A Flex Primer

Flex is a preprocessor. Flex can generate C or C++ compatible code. It is essentially the same program as Lex with a few changes.

A Flex program has 3 sections: definitions, rules, user subroutines. Each section is separated by a pair of percents signs (%) in the first column. The form is roughly like:

```
definitions
%%
rules
%%
user subroutine section
```

The Definition Section

In the definitions section you can have a region of text that is explicitly copied to the output. This section usually contains includes, global variables declarations and prototypes of functions declared in the user subroutine section. This region of copied code is contained between %{ and %} **Warning:** It is NOT %{ and }%!:

```
%{
code to copy into the final program
%}
```

After this can come pattern macros and few other options.

Pattern Macros are defined by giving a name followed by the definition. For example: name [a-z][a-z0-9]*

The Rules Section

The rules section contains pattern/action pairs.

```
pattern action
pattern action
pattern action
```

The user subroutine section, of course, contains user subroutines and the main program which usually invokes the yylex subroutine.

Patterns are regular expressions for matching strings.

	Any character that does not have special meaning matches itself.
\t	matches tab
\n	matches newline
.	dot matches any char except a newline
\	is used to make any special character lose its special status and match just the character, except in the special cases where the \ character has a meaning in C such as tab or newline.
*	matches 0 or more copies of the pattern preceding the *
+	matches 1 or more copies of the previous pattern
?	matches 0 or 1 copies of the previous pattern

{x,y}	x through y copies of the previous patterns
[ ]	character set. If (^) is the first character it is the inverse or complement of the character set about to be specified. - in between two characters in a character set means range of characters as in [a-z] means the lowercase letters and [aeiou] means the vowels and [a-zA-Z] means all letters. Note that if you use the not character (^) that the resulting character class pattern **will match \n**
/	means "match if followed by" for example dog/cat match the 3 characters "dog" if followed by the three characters "cat". "cat" is not actually matched.
$	at the end of a pattern matches at the end of line
^	at the beginning of a pattern matches beginning of line
|	alternation. cat|dog means cat or dog.
""	are takes the contents of the string as characters and not as special pattern characters.
( )	grouping. (cat|dog)+ is one or more occurrances of cat or dog.
{*name*}	is the invocation of the pattern macro named *name* that was defined in the definition section.

Warning: Anytime you want to match a string that contains any suspicious characters such as say > or < you should enclose the constant characters in quotes. For example: dogs"<"[0-9] matches the word dogs followed by < followed by a digit. See A Regular Expression Primer for a comparison with other regular expression using tools.

The **four rules for matching tokens** are:

1. characters are only matched once. That is, each character is matched by only one pattern.
2. longest matching string gets matched first. That is, if one pattern matches "zin" and a later pattern matches "zinjanthropus" the second pattern is the one that matches.
3. if same length of matching string then first rule matches.
4. if no pattern matches then the character is printed to standard output.

The User Subroutine Section

Simply put other routines that you want here including any routines that refer to routines created by Flex such as a main which refers to yylex. See below.

Example

The results of compiling the program with flex will be to create several variables and a scanner routine called yylex which can be called as in the example below. A example program to print out all of the numbers and words in an input file:

```
%{
#include <stdio.h>
%}
letter [a-zA-Z]
%%
[0-9]+ { printf("NUM:  %s\n", yytext); }
{letter}+ { printf("WORD: %s\n", yytext); }
.|\n     ;
%%
main()
{
        yylex();
}
```

If this lex file is stored in **numwords.l** then this can be compiled by using:

```
flex -d  numwords.l   # -d turns on tracing for *hardcore* debugging
gcc -g lex.yy.c -lfl -o numwords
```

This will compile the lex file using flex to generate the lex.yy.c file. The **-d** puts in code to turn on tracing in your scanner. Most of the time you do not want or need this option. Then **gcc** with the debug option **-g** and adding in the flex library **-lfl** and putting the program in **numwords** using the **-o** option. In this case **g++** would work as well.

A makefile for Flex Only

```
BIN  = t0   # name of thing to be built goes here. e.g. this builds t0
CC   = g++
CFLAGS = -g -DCPLUSPLUS

SRCS = $(BIN).l
OBJS = lex.yy.o
LIBS = -lfl

$(BIN): $(OBJS)
        $(CC) $(CCFLAGS) $(OBJS) $(LIBS) -o $(BIN)

lex.yy.c: $(BIN).l
        flex $(BIN).l

clean:
        rm -f $(OBJS) $(BIN) lex.yy.c $(BIN).tab.h $(BIN).tab.c
```

Further Reading

The Flex Manual

Further Flex Examples

Here are some more examples.

Count the number of characters, words, and lines coming from stdin.

```
%{
#include <stdio.h>
int c=0, w=0, l=0;
%}

word [^ \t\n]+
eol  \n

%%
{word} {w++; c+=yyleng;};
{eol}  {c++; l++;}
.      {c++;}
%%
main()
{
        yylex();
        printf("%d %d %d\n", l, w, c);
}
```

Renumber all numbers in stdin.

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

int counter = 0;               ← why here and not in main
%}
%%
[0-9]* { cout << counter++; }  ← why no endl?
%%
main()
{
    yylex();
}
```

Print out all HTTP tags.

```
%{
#include <stdio.h>
%}
%%
"<"[^>]*> { printf("VALUE: %s\n", yytext); }   ← stretches across \n
.|\n     ;
%%
main()
{
        yylex();
}
```
