Lex and Flex

From the Flex man page

In order for Lex/Flex to recognize patterns in text, the pattern must be described by a *regular expression*. The input to Lex/Flex is a machine readable set of regular expressions. The input is in the form of pairs of regular expressions and C code, called rules. Lex/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.

Lex/Flex Examples

The following Lex/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 Lex/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.

The Lex/Flex Input File

The Lex/Flex input file consists of three sections, separated by a line with just %% in it:

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

The Declarations Section

The *definitions* section contains declarations of simple *name* definitions to simplify the scanner specification.

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

The *rules* section of the Lex/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 \underline{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.

In the definitions section, an unindented comment (i.e., a line beginning with ``/*") is also copied verbatim to the output up to the next ``*/".

Lex/Flex Patterns

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

x match the character `x'
. any character except newline
[xyz] a ``character class"; in this case, the pattern matches either an 'x', a 'y', or a `z'
[abj-oZ] a ``character class" with a range in it; matches an `a', a `b', any letter from `j' through `o', or a `Z'
[^A-Z] a ``negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.
[^A-Z\n] any character EXCEPT an uppercase letter ora 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 s is not part of the matched text. This type of pattern is called as ``trailing context''.
^r an r, but only at the beginning of a line
r$ an r, but only at the end of a line. Equivalent to ``r/\n''.
<s>r an r, but only in start condition s
<s1,s2,s3>r an r in any of start conditions s1, s2, s2
...

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

```
foo|bar*
```

is the same as

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

since the `*' operator has higher precedence than concatenation, and concatenation higher than alternation (`|'). This pattern therefore matches either the string ``foo'' or the string ``ba'' followed by zero-or-more 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)*
```

A note on patterns: A negated character class such as the example [^ A-Z] above *will match a newline* unless "\n" (or an equivalent escape sequence) is one of the characters explicitly present in the negated character class (e.g., [^ A-Z\n]). This is unlike how many other regular expression tools treat negated character classes, but unfortunately the inconsistency is historically entrenched. Matching newlines means that a pattern like [^"]* can match an entire input (overflowing the scanner's input buffer) unless there's another quote in the input.

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 Lex/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 Lex/Flex input is:

```
%%
```

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

Lex/Flex Actions

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

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

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

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

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

If the action contains a `{', then the action spans till the balancing `}' is found, and the action may cross multiple lines. Lex/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).

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

Actions are not allowed to modify yytext or yyleng.

The Code Section

The code section contains the definitions of the routines called by the action part of a rule. This section also contains the definition of the function main if the scanner is a stand-alone program.

The Generated Scanner

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

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

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

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