FLEX

Lex (flex) is a tool for building *lexical analyzers* or *lexers*.

When you write a *lex specification*, you create a set of patterns which lex matches against input.

Each time one of the patterns matches, the lex specification invokes C code that you provide, which does something with the matched text.

Layout of a Lex Input file:

```
Definitions
%%
Rules
%%
User Code
```


Example 1

```
%%
#.*\n  ;
```


Here we have nothing in the definitions and user code section.

1. Save the Flex specification in a text file, say `first.flex`.

2. Convert the Flex program file to a C program with the command `flex first.flex`. If there is no error, a file named `lex.yy.c` will exist.

3. Compile this C file
   (`gcc lex.yy.c -lfl -o first`.)

4. Execute first (type in first $<$ input $>$ output.)

With this lexer we will remove from the input file each occurences of a number sign (#) along with anything that comes after it on the same line.

Example 2

```
%{
/* a Lex program that adds line numbers
to lines of text, printing the new text
to standard output
*/
#include<stdio.h>
int lineno = 1;
%}
line .*\n
%%
{line} {printf("%5d %s",lineno++,yytext);}
%%
main()
{yylex();}
```


The Flex notation for symbol patterns extend REs by permitting some useful operations that are not part of the core definitions of REs.

Briefly, flex works like this:

- Flex searches for the longest string that fits any of your LEs (lex expressions).
- The search is in some input text, like a program or document.
- You specify in C what happens when an input string fits a pattern.

- If a character is not matched as part of a specified LE, it is simply copied to the output. Thus in effect there is a default rule: $.|\backslash n$ { ECHO;} that is added as the last rule.
- If two or more patterns are tied for the longest match, the one occurring earliest is used.

$*$ Star in Flex stands for zero or more occurences of its opperand.

| The vertical bar seperates alternatives, instead of $\cup$.

() Parentheses are used in the ordinary way for grouping. They do not add any meaning.

$+$ Plus means one or more occurences of whatever it is applied to.

? A question mark after something makes it optional, so b? stands for $(\varepsilon \cup b)$.

{} Braces around a number indicates that something should be repeated that number of times, for example $[A-Z]\{1,8\}$ matches 1 to 8 capital letters.

[] Brackets denote a choice among characters.
$[aeiou]$ means any vowel just like $(a|e|i|o|u)$.

Inside brackets most special symbols lose their special meaning.
$[*/]$ represents a choice between star and slash.
[.?!] denotes a choice among sentence-ending punctuation.

Characters with consecutive ASCII codes can be expressed with a hyphen in brackets.
$[a-z]$ is a pattern for lowercase letters.
$[a-zA-Z]$ is a pattern for any letter.

$[-+]$ denotes a choice of sign.

$[-+]$? stands for an optional sign.

^ When ^ appears at the start of a bracketed pattern, it negates the remaing characters.

$[^aeiou]+$ means a sequence of one or more symbols that are not vowels.

. A period matches any single character except newline.
$.*$ matches an arbitrary sequence within a line.

{} Braces surround a defined term to invoke its definition.

$\$\{D\} + \.\{D\}\{2\}$ match an amount of dollars and cents if D has been defined as $[0-9]$ in the definition section.

\ Backslash before an operator makes it behave like an ordinary character.

\+ matches a + sign.

\. matches a period.

$\backslash t$ However, just like in C, this stands for tab.

$\backslash n$ matches newline.

[$\backslash t \backslash n$]+ matches whitespace across line boundaries.

[$\hat{}\backslash n \backslash t$]+ matches one or more non-whitespace characters.

" " Double quotes are used in pairs to make the included characters lose their special status.

$\backslash$"[$\hat{}$"$\backslash n$]$*$["$\backslash n$] matches quoted material up to the end of a line.

$\hat{}$ A caret used outside brackets at the start of a pattern requires the matching pattern to appear at the start of the input line.

$ A dollar sign at the end of a pattern requires the material matching the pattern to appear at the end of the input line.

/ Allows a pattern to stipulate a right-hand context.

ab/cd matches ab if and only if the next two characters are cd. Only ab is used up.

EXAMPLE

Find hexadecimal numbers flagged by an x or X and print them out.

```
   int i;
H [0-9a-fA-F]
%%
[xX]({ H })+   {for (i=0;i<yyleng;i++)
               yytext[i]=yytext[i+1];
               printf("%s\n",yytext);}

.|\n   ;
%%
main()
{
   yylex();
}
```

STATES in Flex

- States are used in a manner directly motivated by finite automata.
- We begin in a state which is by default called INITIAL.
- The transitions are in response to the special action BEGIN in the C code.
- When SOMESTATE is the current state, the only active patterns are those that have no state specified or begin with <SOMESTATE>.

- To declare an ordinary state called STATENAME, put %s STATENAME in the definition section.
- You can also use %x STATENAME to declare an *exclusive* state. When you are in an exclusive state, only patterns explicitly specifying that state can be used for matching.
- When processing begins, the state is assumed to be a state called INITIAL that is not declared.
- Execution of BEGIN STATEMENT changes the state to STATENAME.

Example

Write a Flex program that will replace comments in C with " comment begun - comment ended ", and will leave the rest unchanged.

```
%x COMMENT
%x HALFOUT

%%

"/*"    { BEGIN COMMENT;
      printf(" comment begun - ");}
<COMMENT>\*    BEGIN HALFOUT;
<COMMENT>[^*]    ;
<HALFOUT>\*    ;
<HALFOUT>\/    { printf(" comment ended
      "); BEGIN INITIAL;}
<HALFOUT>[^*/]    BEGIN COMMENT;
```


Same example, one state less:

```
%x COMMENT

%%

"/*"    { BEGIN COMMENT;
      printf(" comment begun - ");}
<COMMENT>.|\n ;
<COMMENT>"*/" { BEGIN INITIAL;
      printf(" comment ended ");}
```