BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
HYDERABAD CAMPUS
SECOND SEMESTER 2016 – 2017
Compiler Construction (CS F363)
Lab Sheet – 3

1. Objective of this lab is
    - To experiment with different routines to reprocess input
    - To experiment with different macros enable you to perform special actions

2. Background:

There are number of special directives which can be included within an action. Directives, like keywords in C, are those words whose meaning has been already predefined in Lex tool. Mainly we have three directives in Lex.

1. ECHO:- It copies yytext to the scanner's output. That is, whatever token we have recently found will be copied to the output.
2. BEGIN:- The directive BEGIN followed by the name of the start symbol, places the scanner in the corresponding rules. Lex activates the rules using the directive BEGIN and a start condition.

   To illustrate the uses of start conditions, here is a scanner which provides two different interpretations of a string like "123.456". By default it will treat it as three tokens, the integer "123", a dot ('.'), and the integer "456". But if the string is preceded earlier in the line by the string "expect-floats" it will treat it as a single token, the floating-point number 123.456:

```
%{
#include <math.h>
%}
%s expect
%%
expect-floats           BEGIN(expect);

<expect>[0-9]+"."[0-9]+     {
                            printf( "found a float, = %f\n",
                             atof( yytext ) );
                                    }
<expect>\n                  {
            /* that's the end of the line, so
                * we need another "expect-number"
                * before we'll recognize any more
                * numbers
                */
            BEGIN(INITIAL);
            }

[0-9]+      {
```

```
        printf( "found an integer, = %d\n",
                atoi( yytext ) );
        }

"."     printf( "found a dot\n" );
```

3. REJECT:- It directs the scanner to proceed on to the "scanned best" rule which matched the input (or a prefix of the input). That is, as soon as REJECT statement is executed in the action part, the last letter, will be treated (or prefixed) from the recently matched token and will go ahead with the prefixed input for next best rule.

   Example lex code

```
%%
        pink  {npink++; REJECT;}
        ink   {nink++; REJECT;}
        pin   {npin++; REJECT;}
        .     |
        \n    ;  /*discard other characters */
```

   if the input contains the word "pink" all three patterns will match. Without REJECT statements, only "pink" would match. Scanners that use REJECT are much larger and slower than those that don't, since they need considerable extra information to allow backtracking and re-lexing.

The lex library contains the following subroutines:

main() Invokes the lexical analyser by calling the yylex subroutine.

yywrap() Returns the value 1 when the end of input occurs.

yymore() Appends the next matched string to the current value of the yytext array rather than replacing the contents of the yytext array.

```
%%
bits-    ECHO; yymore();
hyd   ECHO;
```

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

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.

Consider a character string bounded by B's and interspersed with one ``B'' at an arbitrary location. For example:

Exercise:
BoomBoxB
You may want to count the number of characters between the first and second ``B'' and add it to the number of characters between the second and third ``B'', and print the result. (The last ``B'' is not to be counted.)
Code to do this is:

```
%{
  int flag=0;
  %}
  %%
  B[^B]*    { if (flag == 0) {
              flag = 1;
              yymore();
            }
            else     {
              flag = 0;
              printf("%d\n",yyleng);
             }
           }
```

The variable flag is used to distinguish the character sequence terminating just before the second ``B'' from that terminating just before the third. Using the example input ``BoomBoxB'', the pattern first matches the string ``Boom'', causing that string to be put into yytext. Next, the pattern matches the string ``Box'', which would normally cause only that string to be put into yytext. However, yymore() was called in the action associated with the previous pattern match, so yytext contains ``BoomBox'', and yyleng consequently equals 7.

yyless() trim characters from yytext and puts them back on stdin

`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.

The function yyless() resets the end point of the current token. yyless() takes a single integer argument: yyless(n) causes the current token to consist of the first n characters of what was originally matched (that is, up to yytext[n-1]). The remaining yyleng-n characters are returned to the input stream. Consider the following specification:

```
  %%
```

```
[A-Z][a-z]*     {if (yyleng>5)
                yyless(5);
            }
[a-z]*          printf("%s",yytext);
```

The lexical analyzer generated from this specification removes the first 5 letters from any word that starts with an uppercase letter.

yyreject() Allows the lexical analyser to match multiple rules for the same input string. (yyreject is called when the special action REJECT is used.)

Some actions may require reading another character, putting a character back into the input stream, or writing a character to the standard output. lex supplies three macros to handle these
tasks

- input - Returns the next input character
- output(c) - Writes the character c on the output
- unput(c) - Pushes the character c back onto the input stream to be read later by input

`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.

Exercises to try

1. Write a Lex Program to match extract string literal “bits\”hyd”
2. Write a Lex program to check whether any statement has missing parenthesis and also if a C file has missing parenthesis.
3. Write a Lex program which counts number of words in a file other than the word “incl”
4. Write a Lex program to recognize the following tokens in a C file

- Keywords
- Identifiers
- Operators
- String literal
- Floating point literal
- Integer literal
- Character literal
- Special symbols

.

**********That's all folks**********