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

**[Some simple examples](http://dinosaur.compilertools.net/flex/index.html" \l "SEC5)**

First some simple examples to get the flavor of how one uses flex. The following flex input specifies a scanner which whenever it encounters the string "username" will replace it with the user's login name:

%%

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

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

Here's another simple example:

int num\_lines = 0, num\_chars = 0;

%%

\n ++num\_lines; ++num\_chars;

. ++num\_chars;

%%

main()

{

yylex();

printf( "# of lines = %d, # of chars = %d\n",

num\_lines, num\_chars );

}

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

A somewhat more complicated example:

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

%{

/\* need this for the call to atof() below \*/

#include <math.h>

%}

DIGIT [0-9]

ID [a-z][a-z0-9]\*

%%

{DIGIT}+ {

printf( "An integer: %s (%d)\n", yytext,

atoi( yytext ) );

}

{DIGIT}+"."{DIGIT}\* {

printf( "A float: %s (%g)\n", yytext,

atof( yytext ) );

}

if|then|begin|end|procedure|function {

printf( "A keyword: %s\n", yytext );

}

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

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

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

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

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

%%

main( argc, argv )

int argc;

char \*\*argv;

{

++argv, --argc; /\* skip over program name \*/

if ( argc > 0 )

yyin = fopen( argv[0], "r" );

else

yyin = stdin;

yylex();

}

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

The details of this example will be explained in the following sections.

## [Format of the input file](http://dinosaur.compilertools.net/flex/index.html" \l "SEC6)

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

definitions

%%

rules

%%

user code

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

name definition

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

DIGIT [0-9]

ID [a-z][a-z0-9]\*

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

{DIGIT}+"."{DIGIT}\*

is identical to

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

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

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

pattern action

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

See below for a further description of patterns and actions.

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

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

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

In the definitions section (but not in the rules section), an unindented comment (i.e., a line beginning with "/\*") is also copied verbatim to the output up to the next "\*/".

## [Patterns](http://dinosaur.compilertools.net/flex/index.html" \l "SEC7)

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

`x'

match the character `x'

`.'

any character (byte) except newline

`[xyz]'

a "character class"; in this case, the pattern matches either an `x', a `y', or a `z'

`[abj-oZ]'

a "character class" with a range in it; matches an `a', a `b', any letter from `j' through `o', or a `Z'

`[^A-Z]'

a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.

`[^A-Z\n]'

any character EXCEPT an uppercase letter or a newline

`r\*'

zero or more r's, where r is any regular expression

`r+'

one or more r's

`r?'

zero or one r's (that is, "an optional r")

`r{2,5}'

anywhere from two to five r's

`r{2,}'

two or more r's

`r{4}'

exactly 4 r's

`{name}'

the expansion of the "name" definition (see above)

`"[xyz]\"foo"'

the literal string: `[xyz]"foo'

`\x'

if x is an `a', `b', `f', `n', `r', `t', or `v', then the ANSI-C interpretation of \x. Otherwise, a literal `x' (used to escape operators such as `\*')

`\0'

a NUL character (ASCII code 0)

`\123'

the character with octal value 123

`\x2a'

the character with hexadecimal value 2a

`(r)'

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

`rs'

the regular expression r followed by the regular expression s; called "concatenation"

`r|s'

either an r or an s

`r/s'

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

`^r'

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

`r$'

an r, but only at the end of a line (i.e., just before a newline). Equivalent to "r/\n". Note that flex's notion of "newline" is exactly whatever the C compiler used to compile flex interprets '\n' as; in particular, on some DOS systems you must either filter out \r's in the input yourself, or explicitly use r/\r\n for "r$".

`<s>r'

an r, but only in start condition s (see below for discussion of start conditions) <s1,s2,s3>r same, but in any of start conditions s1, s2, or s3

`<\*>r'

an r in any start condition, even an exclusive one.

`<<EOF>>'

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

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

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

foo|bar\*

is the same as

(foo)|(ba(r\*))

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

foo|(bar)\*

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

(foo|bar)\*

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

[:alnum:] [:alpha:] [:blank:]

[:cntrl:] [:digit:] [:graph:]

[:lower:] [:print:] [:punct:]

[:space:] [:upper:] [:xdigit:]

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

For example, the following character classes are all equivalent:

[[:alnum:]]

[[:alpha:][:digit:]

[[:alpha:]0-9]

[a-zA-Z0-9]

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

Some notes on patterns:

* A negated character class such as the example "[^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 '$' which does not occur at the end of a rule loses its special properties and is treated as a normal character. The following are illegal:
* foo/bar$
* <sc1>foo<sc2>bar

Note that the first of these, can be written "foo/bar\n". The following will result in '$' or '^' being treated as a normal character:

foo|(bar$)

foo|^bar

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

foo |

bar$ /\* action goes here \*/

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

## [How the input is matched](http://dinosaur.compilertools.net/flex/index.html" \l "SEC8)

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

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

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

%%

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

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

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

extern char yytext[];

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

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

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

**[Actions](http://dinosaur.compilertools.net/flex/index.html" \l "SEC9)**

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

%%

"zap me"

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

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

%%

[ \t]+ putchar( ' ' );

[ \t]+$ /\* ignore this token \*/

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

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

Actions can include arbitrary C code, including return statements to return a value to whatever routine called `yylex()'. Each time `yylex()' is called it continues processing tokens from where it last left off until it either reaches the end of the file or executes a return.

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

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

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

* `ECHO' copies yytext to the scanner's output.
* BEGIN followed by the name of a start condition places the scanner in the corresponding start condition (see below).
* REJECT directs the scanner to proceed on to the "second best" rule which matched the input (or a prefix of the input). The rule is chosen as described above in "How the Input is Matched", and yytext and yyleng set up appropriately. It may either be one which matched as much text as the originally chosen rule but came later in the flex input file, or one which matched less text. For example, the following will both count the words in the input and call the routine special() whenever "frob" is seen:
* int word\_count = 0;
* %%
* frob special(); REJECT;
* [^ \t\n]+ ++word\_count;

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

%%

a |

ab |

abc |

abcd ECHO; REJECT;

.|\n /\* eat up any unmatched character \*/

(The first three rules share the fourth's action since they use the special '|' action.) REJECT is a particularly expensive feature in terms of scanner performance; if it is used in *any* of the scanner's actions it will slow down *all* of the scanner's matching. Furthermore, REJECT cannot be used with the `-Cf' or `-CF' options (see below). Note also that unlike the other special actions, REJECT is a *branch*; code immediately following it in the action will *not* be executed.

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

First "mega-" is matched and echoed to the output. Then "kludge" is matched, but the previous "mega-" is still hanging around at the beginning of yytext so the `ECHO' for the "kludge" rule will actually write "mega-kludge".

Two notes regarding use of `yymore()'. First, `yymore()' depends on the value of yyleng correctly reflecting the size of the current token, so you must not modify yyleng if you are using `yymore()'. Second, the presence of `yymore()' in the scanner's action entails a minor performance penalty in the scanner's matching speed.

* `yyless(n)' returns all but the first *n* characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. yytext and yyleng are adjusted appropriately (e.g., yyleng will now be equal to *n* ). For example, on the input "foobar" the following will write out "foobarbar":
* %%
* foobar ECHO; yyless(3);
* [a-z]+ ECHO;

An argument of 0 to yyless will cause the entire current input string to be scanned again. Unless you've changed how the scanner will subsequently process its input (using BEGIN, for example), this will result in an endless loop. Note that yyless is a macro and can only be used in the flex input file, not from other source files.

* `unput(c)' puts the character c back onto the input stream. It will be the next character scanned. The following action will take the current token and cause it to be rescanned enclosed in parentheses.
* {
* int i;
* /\* Copy yytext because unput() trashes yytext \*/
* char \*yycopy = strdup( yytext );
* unput( ')' );
* for ( i = yyleng - 1; i >= 0; --i )
* unput( yycopy[i] );
* unput( '(' );
* free( yycopy );
* }

Note that since each `unput()' puts the given character back at the *beginning* of the input stream, pushing back strings must be done back-to-front. An important potential problem when using `unput()' is that if you are using `%pointer' (the default), a call to `unput()' *destroys* the contents of yytext, starting with its rightmost character and devouring one character to the left with each call. If you need the value of yytext preserved after a call to `unput()' (as in the above example), you must either first copy it elsewhere, or build your scanner using `%array' instead (see How The Input Is Matched). Finally, note that you cannot put back EOF to attempt to mark the input stream with an end-of-file.

* `input()' reads the next character from the input stream. For example, the following is one way to eat up C comments:
* %%
* "/\*" {
* register int c;
* for ( ; ; )
* {
* while ( (c = input()) != '\*' &&
* c != EOF )
* ; /\* eat up text of comment \*/
* if ( c == '\*' )
* {
* while ( (c = input()) == '\*' )
* ;
* if ( c == '/' )
* break; /\* found the end \*/
* }
* if ( c == EOF )
* {
* error( "EOF in comment" );
* break;
* }
* }
* }

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

* YY\_FLUSH\_BUFFER flushes the scanner's internal buffer so that the next time the scanner attempts to match a token, it will first refill the buffer using YY\_INPUT (see The Generated Scanner, below). This action is a special case of the more general `yy\_flush\_buffer()' function, described below in the section Multiple Input Buffers.
* `yyterminate()' can be used in lieu of a return statement in an action. It terminates the scanner and returns a 0 to the scanner's caller, indicating "all done". By default, `yyterminate()' is also called when an end-of-file is encountered. It is a macro and may be redefined.

<http://dinosaur.compilertools.net/flex/flex_13.html>

Start of Yacc:

<https://tldp.org/HOWTO/Lex-YACC-HOWTO-6.html>