allow things like embedded escape sequences and newlines. If you try to match strings with a single rule then you'll wind up having to rescan the string anyway to find any escape sequences.

Instead you can use exclusive start conditions and a set of rules, one for matching non-escaped text, one for matching a single escape, one for matching an embedded newline, and one for recognizing the end of the string. Each of these rules is then faced with the question of where to put its intermediary results. The best solution is for the rules to append their local value of `yytext` to the end of a "string literal" buffer. A rule like the escape-matcher will append to the buffer the meaning of the escape sequence rather than the literal text in `yytext`. In this way, `yytext` does not need to be modified at all.

Why do flex scanners call fileno if it is not ANSI compatible?

Flex scanners call `fileno()` in order to get the file descriptor corresponding to `yyin`. The file descriptor may be passed to `isatty()` or `read()`, depending upon which `%options` you specified. If your system does not have `fileno()` support, to get rid of the `read()` call, do not specify `%option read`. To get rid of the `isatty()` call, you must specify one of `%option always-interactive` or `%option never-interactive`.

Does flex support recursive pattern definitions?

e.g.,

```
%%
block   "{"({block}|{statement})*"}"
```

No. You cannot have recursive definitions. The pattern-matching power of regular expressions in general (and therefore flex scanners, too) is limited. In particular, regular expressions cannot "balance" parentheses to an arbitrary degree. For example, it's impossible to write a regular expression that matches all strings containing the same number of '`{`'s as '`}`'s. For more powerful pattern matching, you need a parser, such as *GNU bison*.

How do I skip huge chunks of input (tens of megabytes) while using flex?

Use `fseek()` (or `lseek()`) to position yyin, then call `yyrestart()`.

Flex is not matching my patterns in the same order that I defined them.

`flex` picks the rule that matches the most text (i.e., the longest possible input string). This is because `flex` uses an entirely different matching technique ("deterministic finite automata") that actually does all of the matching simultaneously, in parallel. (Seems impossible, but it's actually a fairly simple technique once you understand the principles.)

A side-effect of this parallel matching is that when the input matches more than one rule, `flex` scanners pick the rule that matched the *most* text. This is explained further in the manual, in the section See ⟨undefined⟩ [Matching], page ⟨undefined⟩.

If you want `flex` to choose a shorter match, then you can work around this behavior by expanding your short rule to match more text, then put back the extra:

```
data_.*         yyless( 5 ); BEGIN BLOCKIDSTATE;
```

Another fix would be to make the second rule active only during the `<BLOCKIDSTATE>` start condition, and make that start condition exclusive by declaring it with `%x` instead of `%s`.

A final fix is to change the input language so that the ambiguity for '`data_`' is removed, by adding characters to it that don't match the identifier rule, or by removing characters (such as '`_`') from the identifier rule so it no longer matches '`data_`'. (Of course, you might also not have the option of changing the input language.)

My actions are executing out of order or sometimes not at all.

Most likely, you have (in error) placed the opening '`{`' of the action block on a different line than the rule, e.g.,

```
^(foo|bar)
{  <<<--- WRONG!

}
```

`flex` requires that the opening '`{`' of an action associated with a rule begin on the same line as does the rule. You need instead to write your rules as follows:

```
^(foo|bar)   {  // CORRECT!

}
```

How can I have multiple input sources feed into the same scanner at the same time?

If . . .

> your scanner is free of backtracking (verified using `flex`'s '`-b`' flag),
>
> AND you run your scanner interactively ('`-I`' option; default unless using special table compression options),
>
> AND you feed it one character at a time by redefining `YY_INPUT` to do so,

then every time it matches a token, it will have exhausted its input buffer (because the scanner is free of backtracking). This means you can safely use `select()` at the point and only call `yylex()` for another token if `select()` indicates there's data available.

That is, move the `select()` out from the input function to a point where it determines whether `yylex()` gets called for the next token.

With this approach, you will still have problems if your input can arrive piecemeal; `select()` could inform you that the beginning of a token is available, you call `yylex()` to get it, but it winds up blocking waiting for the later characters in the token.

Here's another way: Move your input multiplexing inside of `YY_INPUT`. That is, whenever `YY_INPUT` is called, it `select()`'s to see where input is available. If input is available for the scanner, it reads and returns the next byte. If input is available from another source, it calls whatever function is responsible for reading from that source. (If no input is available, it blocks until some input is available.) I've used this technique in an interpreter I wrote that both reads keyboard input using a `flex` scanner and IPC traffic from sockets, and it works fine.

Can I build nested parsers that work with the same input file?

This is not going to work without some additional effort. The reason is that `flex` block-buffers the input it reads from `yyin`. This means that the "outermost" `yylex()`, when called, will automatically slurp up the first 8K of input available on yyin, and subsequent calls to other `yylex()`'s won't see that input. You might be tempted to work around this problem by redefining `YY_INPUT` to only return a small amount of text, but it turns out that that approach is quite difficult. Instead, the best solution is to combine all of your scanners into one large scanner, using a different exclusive start condition for each.

How can I match text only at the end of a file?

There is no way to write a rule which is "match this text, but only if it comes at the end of the file". You can fake it, though, if you happen to have a character lying around that you don't allow in your input. Then you redefine `YY_INPUT` to call your own routine which, if it sees an '`EOF`', returns the magic character first (and remembers to return a real `EOF` next time it's called). Then you could write:

```
COMMENT>(.|\n)*{EOF_CHAR}    /* saw comment at EOF */
```

How can I make REJECT cascade across start condition boundaries?

You can do this as follows. Suppose you have a start condition '`A`', and after exhausting all of the possible matches in '`<A>`', you want to try matches in '`<INITIAL>`'. Then you could use the following:

```
%x A
%%
A>rule_that_is_long    ...; REJECT;
A>rule                 ...; REJECT; /* shorter rule */
A>etc.
...
A>.|\n  {
/* Shortest and last rule in <A>, so
* cascaded REJECT's will eventually
* wind up matching this rule.  We want
* to now switch to the initial state
* and try matching from there instead.
*/
yyless(0);    /* put back matched text */
BEGIN(INITIAL);
}
```

Why can't I use fast or full tables with interactive mode?

One of the assumptions flex makes is that interactive applications are inherently slow (they're waiting on a human after all). It has to do with how the scanner detects that it must be finished scanning a token. For interactive scanners, after scanning each character the current state is looked up in a table (essentially) to see whether there's a chance of another input character possibly extending the length of the match. If not, the scanner halts. For non-interactive scanners, the end-of-token test is much simpler, basically a compare with 0, so no memory bus cycles. Since the test occurs in the innermost scanning loop, one would like to make it go as fast as possible.

Still, it seems reasonable to allow the user to choose to trade off a bit of performance in this area to gain the corresponding flexibility. There might be another reason, though, why fast scanners don't support the interactive option.

How much faster is -F or -f than -C?

Much faster (factor of 2-3).

If I have a simple grammar can't I just parse it with flex?

Is your grammar recursive? That's almost always a sign that you're better off using a parser/scanner rather than just trying to use a scanner alone.

Why doesn't yyrestart() set the start state back to INITIAL?

There are two reasons. The first is that there might be programs that rely on the start state not changing across file changes. The second is that beginning with `flex` version 2.4, use of `yyrestart()` is no longer required, so fixing the problem there doesn't solve the more general problem.

How can I match C-style comments?

You might be tempted to try something like this:

```
/*".*"*/"       // WRONG!
```

or, worse, this:

```
/*"(.|\n)"*/"   // WRONG!
```

The above rules will eat too much input, and blow up on things like:

```
/* a comment */ do_my_thing( "oops */" );
```

Here is one way which allows you to track line information:

```
INITIAL>{
/*"              BEGIN(IN_COMMENT);
}
IN_COMMENT>{
*/"      BEGIN(INITIAL);
[^*\n]+   // eat comment in chunks
*"       // eat the lone star
\n        yylineno++;
}
```

The '.' isn't working the way I expected.

Here are some tips for using '`.`':

A common mistake is to place the grouping parenthesis AFTER an operator, when you really meant to place the parenthesis BEFORE the operator, e.g., you probably want this `(foo|bar)+` and NOT this `(foo|bar+)`.

The first pattern matches the words '`foo`' or '`bar`' any number of times, e.g., it matches the text '`barfoofoobarfoo`'. The second pattern matches a single instance of `foo` or a single instance of `bar` followed by one or more '`r`'s, e.g., it matches the text `barrrr` .

A '`.`' inside '`[]`''s just means a literal'`.`' (period), and NOT "any character except newline".

Remember that '`.`' matches any character EXCEPT '`\n`' (and '`EOF`'). If you really want to match ANY character, including newlines, then use `(.|\n)` Beware that the regex `(.|\n)+` will match your entire input!

Finally, if you want to match a literal '`.`' (a period), then use '`[.]`' or '`"."`'

Can I get the flex manual in another format?

The `flex` source distribution includes a texinfo manual. You are free to convert that texinfo into whatever format you desire. The `texinfo` package includes tools for conversion to a number of formats.

Does there exist a "faster" NDFA->DFA algorithm?

There's no way around the potential exponential running time - it can take you exponential time just to enumerate all of the DFA states. In practice, though, the running time is closer to linear, or sometimes quadratic.

How does flex compile the DFA so quickly?

There are two big speed wins that `flex` uses:

1. It analyzes the input rules to construct equivalence classes for those characters that always make the same transitions. It then rewrites the NFA using equivalence classes for transitions instead of characters. This cuts down the NFA->DFA computation time dramatically, to the point where, for uncompressed DFA tables, the DFA generation is often I/O bound in writing out the tables.
2. It maintains hash values for previously computed DFA states, so testing whether a newly constructed DFA state is equivalent to a previously constructed state can be done very quickly, by first comparing hash values.

How can I use more than 8192 rules?

`Flex` is compiled with an upper limit of 8192 rules per scanner. If you need more than 8192 rules in your scanner, you'll have to recompile `flex` with the following changes in '`flexdef.h`':

```
        #define YY_TRAILING_MASK 0x2000
        #define YY_TRAILING_HEAD_MASK 0x4000
    --
        #define YY_TRAILING_MASK 0x20000000
        #define YY_TRAILING_HEAD_MASK 0x40000000
```

This should work okay as long as your C compiler uses 32 bit integers. But you might want to think about whether using such a huge number of rules is the best way to solve your problem.

The following may also be relevant:

With luck, you should be able to increase the definitions in flexdef.h for:

```
#define JAMSTATE -32766 /* marks a reference to the state that always jams */
#define MAXIMUM_MNS 31999
#define BAD_SUBSCRIPT -32767
```

recompile everything, and it'll all work. Flex only has these 16-bit-like values built into it because a long time ago it was developed on a machine with 16-bit ints. I've given this

advice to others in the past but haven't heard back from them whether it worked okay or not...

How do I abandon a file in the middle of a scan and switch to a new file?

Just call `yyrestart(newfile)`. Be sure to reset the start state if you want a "fresh start, since `yyrestart` does NOT reset the start state back to `INITIAL`.

How do I execute code only during initialization (only before the first scan)?

You can specify an initial action by defining the macro `YY_USER_INIT` (though note that `yyout` may not be available at the time this macro is executed). Or you can add to the beginning of your rules section:

```
%%
/* Must be indented! */
static int did_init = 0;

if ( ! did_init ){
do_my_init();
did_init = 1;
}
```

How do I execute code at termination?

You can specify an action for the `<<EOF>>` rule.

Where else can I find help?

You can find the flex homepage on the web at `http://lex.sourceforge.net/`. See that page for details about flex mailing lists as well.

Can I include comments in the "rules" section of the file?

Yes, just about anywhere you want to. See the manual for the specific syntax.

I get an error about undefined yywrap().

You must supply a `yywrap()` function of your own, or link to '`libfl.a`' (which provides one), or use

```
%option noyywrap
```

in your source to say you don't want a `yywrap()` function.

How can I change the matching pattern at run time?

You can't, it's compiled into a static table when flex builds the scanner.

How can I expand macros in the input?

The best way to approach this problem is at a higher level, e.g., in the parser.

However, you can do this using multiple input buffers.

```
%%
macro/[a-z]+    {
/* Saw the macro "macro" followed by extra stuff. */
main_buffer = YY_CURRENT_BUFFER;
expansion_buffer = yy_scan_string(expand(yytext));
yy_switch_to_buffer(expansion_buffer);
}

<EOF>>{
if ( expansion_buffer )
{
// We were doing an expansion, return to where
// we were.
yy_switch_to_buffer(main_buffer);
yy_delete_buffer(expansion_buffer);
expansion_buffer = 0;
}
else
yyterminate();
}
```

You probably will want a stack of expansion buffers to allow nested macros. From the above though hopefully the idea is clear.

How can I build a two-pass scanner?

One way to do it is to filter the first pass to a temporary file, then process the temporary file on the second pass. You will probably see a performance hit, do to all the disk I/O.

When you need to look ahead far forward like this, it almost always means that the right solution is to build a parse tree of the entire input, then walk it after the parse in order to generate the output. In a sense, this is a two-pass approach, once through the text and once through the parse tree, but the performance hit for the latter is usually an order of magnitude smaller, since everything is already classified, in binary format, and residing in memory.

How do I match any string not matched in the preceding rules?

One way to assign precedence, is to place the more specific rules first. If two rules would match the same input (same sequence of characters) then the first rule listed in the `flex` input wins. e.g.,

```
%%
foo[a-zA-Z_]+    return FOO_ID;
bar[a-zA-Z_]+    return BAR_ID;
[a-zA-Z_]+       return GENERIC_ID;
```

Note that the rule `[a-zA-Z_]+` must come *after* the others. It will match the same amount of text as the more specific rules, and in that case the `flex` scanner will pick the first rule listed in your scanner as the one to match.

I am trying to port code from AT&T lex that uses yysptr and yysbuf.

Those are internal variables pointing into the AT&T scanner's input buffer. I imagine they're being manipulated in user versions of the `input()` and `unput()` functions. If so, what you need to do is analyze those functions to figure out what they're doing, and then replace `input()` with an appropriate definition of `YY_INPUT`. You shouldn't need to (and must not) replace `flex`'s `unput()` function.

Is there a way to make flex treat NULL like a regular character?

Yes, '`\0`' and '`\x00`' should both do the trick. Perhaps you have an ancient version of `flex`. The latest release is version 2.5.31.

Whenever flex can not match the input it says "flex scanner jammed".

You need to add a rule that matches the otherwise-unmatched text. e.g.,

```
%option yylineno
%%
[[a bunch of rules here]]

.       printf("bad input character '%s' at line %d\n", yytext, yylineno);
```

See `%option default` for more information.

Why doesn't flex have non-greedy operators like perl does?

A DFA can do a non-greedy match by stopping the first time it enters an accepting state, instead of consuming input until it determines that no further matching is possible (a "jam" state). This is actually easier to implement than longest leftmost match (which flex does).

But it's also much less useful than longest leftmost match. In general, when you find yourself wishing for non-greedy matching, that's usually a sign that you're trying to make the scanner do some parsing. That's generally the wrong approach, since it lacks the power to do a decent job. Better is to either introduce a separate parser, or to split the scanner into multiple scanners using (exclusive) start conditions.

You might have a separate start state once you've seen the '`BEGIN`'. In that state, you might then have a regex that will match '`END`' (to kick you out of the state), and perhaps '`(.|\n)`' to get a single character within the chunk ...

This approach also has much better error-reporting properties.

Memory leak - 16386 bytes allocated by malloc.

UPDATED 2002-07-10: As of `flex` version 2.5.9, this leak means that you did not call `yylex_destroy()`. If you are using an earlier version of `flex`, then read on.

The leak is about 16426 bytes. That is, (8192 * 2 + 2) for the read-buffer, and about 40 for `struct yy_buffer_state` (depending upon alignment). The leak is in the non-reentrant C scanner only (NOT in the reentrant scanner, NOT in the C++ scanner). Since `flex` doesn't know when you are done, the buffer is never freed.

However, the leak won't multiply since the buffer is reused no matter how many times you call `yylex()`.

If you want to reclaim the memory when you are completely done scanning, then you might try this:

```
/* For non-reentrant C scanner only. */
yy_delete_buffer(YY_CURRENT_BUFFER);
yy_init = 1;
```

Note: `yy_init` is an "internal variable", and hasn't been tested in this situation. It is possible that some other globals may need resetting as well.

How do I track the byte offset for lseek()?

```
We thought that it would be possible to have this number through the
evaluation of the following expression:

seek_position = (no_buffers)*YY_READ_BUF_SIZE + yy_c_buf_p - YY_CURRENT_BUFFER->yy_
```

While this is the right idea, it has two problems. The first is that it's possible that `flex` will request less than `YY_READ_BUF_SIZE` during an invocation of `YY_INPUT` (or that your input source will return less even though `YY_READ_BUF_SIZE` bytes were requested). The second problem is that when refilling its internal buffer, `flex` keeps some characters from the previous buffer (because usually it's in the middle of a match, and needs those characters to construct `yytext` for the match once it's done). Because of this, `yy_c_buf_p - YY_CURRENT_BUFFER->yy_ch_buf` won't be exactly the number of characters already read from the current buffer.

An alternative solution is to count the number of characters you've matched since starting to scan. This can be done by using `YY_USER_ACTION`. For example,

```
#define YY_USER_ACTION num_chars += yyleng;
```

(You need to be careful to update your bookkeeping if you use `yymore()`, `yyless()`, `unput()`, or `input()`.)

How do I use my own I/O classes in a C++ scanner?

When the flex C++ scanning class rewrite finally happens, then this sort of thing should become much easier.

You can do this by passing the various functions (such as `LexerInput()` and `LexerOutput()`) NULL `iostream*`'s, and then dealing with your own I/O classes surreptitiously (i.e., stashing them in special member variables). This works because the only assumption about the lexer regarding what's done with the iostream's is that they're ultimately passed to `LexerInput()` and `LexerOutput`, which then do whatever is necessary with them.

How do I skip as many chars as possible?

How do I skip as many chars as possible – without interfering with the other patterns?

In the example below, we want to skip over characters until we see the phrase "endskip". The following will *NOT* work correctly (do you see why not?)

```
/* INCORRECT SCANNER */
%x SKIP
%%
INITIAL>startskip   BEGIN(SKIP);
...
SKIP>"endskip"       BEGIN(INITIAL);
SKIP>.*             ;
```

The problem is that the pattern .* will eat up the word "endskip." The simplest (but slow) fix is:

```
SKIP>"endskip"      BEGIN(INITIAL);
SKIP>.              ;
```

The fix involves making the second rule match more, without making it match "endskip" plus something else. So for example:

```
SKIP>"endskip"     BEGIN(INITIAL);
SKIP>[^e]+         ;
SKIP>.                ;/* so you eat up e's, too */
```

deleteme00

```
QUESTION:
When was flex born?
```

Vern Paxson took over
the Software Tools lex project from Jef Poskanzer in 1982. At that point it was written in Ratfor. Around 1987 or so, Paxson translated it into C, and a legend was born :-).

Are certain equivalent patterns faster than others?

To: Adoram Rogel <adoram@orna.hybridge.com>
Subject: Re: Flex 2.5.2 performance questions
In-reply-to: Your message of Wed, 18 Sep 96 11:12:17 EDT.
Date: Wed, 18 Sep 96 10:51:02 PDT
From: Vern Paxson <vern>

[Note, the most recent flex release is 2.5.4, which you can get from ftp.ee.lbl.gov. It has bug fixes over 2.5.2 and 2.5.3.]

1. Using the pattern
   ([Ff](oot)?)?[Nn](ote)?(\.)?
   instead of
   (((F|f)oot(N|n)ote)|((N|n)ote)|((N|n)\.)|((F|f)(N|n)(\.)))
   (in a very complicated flex program) caused the program to slow from 300K+/min to 100K/min (no other changes were done).

These two are not equivalent. For example, the first can match "footnote." but the second can only match "footnote". This is almost certainly the cause in the discrepancy - the slower scanner run is matching more tokens, and/or having to do more backing up.

2. Which of these two are better: [Ff]oot or (F|f)oot ?

From a performance point of view, they're equivalent (modulo presumably minor effects such as memory cache hit rates; and the presence of trailing context, see below). From a space point of view, the first is slightly preferable.

3. I have a pattern that look like this:
   pats {p1}|{p2}|{p3}|...|{p50} (50 patterns ORd)

   running yet another complicated program that includes the following rule:
   <snext>{and}/{no4}{bb}{pats}

   gets me to "too complicated - over 32,000 states"...

I can't tell from this example whether the trailing context is variable-length or fixed-length (it could be the latter if {and} is fixed-length). If it's variable length, which flex -p will tell you, then this reflects a basic performance problem, and if you can eliminate it by restructuring your

```
scanner, you will see significant improvement.

    so I divided {pats} to {pats1}, {pats2},..., {pats5} each consists of about
    10 patterns and changed the rule to be 5 rules.
    This did compile, but what is the rule of thumb here ?

The rule is to avoid trailing context other than fixed-length, in which for
a/b, either the 'a' pattern or the 'b' pattern have a fixed length.  Use
of the '|' operator automatically makes the pattern variable length, so in
this case '[Ff]oot' is preferred to '(F|f)oot'.

 4. I changed a rule that looked like this:
    <snext8>{and}{bb}/{ROMAN}[^A-Za-z] { BEGIN...

    to the next 2 rules:
    <snext8>{and}{bb}/{ROMAN}[A-Za-z] { ECHO;}
    <snext8>{and}{bb}/{ROMAN}         { BEGIN...

    Again, I understand the using [^...] will cause a great performance loss

Actually, it doesn't cause any sort of performance loss.  It's a surprising
fact about regular expressions that they always match in linear time
regardless of how complex they are.

    but are there any specific rules about it ?

See the "Performance Considerations" section of the man page, and also
the example in MISC/fastwc/.

                Vern
```

Is backing up a big deal?

```
To: Adoram Rogel <adoram@hybridge.com>
Subject: Re: Flex 2.5.2 performance questions
In-reply-to: Your message of Thu, 19 Sep 96 10:16:04 EDT.
Date: Thu, 19 Sep 96 09:58:00 PDT
From: Vern Paxson <vern>

 a lot about the backing up problem.
 I believe that there lies my biggest problem, and I'll try to improve
 it.

Since you have variable trailing context, this is a bigger performance
problem.  Fixing it is usually easier than fixing backing up, which in a
complicated scanner (yours seems to fit the bill) can be extremely
difficult to do correctly.
```

```
You also don't mention what flags you are using for your scanner.
-f makes a large speed difference, and -Cfe buys you nearly as much
speed but the resulting scanner is considerably smaller.

 I have an | operator in {and} and in {pats} so both of them are variable
 length.

-p should have reported this.

 Is changing one of them to fixed-length is enough ?

Yes.

 Is it possible to change the 32,000 states limit ?

Yes.  I've appended instructions on how.  Before you make this change,
though, you should think about whether there are ways to fundamentally
simplify your scanner - those are certainly preferable!

                Vern

To increase the 32K limit (on a machine with 32 bit integers), you increase
the magnitude of the following in flexdef.h:

#define JAMSTATE -32766 /* marks a reference to the state that always jams */
#define MAXIMUM_MNS 31999
#define BAD_SUBSCRIPT -32767
#define MAX_SHORT 32700

Adding a 0 or two after each should do the trick.
```

Can I fake multi-byte character support?

```
To: Heeman_Lee@hp.com
Subject: Re: flex - multi-byte support?
In-reply-to: Your message of Thu, 03 Oct 1996 17:24:04 PDT.
Date: Fri, 04 Oct 1996 11:42:18 PDT
From: Vern Paxson <vern>

      I assume as long as my *.l file defines the
      range of expected character code values (in octal format), flex will
      scan the file and read multi-byte characters correctly. But I have no
      confidence in this assumption.

Your lack of confidence is justified - this won't work.

Flex has in it a widespread assumption that the input is processed
one byte at a time.  Fixing this is on the to-do list, but is involved,
```

so it won't happen any time soon. In the interim, the best I can suggest (unless you want to try fixing it yourself) is to write your rules in terms of pairs of bytes, using definitions in the first section:

```
X       \xfe\xc2
...
%%
foo{X}bar       found_foo_fe_c2_bar();
```

etc. Definitely a pain - sorry about that.

By the way, the email address you used for me is ancient, indicating you have a very old version of flex. You can get the most recent, 2.5.4, from ftp.ee.lbl.gov.

Vern

deleteme01

To: moleary@primus.com
Subject: Re: Flex / Unicode compatibility question
In-reply-to: Your message of Tue, 22 Oct 1996 10:15:42 PDT.
Date: Tue, 22 Oct 1996 11:06:13 PDT
From: Vern Paxson <vern>

Unfortunately flex at the moment has a widespread assumption within it that characters are processed 8 bits at a time. I don't see any easy fix for this (other than writing your rules in terms of double characters - a pain). I also don't know of a wider lex, though you might try surfing the Plan 9 stuff because I know it's a Unicode system, and also the PCCT toolkit (try searching say Alta Vista for "Purdue Compiler Construction Toolkit").

Fixing flex to handle wider characters is on the long-term to-do list. But since flex is a strictly spare-time project these days, this probably won't happen for quite a while, unless someone else does it first.

Vern

Can you discuss some flex internals?

To: Johan Linde <jl@theophys.kth.se>
Subject: Re: translation of flex
In-reply-to: Your message of Sun, 10 Nov 1996 09:16:36 PST.
Date: Mon, 11 Nov 1996 10:33:50 PST
From: Vern Paxson <vern>

> I'm working for the Swedish team translating GNU program, and I'm currently working with flex. I have a few questions about some of the messages which I hope you can answer.

All of the things you're wondering about, by the way, concerning flex internals - probably the only person who understands what they mean in English is me! So I wouldn't worry too much about getting them right. That said ...

> #: main.c:545
> msgid " %d protos created\n"
>
> Does proto mean prototype?

Yes - prototypes of state compression tables.

> #: main.c:539
> msgid " %d/%d (peak %d) template nxt-chk entries created\n"
>
> Here I'm mainly puzzled by 'nxt-chk'. I guess it means 'next-check'. (?) However, 'template next-check entries' doesn't make much sense to me. To be able to find a good translation I need to know a little bit more about it.

There is a scheme in the Aho/Sethi/Ullman compiler book for compressing scanner tables. It involves creating two pairs of tables. The first has base" and "default" entries, the second has "next" and "check" entries. The "base" entry is indexed by the current state and yields an index into the next/check table. The "default" entry gives what to do if the state transition isn't found in next/check. The "next" entry gives the next state to enter, but only if the "check" entry verifies that this entry is correct for the current state. Flex creates templates of series of next/check entries and then encodes differences from these templates as a way to compress the tables.

> #: main.c:533
> msgid " %d/%d base-def entries created\n"
>
> The same problem here for 'base-def'.

See above.

Vern

unput() messes up yy_at_bol

To: Xinying Li <xli@npac.syr.edu>
Subject: Re: FLEX ?
In-reply-to: Your message of Wed, 13 Nov 1996 17:28:38 PST.

```
Date: Wed, 13 Nov 1996 19:51:54 PST
From: Vern Paxson <vern>

 "unput()" them to input flow, question occurs. If I do this after I scan
 a carriage, the variable "YY_CURRENT_BUFFER->yy_at_bol" is changed. That
 means the carriage flag has gone.

You can control this by calling yy_set_bol().  It's described in the manual.

      And if in pre-reading it goes to the end of file, is anything done
 to control the end of curren buffer and end of file?

No, there's no way to put back an end-of-file.

      By the way I am using flex 2.5.2 and using the "-l".

The latest release is 2.5.4, by the way.  It fixes some bugs in 2.5.2 and
2.5.3.  You can get it from ftp.ee.lbl.gov.

                Vern
```

The | operator is not doing what I want

```
To: Alain.ISSARD@st.com
Subject: Re: Start condition with FLEX
In-reply-to: Your message of Mon, 18 Nov 1996 09:45:02 PST.
Date: Mon, 18 Nov 1996 10:41:34 PST
From: Vern Paxson <vern>

 I am not able to use the start condition scope and to use the | (OR) with
 rules having start conditions.

The problem is that if you use '|' as a regular expression operator, for
example "a|b" meaning "match either 'a' or 'b'", then it must *not* have
any blanks around it.  If you instead want the special '|' *action* (which
from your scanner appears to be the case), which is a way of giving two
different rules the same action:

        foo     |
        bar     matched_foo_or_bar();

then '|' *must* be separated from the first rule by whitespace and *must*
be followed by a new line.  You *cannot* write it as:

        foo | bar       matched_foo_or_bar();

even though you might think you could because yacc supports this syntax.
The reason for this unfortunately incompatibility is historical, but it's
```

unlikely to be changed.

Your problems with start condition scope are simply due to syntax errors
from your use of '|' later confusing flex.

Let me know if you still have problems.

Vern

Why can't flex understand this variable trailing context pattern?

To: Gregory Margo <gmargo@newton.vip.best.com>
Subject: Re: flex-2.5.3 bug report
In-reply-to: Your message of Sat, 23 Nov 1996 16:50:09 PST.
Date: Sat, 23 Nov 1996 17:07:32 PST
From: Vern Paxson <vern>

Enclosed is a lex file that "real" lex will process, but I cannot get
flex to process it. Could you try it and maybe point me in the right direction?

Your problem is that some of the definitions in the scanner use the '/'
trailing context operator, and have it enclosed in ()'s. Flex does not
allow this operator to be enclosed in ()'s because doing so allows undefined
regular expressions such as "(a/b)+". So the solution is to remove the
parentheses. Note that you must also be building the scanner with the -l
option for AT&T lex compatibility. Without this option, flex automatically
encloses the definitions in parentheses.

Vern

The ^ operator isn't working

To: Thomas Hadig <hadig@toots.physik.rwth-aachen.de>
Subject: Re: Flex Bug ?
In-reply-to: Your message of Tue, 26 Nov 1996 14:35:01 PST.
Date: Tue, 26 Nov 1996 11:15:05 PST
From: Vern Paxson <vern>

In my lexer code, i have the line :
^*.* { }

Thus all lines starting with an astrix (*) are comment lines.
This does not work !

I can't get this problem to reproduce - it works fine for me. Note
though that if what you have is slightly different:

```
COMMENT ^\*.*
%%
{COMMENT}       { }
```

then it won't work, because flex pushes back macro definitions enclosed in ()'s, so the rule becomes

```
(^\*.*)         { }
```

and now that the '^' operator is not at the immediate beginning of the line, it's interpreted as just a regular character. You can avoid this behavior by using the "-l" lex-compatibility flag, or "%option lex-compat".

Vern

Trailing context is getting confused with trailing optional patterns

To: Adoram Rogel <adoram@hybridge.com>
Subject: Re: Flex 2.5.4 BOF ???
In-reply-to: Your message of Tue, 26 Nov 1996 16:10:41 PST.
Date: Wed, 27 Nov 1996 10:56:25 PST
From: Vern Paxson <vern>

```
Organization(s)?/[a-z]

This matched "Organizations" (looking in debug mode, the trailing s
was matched with trailing context instead of the optional (s) in the
end of the word.
```

That should only happen with lex. Flex can properly match this pattern. (That might be what you're saying, I'm just not sure.)

Is there a way to avoid this dangerous trailing context problem ?

Unfortunately, there's no easy way. On the other hand, I don't see why it should be a problem. Lex's matching is clearly wrong, and I'd hope that usually the intent remains the same as expressed with the pattern, so flex's matching will be correct.

Vern

Is flex GNU or not?

To: Cameron MacKinnon <mackin@interlog.com>
Subject: Re: Flex documentation bug

```
In-reply-to: Your message of Mon, 02 Dec 1996 00:07:08 PST.
Date: Sun, 01 Dec 1996 22:29:39 PST
From: Vern Paxson <vern>

 I'm not sure how or where to submit bug reports (documentation or
 otherwise) for the GNU project stuff ...

Well, strictly speaking flex isn't part of the GNU project.  They just
distribute it because no one's written a decent GPL'd lex replacement.
So you should send bugs directly to me.  Those sent to the GNU folks
sometimes find there way to me, but some may drop between the cracks.

 In GNU Info, under the section 'Start Conditions', and also in the man
 page (mine's dated April '95) is a nice little snippet showing how to
 parse C quoted strings into a buffer, defined to be MAX_STR_CONST in
 size. Unfortunately, no overflow checking is ever done ...

This is already mentioned in the manual:

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):

The reason for not doing the overflow checking is that it will needlessly
clutter up an example whose main purpose is just to demonstrate how to
use flex.

The latest release is 2.5.4, by the way, available from ftp.ee.lbl.gov.

                Vern
```

ERASEME53

```
To: tsv@cs.UManitoba.CA
Subject: Re: Flex (reg)..
In-reply-to: Your message of Thu, 06 Mar 1997 23:50:16 PST.
Date: Thu, 06 Mar 1997 15:54:19 PST
From: Vern Paxson <vern>

 [:alpha:] ([:alnum:] | \\_)*

If your rule really has embedded blanks as shown above, then it won't
work, as the first blank delimits the rule from the action.  (It wouldn't
even compile ...)  You need instead:

[:alpha:]([:alnum:]|\\_)*
```

and that should work fine - there's no restriction on what can go inside of ()'s except for the trailing context operator, '/'.

Vern

I need to scan if-then-else blocks and while loops

To: "Mike Stolnicki" <mstolnic@ford.com>
Subject: Re: FLEX help
In-reply-to: Your message of Fri, 30 May 1997 13:33:27 PDT.
Date: Fri, 30 May 1997 10:46:35 PDT
From: Vern Paxson <vern>

> We'd like to add "if-then-else", "while", and "for" statements to our language ...
> We've investigated many possible solutions. The one solution that seems the most reasonable involves knowing the position of a TOKEN in yyin.

I strongly advise you to instead build a parse tree (abstract syntax tree) and loop over that instead. You'll find this has major benefits in keeping your interpreter simple and extensible.

That said, the functionality you mention for get_position and set_position have been on the to-do list for a while. As flex is a purely spare-time project for me, no guarantees when this will be added (in particular, it for sure won't be for many months to come).

Vern

ERASEME55

To: Colin Paul Adams <colin@colina.demon.co.uk>
Subject: Re: Flex C++ classes and Bison
In-reply-to: Your message of 09 Aug 1997 17:11:41 PDT.
Date: Fri, 15 Aug 1997 10:48:19 PDT
From: Vern Paxson <vern>

> #define YY_DECL int yylex (YYSTYPE *lvalp, struct parser_control *parm)
>
> I have been trying to get this to work as a C++ scanner, but it does not appear to be possible (warning that it matches no declarations in yyFlexLexer, or something like that).
>
> Is this supposed to be possible, or is it being worked on (I DID notice the comment that scanner classes are still experimental, so I'm not too hopeful)?

What you need to do is derive a subclass from yyFlexLexer that provides
the above yylex() method, squirrels away lvalp and parm into member
variables, and then invokes yyFlexLexer::yylex() to do the regular scanning.

Vern

ERASEME56

To: Mikael.Latvala@lmf.ericsson.se
Subject: Re: Possible mistake in Flex v2.5 document
In-reply-to: Your message of Fri, 05 Sep 1997 16:07:24 PDT.
Date: Fri, 05 Sep 1997 10:01:54 PDT
From: Vern Paxson <vern>

> In that example you show how to count comment lines when using
> C style /* ... */ comments. My question is, shouldn't you take into
> account a scenario where end of a comment marker occurs inside
> character or string literals?

The scanner certainly needs to also scan character and string literals.
However it does that (there's an example in the man page for strings), the
lexer will recognize the beginning of the literal before it runs across the
embedded "/*". Consequently, it will finish scanning the literal before it
even considers the possibility of matching "/*".

Example:

```
'([^']*|{ESCAPE_SEQUENCE})'
```

will match all the text between the ''s (inclusive). So the lexer
considers this as a token beginning at the first ', and doesn't even
attempt to match other tokens inside it.

I thinnk this subtlety is not worth putting in the manual, as I suspect
it would confuse more people than it would enlighten.

Vern

ERASEME57

To: "Marty Leisner" <leisner@sdsp.mc.xerox.com>
Subject: Re: flex limitations
In-reply-to: Your message of Sat, 06 Sep 1997 11:27:21 PDT.
Date: Mon, 08 Sep 1997 11:38:08 PDT
From: Vern Paxson <vern>

```
%%
[a-zA-Z]+       /* skip a line */
                {  printf("got %s\n", yytext); }
%%
```

```
What version of flex are you using?  If I feed this to 2.5.4, it complains:

        "bug.l", line 5: EOF encountered inside an action
        "bug.l", line 5: unrecognized rule
        "bug.l", line 5: fatal parse error

Not the world's greatest error message, but it manages to flag the problem.

(With the introduction of start condition scopes, flex can't accommodate
an action on a separate line, since it's ambiguous with an indented rule.)

You can get 2.5.4 from ftp.ee.lbl.gov.

                Vern
```

Is there a repository for flex scanners?

Not that we know of. You might try asking on comp.compilers.

How can I conditionally compile or preprocess my flex input file?

Flex doesn't have a preprocessor like C does. You might try using m4, or the C preprocessor plus a sed script to clean up the result.

Where can I find grammars for lex and yacc?

In the sources for flex and bison.

I get an end-of-buffer message for each character scanned.

This will happen if your LexerInput() function returns only one character at a time, which can happen either if you're scanner is "interactive", or if the streams library on your platform always returns 1 for yyin->gcount().

Solution: override LexerInput() with a version that returns whole buffers.

unnamed-faq-62

```
To: Georg.Rehm@CL-KI.Uni-Osnabrueck.DE
Subject: Re: Flex maximums
In-reply-to: Your message of Mon, 17 Nov 1997 17:16:06 PST.
Date: Mon, 17 Nov 1997 17:16:15 PST
```

```
From: Vern Paxson <vern>

 I took a quick look into the flex-sources and altered some #defines in
 flexdefs.h:

 #define INITIAL_MNS 64000
 #define MNS_INCREMENT 1024000
 #define MAXIMUM_MNS 64000

The things to fix are to add a couple of zeroes to:

#define JAMSTATE -32766 /* marks a reference to the state that always jams */
#define MAXIMUM_MNS 31999
#define BAD_SUBSCRIPT -32767
#define MAX_SHORT 32700

and, if you get complaints about too many rules, make the following change too:

        #define YY_TRAILING_MASK 0x200000
        #define YY_TRAILING_HEAD_MASK 0x400000

- Vern
```

unnamed-faq-63

```
To: jimmey@lexis-nexis.com (Jimmey Todd)
Subject: Re: FLEX question regarding istream vs ifstream
In-reply-to: Your message of Mon, 08 Dec 1997 15:54:15 PST.
Date: Mon, 15 Dec 1997 13:21:35 PST
From: Vern Paxson <vern>

         stdin_handle = YY_CURRENT_BUFFER;
         ifstream fin( "aFile" );
         yy_switch_to_buffer( yy_create_buffer( fin, YY_BUF_SIZE ) );

 What I'm wanting to do, is pass the contents of a file thru one set
 of rules and then pass stdin thru another set... It works great if, I
 don't use the C++ classes. But since everything else that I'm doing is
 in C++, I thought I'd be consistent.

 The problem is that 'yy_create_buffer' is expecting an istream* as it's
 first argument (as stated in the man page). However, fin is a ifstream
 object. Any ideas on what I might be doing wrong? Any help would be
 appreciated. Thanks!!

You need to pass &fin, to turn it into an ifstream* instead of an ifstream.
Then its type will be compatible with the expected istream*, because ifstream
is derived from istream.
```

```
                Vern
```

unnamed-faq-64

```
To: Enda Fadian <fadiane@piercom.ie>
Subject: Re: Question related to Flex man page?
In-reply-to: Your message of Tue, 16 Dec 1997 15:17:34 PST.
Date: Tue, 16 Dec 1997 14:17:09 PST
From: Vern Paxson <vern>

 Can you explain to me what is ment by a long-jump in relation to flex?

Using the longjmp() function while inside yylex() or a routine called by it.

 what is the flex activation frame.

Just yylex()'s stack frame.

 As far as I can see yyrestart will bring me back to the sart of the input
 file and using flex++ isnot really an option!

No, yyrestart() doesn't imply a rewind, even though its name might sound
like it does.  It tells the scanner to flush its internal buffers and
start reading from the given file at its present location.

                Vern
```

unnamed-faq-65

```
To: hassan@larc.info.uqam.ca (Hassan Alaoui)
Subject: Re: Need urgent Help
In-reply-to: Your message of Sat, 20 Dec 1997 19:38:19 PST.
Date: Sun, 21 Dec 1997 21:30:46 PST
From: Vern Paxson <vern>

 /usr/lib/yaccpar: In function `int yyparse()':
 /usr/lib/yaccpar:184: warning: implicit declaration of function `int yylex(...)'

 ld: Undefined symbol
    _yylex
    _yyparse
    _yyin

This is a known problem with Solaris C++ (and/or Solaris yacc).  I believe
the fix is to explicitly insert some 'extern "C"' statements for the
corresponding routines/symbols.
```

Vern

unnamed-faq-66

To: mc0307@mclink.it
Cc: gnu@prep.ai.mit.edu
Subject: Re: [mc0307@mclink.it: Help request]
In-reply-to: Your message of Fri, 12 Dec 1997 17:57:29 PST.
Date: Sun, 21 Dec 1997 22:33:37 PST
From: Vern Paxson <vern>

> This is my definition for float and integer types:
> . . .
> NZD [1-9]
> ...
> I've tested my program on other lex version (on UNIX Sun Solaris an HP
> UNIX) and it work well, so I think that my definitions are correct.
> There are any differences between Lex and Flex?

There are indeed differences, as discussed in the man page. The one
you are probably running into is that when flex expands a name definition,
it puts parentheses around the expansion, while lex does not. There's
an example in the man page of how this can lead to different matching.
Flex's behavior complies with the POSIX standard (or at least with the
last POSIX draft I saw).

Vern

unnamed-faq-67

To: hassan@larc.info.uqam.ca (Hassan Alaoui)
Subject: Re: Thanks
In-reply-to: Your message of Mon, 22 Dec 1997 16:06:35 PST.
Date: Mon, 22 Dec 1997 14:35:05 PST
From: Vern Paxson <vern>

> Thank you very much for your help. I compile and link well with C++ while
> declaring 'yylex ...' extern, But a little problem remains. I get a
> segmentation default when executing (I linked with lfl library) while it
> works well when using LEX instead of flex. Do you have some ideas about the
> reason for this ?

The one possible reason for this that comes to mind is if you've defined
yytext as "extern char yytext[]" (which is what lex uses) instead of
extern char *yytext" (which is what flex uses). If it's not that, then
I'm afraid I don't know what the problem might be.

Vern

unnamed-faq-68

To: "Bart Niswonger" <NISWONGR@almaden.ibm.com>
Subject: Re: flex 2.5: c++ scanners & start conditions
In-reply-to: Your message of Tue, 06 Jan 1998 10:34:21 PST.
Date: Tue, 06 Jan 1998 19:19:30 PST
From: Vern Paxson <vern>

> The problem is that when I do this (using %option c++) start
> conditions seem to not apply.

The BEGIN macro modifies the yy_start variable. For C scanners, this is a static with scope visible through the whole file. For C++ scanners, it's a member variable, so it only has visible scope within a member function. Your lexbegin() routine is not a member function when you build a C++ scanner, so it's not modifying the correct yy_start. The diagnostic that indicates this is that you found you needed to add a declaration of yy_start in order to get your scanner to compile when using C++; instead, the correct fix is to make lexbegin() a member function (by deriving from yyFlexLexer).

Vern

unnamed-faq-69

To: "Boris Zinin" <boris@ippe.rssi.ru>
Subject: Re: current position in flex buffer
In-reply-to: Your message of Mon, 12 Jan 1998 18:58:23 PST.
Date: Mon, 12 Jan 1998 12:03:15 PST
From: Vern Paxson <vern>

> The problem is how to determine the current position in flex active
> buffer when a rule is matched....

You will need to keep track of this explicitly, such as by redefining YY_USER_ACTION to count the number of characters matched.

The latest flex release, by the way, is 2.5.4, available from ftp.ee.lbl.gov.

Vern

unnamed-faq-70

```
To: Bik.Dhaliwal@bis.org
Subject: Re: Flex question
In-reply-to: Your message of Mon, 26 Jan 1998 13:05:35 PST.
Date: Tue, 27 Jan 1998 22:41:52 PST
From: Vern Paxson <vern>

 That requirement involves knowing
 the character position at which a particular token was matched
 in the lexer.

The way you have to do this is by explicitly keeping track of where
you are in the file, by counting the number of characters scanned
for each token (available in yyleng).  It may prove convenient to
do this by redefining YY_USER_ACTION, as described in the manual.

                Vern
```

unnamed-faq-71

```
To: Vladimir Alexiev <vladimir@cs.ualberta.ca>
Subject: Re: flex: how to control start condition from parser?
In-reply-to: Your message of Mon, 26 Jan 1998 05:50:16 PST.
Date: Tue, 27 Jan 1998 22:45:37 PST
From: Vern Paxson <vern>

 It seems useful for the parser to be able to tell the lexer about such
 context dependencies, because then they don't have to be limited to
 local or sequential context.

One way to do this is to have the parser call a stub routine that's
included in the scanner's .l file, and consequently that has access ot
BEGIN.  The only ugliness is that the parser can't pass in the state
it wants, because those aren't visible - but if you don't have many
such states, then using a different set of names doesn't seem like
to much of a burden.

While generating a .h file like you suggests is certainly cleaner,
flex development has come to a virtual stand-still :-(, so a workaround
like the above is much more pragmatic than waiting for a new feature.

                Vern
```

unnamed-faq-72

```
To: Barbara Denny <denny@3com.com>
Subject: Re: freebsd flex bug?
In-reply-to: Your message of Fri, 30 Jan 1998 12:00:43 PST.
```

```
Date: Fri, 30 Jan 1998 12:42:32 PST
From: Vern Paxson <vern>

 lex.yy.c:1996: parse error before `='

This is the key, identifying this error.  (It may help to pinpoint
it by using flex -L, so it doesn't generate #line directives in its
output.)  I will bet you heavy money that you have a start condition
name that is also a variable name, or something like that; flex spits
out #define's for each start condition name, mapping them to a number,
so you can wind up with:

        %x foo
        %%
                ...
        %%
        void bar()
                {
                int foo = 3;
                }

and the penultimate will turn into "int 1 = 3" after C preprocessing,
since flex will put "#define foo 1" in the generated scanner.

                Vern
```

unnamed-faq-73

```
To: Maurice Petrie <mpetrie@infoscigroup.com>
Subject: Re: Lost flex .l file
In-reply-to: Your message of Mon, 02 Feb 1998 14:10:01 PST.
Date: Mon, 02 Feb 1998 11:15:12 PST
From: Vern Paxson <vern>

 I am curious as to
 whether there is a simple way to backtrack from the generated source to
 reproduce the lost list of tokens we are searching on.

In theory, it's straight-forward to go from the DFA representation
back to a regular-expression representation - the two are isomorphic.
In practice, a huge headache, because you have to unpack all the tables
back into a single DFA representation, and then write a program to munch
on that and translate it into an RE.

Sorry for the less-than-happy news ...

                Vern
```

unnamed-faq-74

```
To: jimmey@lexis-nexis.com (Jimmey Todd)
Subject: Re: Flex performance question
In-reply-to: Your message of Thu, 19 Feb 1998 11:01:17 PST.
Date: Thu, 19 Feb 1998 08:48:51 PST
From: Vern Paxson <vern>

 What I have found, is that the smaller the data chunk, the faster the
 program executes. This is the opposite of what I expected. Should this be
 happening this way?

This is exactly what will happen if your input file has embedded NULs.
From the man page:

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.

So that's the first thing to look for.

                Vern
```

unnamed-faq-75

```
To: jimmey@lexis-nexis.com (Jimmey Todd)
Subject: Re: Flex performance question
In-reply-to: Your message of Thu, 19 Feb 1998 11:01:17 PST.
Date: Thu, 19 Feb 1998 15:42:25 PST
From: Vern Paxson <vern>

So there are several problems.

First, to go fast, you want to match as much text as possible, which
your scanners don't in the case that what they're scanning is *not*
a <RN> tag.  So you want a rule like:

        [^<]+

Second, C++ scanners are particularly slow if they're interactive,
which they are by default.  Using -B speeds it up by a factor of 3-4
on my workstation.

Third, C++ scanners that use the istream interface are slow, because
of how poorly implemented istream's are.  I built two versions of
the following scanner:
```

```
%%
.*\n
.*
%%
```

and the C version inhales a 2.5MB file on my workstation in 0.8 seconds.
The C++ istream version, using -B, takes 3.8 seconds.

Vern

unnamed-faq-76

To: "Frescatore, David (CRD, TAD)" <frescatore@exc01crdge.crd.ge.com>
Subject: Re: FLEX 2.5 & THE YEAR 2000
In-reply-to: Your message of Wed, 03 Jun 1998 11:26:22 PDT.
Date: Wed, 03 Jun 1998 10:22:26 PDT
From: Vern Paxson <vern>

> I am researching the Y2K problem with General Electric R&D
> and need to know if there are any known issues concerning
> the above mentioned software and Y2K regardless of version.

There shouldn't be, all it ever does with the date is ask the system
for it and then print it out.

Vern

unnamed-faq-77

To: "Hans Dermot Doran" <htd@ibhdoran.com>
Subject: Re: flex problem
In-reply-to: Your message of Wed, 15 Jul 1998 21:30:13 PDT.
Date: Tue, 21 Jul 1998 14:23:34 PDT
From: Vern Paxson <vern>

> To overcome this, I gets() the stdin into a string and lex the string. The
> string is lexed OK except that the end of string isn't lexed properly
> (yy_scan_string()), that is the lexer dosn't recognise the end of string.

Flex doesn't contain mechanisms for recognizing buffer endpoints. But if
you use fgets instead (which you should anyway, to protect against buffer
overflows), then the final \n will be preserved in the string, and you can
scan that in order to find the end of the string.

Vern

unnamed-faq-78

```
To: soumen@almaden.ibm.com
Subject: Re: Flex++ 2.5.3 instance member vs. static member
In-reply-to: Your message of Mon, 27 Jul 1998 02:10:04 PDT.
Date: Tue, 28 Jul 1998 01:10:34 PDT
From: Vern Paxson <vern>

 %{
 int mylineno = 0;
 %}
 ws      [ \t]+
 alpha   [A-Za-z]
 dig     [0-9]
 %%

 Now you'd expect mylineno to be a member of each instance of class
 yyFlexLexer, but is this the case?  A look at the lex.yy.cc file seems to
 indicate otherwise; unless I am missing something the declaration of
 mylineno seems to be outside any class scope.

 How will this work if I want to run a multi-threaded application with each
 thread creating a FlexLexer instance?

Derive your own subclass and make mylineno a member variable of it.

                Vern
```

unnamed-faq-79

```
To: Adoram Rogel <adoram@hybridge.com>
Subject: Re: More than 32K states change hangs
In-reply-to: Your message of Tue, 04 Aug 1998 16:55:39 PDT.
Date: Tue, 04 Aug 1998 22:28:45 PDT
From: Vern Paxson <vern>

 Vern Paxson,

 I followed your advice, posted on Usenet bu you, and emailed to me
 personally by you, on how to overcome the 32K states limit. I'm running
 on Linux machines.
 I took the full source of version 2.5.4 and did the following changes in
 flexdef.h:
 #define JAMSTATE -327660
 #define MAXIMUM_MNS 319990
 #define BAD_SUBSCRIPT -327670
 #define MAX_SHORT 327000
```

```
 and compiled.
 All looked fine, including check and bigcheck, so I installed.

Hmmm, you shouldn't increase MAX_SHORT, though looking through my email
archives I see that I did indeed recommend doing so.  Try setting it back
to 32700; that should suffice that you no longer need -Ca.  If it still
hangs, then the interesting question is - where?

 Compiling the same hanged program with a out-of-the-box (RedHat 4.2
 distribution of Linux)
 flex 2.5.4 binary works.

Since Linux comes with source code, you should diff it against what
you have to see what problems they missed.

 Should I always compile with the -Ca option now ? even short and simple
 filters ?

No, definitely not.  It's meant to be for those situations where you
absolutely must squeeze every last cycle out of your scanner.

                Vern
```

unnamed-faq-80

```
To: "Schmackpfeffer, Craig" <Craig.Schmackpfeffer@usa.xerox.com>
Subject: Re: flex output for static code portion
In-reply-to: Your message of Tue, 11 Aug 1998 11:55:30 PDT.
Date: Mon, 17 Aug 1998 23:57:42 PDT
From: Vern Paxson <vern>

 I would like to use flex under the hood to generate a binary file
 containing the data structures that control the parse.

This has been on the wish-list for a long time.  In principle it's
straight-forward - you redirect mkdata() et al's I/O to another file,
and modify the skeleton to have a start-up function that slurps these
into dynamic arrays.  The concerns are (1) the scanner generation code
is hairy and full of corner cases, so it's easy to get surprised when
going down this path :-( ; and (2) being careful about buffering so
that when the tables change you make sure the scanner starts in the
correct state and reading at the right point in the input file.

 I was wondering if you know of anyone who has used flex in this way.

I don't - but it seems like a reasonable project to undertake (unlike
numerous other flex tweaks :-).
```

Vern

unnamed-faq-81

```
Received: from 131.173.17.11 (131.173.17.11 [131.173.17.11])
        by ee.lbl.gov (8.9.1/8.9.1) with ESMTP id AAA03838
        for <vern@ee.lbl.gov>; Thu, 20 Aug 1998 00:47:57 -0700 (PDT)
Received: from hal.cl-ki.uni-osnabrueck.de (hal.cl-ki.Uni-Osnabrueck.DE [131.173.141.2
        by deimos.rz.uni-osnabrueck.de (8.8.7/8.8.8) with ESMTP id JAA34694
        for <vern@ee.lbl.gov>; Thu, 20 Aug 1998 09:47:55 +0200
Received: (from georg@localhost) by hal.cl-ki.uni-osnabrueck.de (8.6.12/8.6.12) id JAA
From: Georg Rehm <georg@hal.cl-ki.uni-osnabrueck.de>
Message-Id: <199808200747.JAA34834@hal.cl-ki.uni-osnabrueck.de>
Subject: "flex scanner push-back overflow"
To: vern@ee.lbl.gov
Date: Thu, 20 Aug 1998 09:47:54 +0200 (MEST)
Reply-To: Georg.Rehm@CL-KI.Uni-Osnabrueck.DE
X-NoJunk: Do NOT send commercial mail, spam or ads to this address!
X-URL: http://www.cl-ki.uni-osnabrueck.de/~georg/
X-Mailer: ELM [version 2.4ME+ PL28 (25)]
MIME-Version: 1.0
Content-Type: text/plain; charset=US-ASCII
Content-Transfer-Encoding: 7bit

Hi Vern,

Yesterday, I encountered a strange problem: I use the macro processor m4
to include some lengthy lists into a .l file. Following is a flex macro
definition that causes some serious pain in my neck:

AUTHOR           ("A. Boucard / L. Boucard"|"A. Dastarac / M. Levent"|"A.Boucaud / L.B

The complete list contains about 10kB. When I try to "flex" this file
(on a Solaris 2.6 machine, using a modified flex 2.5.4 (I only increased
some of the predefined values in flexdefs.h) I get the error:

myflex/flex -8  sentag.tmp.l
flex scanner push-back overflow

When I remove the slashes in the macro definition everything works fine.
As I understand it, the double quotes escape the slash-character so it
really means "/" and not "trailing context". Furthermore, I tried to
escape the slashes with backslashes, but with no use, the same error message
appeared when flexing the code.

Do you have an idea what's going on here?

Greetings from Germany,
```

```
        Georg
--
Georg Rehm                                          georg@cl-ki.uni-osnabrueck.de
Institute for Semantic Information Processing, University of Osnabrueck, FRG
```

unnamed-faq-82

```
To: Georg.Rehm@CL-KI.Uni-Osnabrueck.DE
Subject: Re: "flex scanner push-back overflow"
In-reply-to: Your message of Thu, 20 Aug 1998 09:47:54 PDT.
Date: Thu, 20 Aug 1998 07:05:35 PDT
From: Vern Paxson <vern>

 myflex/flex -8  sentag.tmp.l
 flex scanner push-back overflow

Flex itself uses a flex scanner.  That scanner is running out of buffer
space when it tries to unput() the humongous macro you've defined.  When
you remove the '/'s, you make it small enough so that it fits in the buffer;
removing spaces would do the same thing.

The fix is to either rethink how come you're using such a big macro and
perhaps there's another/better way to do it; or to rebuild flex's own
scan.c with a larger value for

        #define YY_BUF_SIZE 16384

- Vern
```

unnamed-faq-83

```
To: Jan Kort <jan@research.techforce.nl>
Subject: Re: Flex
In-reply-to: Your message of Fri, 04 Sep 1998 12:18:43 +0200.
Date: Sat, 05 Sep 1998 00:59:49 PDT
From: Vern Paxson <vern>

 %%

 "TEST1\n"       { fprintf(stderr, "TEST1\n"); yyless(5); }
 ^\n             { fprintf(stderr, "empty line\n"); }
 .               { }
 \n              { fprintf(stderr, "new line\n"); }

 %%
 -- input ---------------------------------------
 TEST1
```

```
-- output --------------------------------------
TEST1
empty line
------------------------------------------------
```

IMHO, it's not clear whether or not this is in fact a bug. It depends on whether you view yyless() as backing up in the input stream, or as pushing new characters onto the beginning of the input stream. Flex interprets it as the latter (for implementation convenience, I'll admit), and so considers the newline as in fact matching at the beginning of a line, as after all the last token scanned an entire line and so the scanner is now at the beginning of a new line.

I agree that this is counter-intuitive for yyless(), given its functional description (it's less so for unput(), depending on whether you're unput()'ing new text or scanned text). But I don't plan to change it any time soon, as it's a pain to do so. Consequently, you do indeed need to use yy_set_bol() and YY_AT_BOL() to tweak your scanner into the behavior you desire.

Sorry for the less-than-completely-satisfactory answer.

Vern

unnamed-faq-84

To: Patrick Krusenotto <krusenot@mac-info-link.de>
Subject: Re: Problems with restarting flex-2.5.2-generated scanner
In-reply-to: Your message of Thu, 24 Sep 1998 10:14:07 PDT.
Date: Thu, 24 Sep 1998 23:28:43 PDT
From: Vern Paxson <vern>

> I am using flex-2.5.2 and bison 1.25 for Solaris and I am desperately trying to make my scanner restart with a new file after my parser stops with a parse error. When my compiler restarts, the parser always receives the token after the token (in the old file!) that caused the parser error.

I suspect the problem is that your parser has read ahead in order to attempt to resolve an ambiguity, and when it's restarted it picks up with that token rather than reading a fresh one. If you're using yacc, then the special "error" production can sometimes be used to consume tokens in an attempt to get the parser into a consistent state.

Vern

unnamed-faq-85

```
To: Henric Jungheim <junghelh@pe-nelson.com>
Subject: Re: flex 2.5.4a
In-reply-to: Your message of Tue, 27 Oct 1998 16:41:42 PST.
Date: Tue, 27 Oct 1998 16:50:14 PST
From: Vern Paxson <vern>

 This brings up a feature request:  How about a command line
 option to specify the filename when reading from stdin?  That way one
 doesn't need to create a temporary file in order to get the "#line"
 directives to make sense.

Use -o combined with -t (per the man page description of -o).

 P.S., Is there any simple way to use non-blocking IO to parse multiple
 streams?

Simple, no.

One approach might be to return a magic character on EWOULDBLOCK and
have a rule

        .*<magic-character>     // put back .*, eat magic character

This is off the top of my head, not sure it'll work.

                Vern
```

unnamed-faq-86

```
To: "Repko, Billy D" <billy.d.repko@intel.com>
Subject: Re: Compiling scanners
In-reply-to: Your message of Wed, 13 Jan 1999 10:52:47 PST.
Date: Thu, 14 Jan 1999 00:25:30 PST
From: Vern Paxson <vern>

 It appears that maybe it cannot find the lfl library.

The Makefile in the distribution builds it, so you should have it.
It's exceedingly trivial, just a main() that calls yylex() and
a yyrap() that always returns 1.

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

You can't indent your rules like this - that's where the errors are coming
```

```
from.  Flex copies indented text to the output file, it's how you do things
like

        int num_lines_seen = 0;

to declare local variables.

                Vern
```

unnamed-faq-87

```
To: Erick Branderhorst <Erick.Branderhorst@asml.nl>
Subject: Re: flex input buffer
In-reply-to: Your message of Tue, 09 Feb 1999 13:53:46 PST.
Date: Tue, 09 Feb 1999 21:03:37 PST
From: Vern Paxson <vern>

 In the flex.skl file the size of the default input buffers is set.  Can you
 explain why this size is set and why it is such a high number.

It's large to optimize performance when scanning large files.  You can
safely make it a lot lower if needed.

                Vern
```

unnamed-faq-88

```
To: "Guido Minnen" <guidomi@cogs.susx.ac.uk>
Subject: Re: Flex error message
In-reply-to: Your message of Wed, 24 Feb 1999 15:31:46 PST.
Date: Thu, 25 Feb 1999 00:11:31 PST
From: Vern Paxson <vern>

 I'm extending a larger scanner written in Flex and I keep running into
 problems. More specifically, I get the error message:
 "flex: input rules are too complicated (>= 32000 NFA states)"

Increase the definitions in flexdef.h for:

#define JAMSTATE -32766 /* marks a reference to the state that always j
ams */
#define MAXIMUM_MNS 31999
#define BAD_SUBSCRIPT -32767

recompile everything, and it should all work.

                Vern
```

unnamed-faq-90

```
To: "Dmitriy Goldobin" <gold@ems.chel.su>
Subject: Re: FLEX trouble
In-reply-to: Your message of Mon, 31 May 1999 18:44:49 PDT.
Date: Tue, 01 Jun 1999 00:15:07 PDT
From: Vern Paxson <vern>

   I have a trouble with FLEX. Why rule "/*".*"*/" work properly,=20
 but rule "/*"(.|\n)*"*/" don't work ?

The second of these will have to scan the entire input stream (because
(.|\n)*" matches an arbitrary amount of any text) in order to see if
it ends with "*/", terminating the comment.  That potentially will overflow
the input buffer.

   More complex rule "/*"([^*]|(\*/[^/]))*"*/ give an error
 'unrecognized rule'.

You can't use the '/' operator inside parentheses.  It's not clear
what "(a/b)*" actually means.

   I now use workaround with state <comment>, but single-rule is
 better, i think.

Single-rule is nice but will always have the problem of either setting
restrictions on comments (like not allowing multi-line comments) and/or
running the risk of consuming the entire input stream, as noted above.

                Vern
```

unnamed-faq-91

```
Received: from mc-qout4.whowhere.com (mc-qout4.whowhere.com [209.185.123.18])
        by ee.lbl.gov (8.9.3/8.9.3) with SMTP id IAA05100
        for <vern@ee.lbl.gov>; Tue, 15 Jun 1999 08:56:06 -0700 (PDT)
Received: from Unknown/Local ([?.?.?.?]) by my-deja.com; Tue Jun 15 08:55:43 1999
To: vern@ee.lbl.gov
Date: Tue, 15 Jun 1999 08:55:43 -0700
From: "Aki Niimura" <neko@my-deja.com>
Message-ID: <KNONDOHDOBGAEAAA@my-deja.com>
Mime-Version: 1.0
Cc:
X-Sent-Mail: on
Reply-To:
X-Mailer: MailCity Service
Subject: A question on flex C++ scanner
X-Sender-Ip: 12.72.207.61
```

```
Organization: My Deja Email  (http://www.my-deja.com:80)
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit

Dear Dr. Paxon,

I have been using flex for years.
It works very well on many projects.
Most case, I used it to generate a scanner on C language.
However, one project I needed to generate  a scanner
on C++ lanuage. Thanks to your enhancement, flex did
the job.

Currently, I'm working on enhancing my previous project.
I need to deal with multiple input streams (recursive
inclusion) in this scanner (C++).
I did similar thing for another scanner (C) as you
explained in your documentation.

The generated scanner (C++) has necessary methods:
- switch_to_buffer(struct yy_buffer_state *b)
- yy_create_buffer(istream *is, int sz)
- yy_delete_buffer(struct yy_buffer_state *b)

However, I couldn't figure out how to access current
buffer (yy_current_buffer).

yy_current_buffer is a protected member of yyFlexLexer.
I can't access it directly.
Then, I thought yy_create_buffer() with is = 0 might
return current stream buffer. But it seems not as far
as I checked the source. (flex 2.5.4)

I went through the Web in addition to Flex documentation.
However, it hasn't been successful, so far.

It is not my intention to bother you, but, can you
comment about how to obtain the current stream buffer?

Your response would be highly appreciated.

Best regards,
Aki Niimura

--== Sent via Deja.com http://www.deja.com/ ==--
Share what you know. Learn what you don't.
```

unnamed-faq-92

```
To: neko@my-deja.com
Subject: Re: A question on flex C++ scanner
In-reply-to: Your message of Tue, 15 Jun 1999 08:55:43 PDT.
Date: Tue, 15 Jun 1999 09:04:24 PDT
From: Vern Paxson <vern>

 However, I couldn't figure out how to access current
 buffer (yy_current_buffer).

Derive your own subclass from yyFlexLexer.

                Vern
```

unnamed-faq-93

```
To: "Stones, Darren" <Darren.Stones@nectech.co.uk>
Subject: Re: You're the man to see?
In-reply-to: Your message of Wed, 23 Jun 1999 11:10:29 PDT.
Date: Wed, 23 Jun 1999 09:01:40 PDT
From: Vern Paxson <vern>

 I hope you can help me.  I am using Flex and Bison to produce an interpreted
 language.  However all goes well until I try to implement an IF statement or
 a WHILE.  I cannot get this to work as the parser parses all the conditions
 eg. the TRUE and FALSE conditons to check for a rule match.  So I cannot
 make a decision!!

You need to use the parser to build a parse tree (= abstract syntax trwee),
and when that's all done you recursively evaluate the tree, binding variables
to values at that time.

                Vern
```

unnamed-faq-94

```
To: Petr Danecek <petr@ics.cas.cz>
Subject: Re: flex - question
In-reply-to: Your message of Mon, 28 Jun 1999 19:21:41 PDT.
Date: Fri, 02 Jul 1999 16:52:13 PDT
From: Vern Paxson <vern>

 file, it takes an enormous amount of time. It is funny, because the
 source code has only 12 rules!!! I think it looks like an exponencial
 growth.
```

Right, that's the problem - some patterns (those with a lot of ambiguity, where yours has because at any given time the scanner can be in the middle of all sorts of combinations of the different rules) blow up exponentially.

For your rules, there is an easy fix. Change the ".*" that comes fater the directory name to "[^]*". With that in place, the rules are no longer nearly so ambiguous, because then once one of the directories has been matched, no other can be matched (since they all require a leading blank).

If that's not an acceptable solution, then you can enter a start state to pick up the .*\n after each directory is matched.

Also note that for speed, you'll want to add a ".*" rule at the end, otherwise rules that don't match any of the patterns will be matched very slowly, a character at a time.

Vern

unnamed-faq-95

To: Tielman Koekemoer <tielman@spi.co.za>
Subject: Re: Please help.
In-reply-to: Your message of Thu, 08 Jul 1999 13:20:37 PDT.
Date: Thu, 08 Jul 1999 08:20:39 PDT
From: Vern Paxson <vern>

I was hoping you could help me with my problem.

I tried compiling (gnu)flex on a Solaris 2.4 machine
but when I ran make (after configure) I got an error.

```
---------------------------------------------------------------
gcc -c -I. -I. -g -O parse.c
./flex -t -p  ./scan.l >scan.c
sh: ./flex: not found
*** Error code 1
make: Fatal error: Command failed for target `scan.c'
---------------------------------------------------------------
```

What's strange to me is that I'm only
trying to install flex now. I then edited the Makefile to
and changed where it says "FLEX = flex" to "FLEX = lex"
(lex: the native Solaris one) but then it complains about
the "-p" option. Is there any way I can compile flex without
using flex or lex?

```
 Thanks so much for your time.

You managed to step on the bootstrap sequence, which first copies
initscan.c to scan.c in order to build flex.  Try fetching a fresh
distribution from ftp.ee.lbl.gov.  (Or you can first try removing
.bootstrap" and doing a make again.)

                Vern
```

unnamed-faq-96

```
To: Tielman Koekemoer <tielman@spi.co.za>
Subject: Re: Please help.
In-reply-to: Your message of Fri, 09 Jul 1999 09:16:14 PDT.
Date: Fri, 09 Jul 1999 00:27:20 PDT
From: Vern Paxson <vern>

 First I removed .bootstrap (and ran make) - no luck. I downloaded the
 software but I still have the same problem. Is there anything else I
 could try.

Try:

        cp initscan.c scan.c
        touch scan.c
        make scan.o

If this last tries to first build scan.c from scan.l using ./flex, then
your "make" is broken, in which case compile scan.c to scan.o by hand.

                Vern
```

unnamed-faq-97

```
To: Sumanth Kamenani <skamenan@crl.nmsu.edu>
Subject: Re: Error
In-reply-to: Your message of Mon, 19 Jul 1999 23:08:41 PDT.
Date: Tue, 20 Jul 1999 00:18:26 PDT
From: Vern Paxson <vern>

 I am getting a compilation error. The error is given as "unknown symbol- yylex".

The parser relies on calling yylex(), but you're instead using the C++ scanning
class, so you need to supply a yylex() "glue" function that calls an instance
scanner of the scanner (e.g., "scanner->yylex()").

                Vern
```

unnamed-faq-98

```
To: daniel@synchrods.synchrods.COM (Daniel Senderowicz)
Subject: Re: lex
In-reply-to: Your message of Mon, 22 Nov 1999 11:19:04 PST.
Date: Tue, 23 Nov 1999 15:54:30 PST
From: Vern Paxson <vern>

Well, your problem is the

switch (yybgin-yysvec-1) {      /* witchcraft */

at the beginning of lex rules.  "witchcraft" == "non-portable".  It's
assuming knowledge of the AT&T lex's internal variables.

For flex, you can probably do the equivalent using a switch on YYSTATE.

                Vern
```

unnamed-faq-99

```
To: archow@hss.hns.com
Subject: Re: Regarding distribution of flex and yacc based grammars
In-reply-to: Your message of Sun, 19 Dec 1999 17:50:24 +0530.
Date: Wed, 22 Dec 1999 01:56:24 PST
From: Vern Paxson <vern>

 When we provide the customer with an object code distribution, is it
 necessary for us to provide source
 for the generated C files from flex and bison since they are generated by
 flex and bison ?

For flex, no.  I don't know what the current state of this is for bison.

 Also, is there any requrirement for us to neccessarily  provide source for
 the grammar files which are fed into flex and bison ?

Again, for flex, no.

See the file "COPYING" in the flex distribution for the legalese.

                Vern
```

unnamed-faq-100

```
To: Martin Gallwey <gallweym@hyperion.moe.ul.ie>
Subject: Re: Flex, and self referencing rules
```

```
In-reply-to: Your message of Sun, 20 Feb 2000 01:01:21 PST.
Date: Sat, 19 Feb 2000 18:33:16 PST
From: Vern Paxson <vern>

 However, I do not use unput anywhere. I do use self-referencing
 rules like this:

 UnaryExpr               ({UnionExpr})|("-"{UnaryExpr})

You can't do this - flex is *not* a parser like yacc (which does indeed
allow recursion), it is a scanner that's confined to regular expressions.

                Vern
```

unnamed-faq-101

```
To: slg3@lehigh.edu (SAMUEL L. GULDEN)
Subject: Re: Flex problem
In-reply-to: Your message of Thu, 02 Mar 2000 12:29:04 PST.
Date: Thu, 02 Mar 2000 23:00:46 PST
From: Vern Paxson <vern>

If this is exactly your program:

 digit [0-9]
 digits {digit}+
 whitespace [ \t\n]+

 %%
 "[" { printf("open_brac\n");}
 "]" { printf("close_brac\n");}
 "+" { printf("addop\n");}
 "*" { printf("multop\n");}
 {digits} { printf("NUMBER = %s\n", yytext);}
 whitespace ;

then the problem is that the last rule needs to be "{whitespace}" !

                Vern
```

Appendix A Appendices

A.1 Makefiles and Flex

In this appendix, we provide tips for writing Makefiles to build your scanners.

In a traditional build environment, we say that the '`.c`' files are the sources, and the '`.o`' files are the intermediate files. When using `flex`, however, the '`.l`' files are the sources, and the generated '`.c`' files (along with the '`.o`' files) are the intermediate files. This requires you to carefully plan your Makefile.

Modern `make` programs understand that '`foo.l`' is intended to generate '`lex.yy.c`' or '`foo.c`', and will behave accordingly[1]. The following Makefile does not explicitly instruct `make` how to build '`foo.c`' from '`foo.l`'. Instead, it relies on the implicit rules of the `make` program to build the intermediate file, '`scan.c`':

```
# Basic Makefile -- relies on implicit rules
# Creates "myprogram" from "scan.l" and "myprogram.c"
#
LEX=flex
myprogram: scan.o myprogram.o
scan.o: scan.l
```

For simple cases, the above may be sufficient. For other cases, you may have to explicitly instruct `make` how to build your scanner. The following is an example of a Makefile containing explicit rules:

```
# Basic Makefile -- provides explicit rules
# Creates "myprogram" from "scan.l" and "myprogram.c"
#
LEX=flex
myprogram: scan.o myprogram.o
        $(CC) -o $@  $(LDFLAGS) $^

myprogram.o: myprogram.c
        $(CC) $(CPPFLAGS) $(CFLAGS) -o $@ -c $^

scan.o: scan.c
        $(CC) $(CPPFLAGS) $(CFLAGS) -o $@ -c $^

scan.c: scan.l
        $(LEX) $(LFLAGS) -o $@ $^

clean:
        $(RM) *.o scan.c
```

[1] GNU `make` and GNU `automake` are two such programs that provide implicit rules for flex-generated scanners.

Notice in the above example that '`scan.c`' is in the `clean` target. This is because we consider the file '`scan.c`' to be an intermediate file.

Finally, we provide a realistic example of a `flex` scanner used with a `bison` parser[2]. There is a tricky problem we have to deal with. Since a `flex` scanner will typically include a header file (e.g., '`y.tab.h`') generated by the parser, we need to be sure that the header file is generated BEFORE the scanner is compiled. We handle this case in the following example:

```
# Makefile example -- scanner and parser.
# Creates "myprogram" from "scan.l", "parse.y", and "myprogram.c"
#
LEX     = flex
YACC    = bison -y
YFLAGS  = -d
objects = scan.o parse.o myprogram.o

myprogram: $(objects)
scan.o: scan.l parse.c
parse.o: parse.y
myprogram.o: myprogram.c
```

In the above example, notice the line,

```
scan.o: scan.l parse.c
```

, which lists the file '`parse.c`' (the generated parser) as a dependency of '`scan.o`'. We want to ensure that the parser is created before the scanner is compiled, and the above line seems to do the trick. Feel free to experiment with your specific implementation of `make`.

For more details on writing Makefiles, see section "Top" in *The GNU Make Manual*.

A.2 C Scanners with Bison Parsers

This section describes the `flex` features useful when integrating `flex` with `GNU bison`[3]. Skip this section if you are not using `bison` with your scanner. Here we discuss only the `flex` half of the `flex` and `bison` pair. We do not discuss `bison` in any detail. For more information about generating `bison` parsers, see section "Top" in *the GNU Bison Manual*.

A compatible `bison` scanner is generated by declaring '`%option bison-bridge`' or by supplying '`--bison-bridge`' when invoking `flex` from the command line. This instructs `flex` that the macro `yylval` may be used. The data type for `yylval`, `YYSTYPE`, is typically defined in a header file, included in section 1 of the `flex` input file. For a list of functions and macros available, See ⟨undefined⟩ [bison-functions], page ⟨undefined⟩.

The declaration of yylex becomes,

[2] This example also applies to yacc parsers.

[3] The features described here are purely optional, and are by no means the only way to use flex with bison. We merely provide some glue to ease development of your parser-scanner pair.

```
int yylex ( YYSTYPE * lvalp, yyscan_t scanner );
```

If `%option bison-locations` is specified, then the declaration becomes,

```
int yylex ( YYSTYPE * lvalp, YYLTYPE * llocp, yyscan_t scanner );
```

Note that the macros `yylval` and `yylloc` evaluate to pointers. Support for `yylloc` is optional in `bison`, so it is optional in `flex` as well. The following is an example of a `flex` scanner that is compatible with `bison`.

```
/* Scanner for "C" assignment statements... sort of. */
%{
#include "y.tab.h"  /* Generated by bison. */
%}

%option bison-bridge bison-locations
%

[[:digit:]]+  { yylval->num = atoi(yytext);   return NUMBER;}
[[:alnum:]]+  { yylval->str = strdup(yytext); return STRING;}
"="|";"       { return yytext[0];}
.  {}
%
```

As you can see, there really is no magic here. We just use `yylval` as we would any other variable. The data type of `yylval` is generated by `bison`, and included in the file '`y.tab.h`'. Here is the corresponding `bison` parser:

```
/* Parser to convert "C" assignments to lisp. */
%{
/* Pass the argument to yyparse through to yylex. */
#define YYPARSE_PARAM scanner
#define YYLEX_PARAM   scanner
%}
%locations
%pure_parser
%union {
    int num;
    char* str;
}
%token <str> STRING
%token <num> NUMBER
%%
assignment:
    STRING '=' NUMBER ';' {
        printf( "(setf %s %d)", $1, $3 );
   }
;
```

A.3 M4 Dependency

The macro processor `m4`[4] must be installed wherever flex is installed. `flex` invokes '`m4`', found by searching the directories in the `PATH` environment variable. Any code you place in section 1 or in the actions will be sent through m4. Please follow these rules to protect your code from unwanted `m4` processing.

Do not use symbols that begin with, '`m4_`', such as, '`m4_define`', or '`m4_include`', since those are reserved for `m4` macro names.

Do not use the strings '`[[`' or '`]]`' anywhere in your code. The former is not valid in C, except within comments, but the latter is valid in code such as `x[y[z]]`.

`m4` is only required at the time you run `flex`. The generated scanner is ordinary C or C++, and does *not* require `m4`.

[4] The use of m4 is subject to change in future revisions of flex.

Indices

Concept Index

(Index is nonexistent)

Index of Functions and Macros

This is an index of functions and preprocessor macros that look like functions. For macros that expand to variables or constants, see ⟨undefined⟩ [Index of Variables], page ⟨undefined⟩.

(Index is nonexistent)

Index of Variables

This is an index of variables, constants, and preprocessor macros that expand to variables or constants.

(Index is nonexistent)

Index of Data Types

(Index is nonexistent)

Index of Hooks

This is an index of "hooks" that the user may define. These hooks typically correspond to specific locations in the generated scanner, and may be used to insert arbitrary code.

(Index is nonexistent)

Index of Scanner Options

(Index is nonexistent)