An Introduction to Flex

`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.

Our First Flex Example:

```
%{
#include <stdio.h>

int chars=0;
int lines=0;
%}
%%
\n {lines++; chars++;}
. {chars++;}
%%
int main(void)
{
    yylex();
    printf("Char count: %d, Line Count: %d\n", chars, lines);
    return 0;
}
```

Format of the Flex Source Code:

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

Definitions:

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.

Rules:

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.

User Code:

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.

Patterns:

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

Pattern	**Description**
`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'
`[a-z]`	a "character class" with a range in it; any letter from ‘a’ to ‘z’
`[a-zA-Z]`	any alphabetical character
`[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
`[a-z]{-}[aeiou]`	the lowercase consonants
`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

`(r)`	match an ‘r’; parentheses are used to override precedence
`rs`	the regular expression ‘r’ followed by the regular expression ‘s’; called concatenation
`r|s`	either an ‘r’ or an ‘s’
`^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’

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, 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.

Practicing all this together:

```
%{
#include <iostream>
#include <cstring>

using namespace std;

int lines = 0;
int words = 0;
int chars = 0;
%}
%%
[a-zA-Z]+ {
    words += 1;
    chars += strlen(yytext);
}
[^ \t\n\f\v\r]+ {
    words += 1;
    chars += strlen(yytext);
}
"\n" {
    lines += 1;
    chars += 1;
}
. {
    chars += 1;
}
%%
int main(int argc, char **argv)
{
    yylex();
    cout<<"Line Count: "<<lines<<endl;
    cout<<"Word Count: "<<words<<endl;
    cout<<"Char Count: "<<chars<<endl;
    return 0;
}
```

A more complex example:

```
%{
#include <stdio.h>
#include <math.h>

int lineCount = 0;
%}

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] {
      lineCount += 1;
}
[ \t]+ {
      //do nothing
}

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

%%

int main(void)
{
      yylex();
      return 0;
}
```