```
%%
"un" {      /* First rule that matches un */
       printf("\nThe unput char =" );
       ECHO;
       }
[a-z]+  {   /* Second Rule */
       printf("\nThe lower case token is =");
       ECHO;unput('n');unput('u');
       printf("\nThe token after unput =");ECHO;
       }
[a-zA-Z]+  {   /* Third rule */
              printf("\nThe mixed token is =");
              ECHO;
              }
%%
main()
{
  yylex();
}
```

```
---------------------------------- output ----------------------------------

[root@vinu yytextTest]# ./a.out
good Day

The lower case token is = good
The token after unput = goun
The unput char = un
The mixed token is = Day

[1]+ Stopped ./a.out
[root@vinu yytextTest]#
-----------------------------------------------------------------------------
```

For a given input *good Day*, the second rule matches the token *good* and the following action statements are executed including *unput(n)* and *unput(u)*. It will return the character *n* & *u* back to the input stream to be scanned again. The token *n* & *u* will be matched by the first rule to proceed to the next of the input stream, which is *Day*, matched by the third rule.

Note that since each *unput()* puts the given character back to the beginning of the input stream, pushing back strings must be done back-to-front. That is, after the execution of *unput(n)*, *yytext* changed from *good* to *goon* replacing the last character of the token *good* to *n*, as you can see in the output. The *unput(u)* has changed the second character from the right to *u*, that is, the content of *yytext* is changed from *goon* to *goun*. The unput character *nu* must be put back to the input stream by passing back-to-front.

An important potential problem while using *unput()* is that if you are defining the *yytext* using the directives, *%pointer* (by default) will destroy the contents of *yytext*,

starting with its right most character and devouring one character to the left with each *unput()* call. If you need the value of *yytext* to be preserved after a call to *unput()*, you must either first copy it elsewhere, or build the Lex program using *%arrays* to define *yytext* in the definition section. This prevents destroying the present content of the *yytext*. Program 3.13 explains the same.

PROGRAM 3.13

```
---------------------------------------------------------------------------
%{
//*************************************************************************
//FLEX program that check unput() without destroying the
//content of yytext, by re-declaring the yytext variable as arrays
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************

#define YYLMAX 10
%}

%array yytext

%%

"un" {  printf("\nThe unput char =");
        ECHO;
        }

[a-z]+ {printf("\nThe lower case token is = "); ECHO;
           unput('n');
           printf("\nThe token after first unput = ");ECHO;
           unput('u');
           printf("\nThe token after second unput = ");ECHO;
           }

[a-zA-Z]+ {   printf("\nThe mixed token is =");
              ECHO;
       }

%%

main()
{
  yylex();
}
```

```
---------- output ----------
[root@vinu yytextTest]# ./a.out
good Day
The lower case token is = good
The token after first unput = good
The token after second unput = good
The unput char = un
The mixed token is = Day
[2]+ Stopped ./a.out
[root@vinu yytextTest]#
```

#define YYLMAX 10 statement, declaring the size of the character variable *yytext* to 10, and the statement *%array yytext* defining the *yytext* as an array. For a given input good Day, the second rule matches the token *good* and the following action statements are entered from the output that even after the execution of the *unput()*, the content of the *yytext* is preserved.

3.8.4 input()

input() reads the next character from the input stream. The read character will not be made available to the scanner. Program 3.14 illustrates the concept of *input()*.

PROGRAM 3.14

```
%{
//*************************************************************************
//FLEX program to match mixed letters of type
//a to z, A to Z and 0 to 9 & eat up the C comment lines
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************
%}
%%
[a-zA-Z0-9]+ {   printf("\nIt contains mixed letters =");
                 ECHO;
             }
"/*" {printf("\nThe comment begins");
      char c;
      while( (c=input()) != '*');
```

```
        if( (c=input()) == '/')
              printf("\nThe comment ends");
    }
%%

main()
{
  yylex();
}
```

---------- output ----------

```
[root@localhost inputTest]# ./a.out
This program is coded by /* the Author */

It contains mixed letters = This
It contains mixed letters = program
It contains mixed letters = is
It contains mixed letters = coded
It contains mixed letters = by
The comment begins
The comment ends

[3]+ Stopped ./a.out
[root@localhost inputTest]#
```

The first rule in this program matches the token that consists of lower and upper case letters and digits. The second rule will match whenever a '/*' is found in the input stream. The *input()* function will execute until the '*' is found, this is one way to eat up C comments.

This program gets the input *'This program is coded by /* the Author */'* and the first five words are matched by the first rule. When "/*" is encountered in the input, the second rule will be matched and its action statements will be executed, where it accepts all the characters using the function *input()* and does nothing until it finds "*/".

In effect, the second rule will be matched when any C comment begins (with "/*") and the function input() reads all the next character(s) from the input stream until it finds "*/" in the input, which is same as eat up C comments.

3.8.5 yyterminate()

yyterminate() can be used in lieu of a return statement in an action. *yyterminate()* terminates the execution of the scanner and returns a 0 to the function where the scanner is called, indicating that "all done" (see Program 3.15).

PROGRAM 3.15

```
%{
//*************************************************************************
//FLEX program that checks the function of yyterminate()
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************
%}
%%
[a-z]+ {    printf("\nIts a lower case letter =");
            ECHO;
            printf("\nBeginning the yyterminate");
            yyterminate();
            printf("\nEnd of yyterminate\n");
       }
[a-zA-Z]+ {    printf("\n\nMixed case token =");
               ECHO;
               }
%%
main()
{
  yylex();
}
```

```
---------------------------------- output ----------------------------------
[root@localhost yymmoreTest]# ./a.out
Good morning

Mixed case token = Good
It is a lower case letter = morning
Beginning the yyterminate

[1]+ Stopped ./a.out
[root@localhost yymmoreTest]#
----------------------------------------------------------------------------
```

Program 3.15 gets the input *Good morning*. Remember that the input *Good morning* matches three tokens and not two. The first token *Good* matched by the second rule (i.e. [a–zA–Z]+) and the following action statements are executed. The second token while spacing between the input stream *Good* and *morning* was supposed to be matched with ". | \n" (which is what Flex default rule matches a single character of any type). But we can even explicitly write a rule to match the white space. The third token *morning* matched by the first rule (i.e. [a–z]+) and the following statements are executed

including *yyterminate()*. Note that as soon as *yyterminate* executes, it quits all the executions and returns to the scanners caller. By default *yyterminate()* is also called when an end of file is encountered. It is a macro and may be redefined.

3.8.6 YY Flush Buffer

YY FLUSH BUFFER flushes the scanner's internal buffer so that the next time the scanner attempts to match a token, it will first refill the buffer using YY_INPUT (YYINPUT is discussed along with the next topic). This action is a special case of the more general *yy_flush_buffer()* function.

Program 3.16 describes how a Flex program behaves in ECHO and prints the *yytext* when we use *yy_flush_buffer*.

PROGRAM 3.16

```
---------------------------------------------------------------------------
%{
//************************************************************************
//FLEX program to implement the function of the special directive
//function YY_FLUSH_BUFFER
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//************************************************************************
%}
%%
[a-z]+  {
        printf("\nThe lower case token is (Using yytext) = %s",yytext);
        printf("\nThe length of the token = %d",yyleng);
        printf("\nThe lower case token is (Using Echo) =");ECHO;
        YY_FLUSH_BUFFER;
        printf("\nToken after yyflush (Using yytext) = %s",yytext);
        printf("\nThe length of the token after yyflush = %d",yyleng);
        printf("\nToken after yyflush (Using Echo) =");ECHO;
        }
%%
main()
{
 yylex();
}
---------------------------------- output ---------------------------------
[root@vinu yyFlush]# ./a.out
good
```

```
The lower case token is (Using yytext) = good
The length of the token = 4
The lower case token is (Using Echo) = good
Token after yyflush (Using yytext) =
The length of the token after yyflush = 4
Token after yyflush (Using Echo) = od

[1]+ Stopped ./a.out
[root@vinu yyFlush]#
```

Program 3.16 receives the input *good* and it matches with the first rule, the following action statements are executed. The action statements have displayed the *yytext* and ECHOed the token, after and before the execution of *yy_flush_buffer*. Note that after the execution of the *yy_flush_buffer*, *yytext* does not show anything when it displayed, but ECHO shows *od*. Also note when we tried to find the length of *yytext*, it is 4 instead of 0. Even though, theoretically, *yy_flush_buffer* flushes out the scanners internal buffer, practically it flushes out only the first two characters of the token (i.e. *yytext*) by making it to the NULL ('\0') character. That is why when we echoed after *yy_flush_buffer*, it showed the output as *od* (or *od*) and the internal buffer length was printed as 4. Generally, whenever asked to print the *yytext*, it will print character stream in the internal buffer starting from the first character array index to the place, where it finds "\0" (or NULL character). In this case, the scanner could find the NULL character at the first place of the *yytext*. It is for this reason that, it has not shown anything for *yytext* when it is printed using *printf*.

3.9 REDEFINING MACROS

As we have discussed earlier, the output of Flex is in the file *lex.yy.c*, which contains the scanning routine *yylex()*, and a number of auxiliary routines/functions and macros. By default, *yylex()* is declared as follows.

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

This definition may be changed by defining the YY_DECL macro, for example you could use

```
#define YY_DECL int yylex(int a, flex b)
```

Program 3.17 explains how we can change the datatype of the values that return from the *yylex()* and pass the parameters to the functions.

PROGRAM 3.17

```
---------------------------------------------------------------------------
%{
//*************************************************************************
//FLEX program to change the data type of yylex() by specifically defining it.
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************
#define YY_DECL int yylex(int flag)
%}
%%
[a-zA-Z]+[\n]{  /* First Rule */
                printf("\nHi....%s Good morning",yytext);
                printf("\nThe length of the name is = %d\n",(yyleng-1));
                flag++;
                return(flag);
             }
.+ {        /* Second Rule */
       int flag=1;
       return(flag);
    }
%%
main()
{
  int flag=-1;
  printf("\nEnter a word =");
  scanf("%s",yyin);
  flag=yylex(flag);
  if(flag==1)
    printf("\nThe given string is NOT a alphabetical word\n");
}
---------------------------------------------------------------------------
```

In the above program, *yylex()* has been redefined as *int yylex*(int flag) using YY_DECL. Whenever the *yylex()* is called, it passes an integer as arguments, and it scans tokens from the global input file *yyin* (by default, it is *stdin*—standard input), which has been scanned from the keyboard using the function scanf();. The function *yylex()* continues to run and scans the tokens from the input files which are pointed by the file pointer *yyin* (by default, it is defined in the flex tool) until it either reaches on end-of-file (at which point it returns the value 0) or one of its actions executes a return statement.

Program 3.17 matches the first rule, when we input any English alphabet and an enter key. During the execution of the following action statements of the first rule, the value of *yytext* and *yyleng* is displayed and the value of variable is incremented to '0' and returned to the caller, which is a main function. Whenever you input anything other than English alphabets, the variable *flag* is set to 1 and returned to the caller.

If the scanner reaches on end-of-file (EOF), subsequent calls are undefined unless either *yyin* is pointed at a new input file (in this case scanning continuous from that file to match the tokens) or *yyrestart()* is called.

Program 3.18 shows how the *yyrestart()* is implemented in Flex program. *yyrestart()* takes one argument, a FILE* pointer (which could be NULL, if you have set up YY_INPUT to scan from a source other than *yyin*) and initializes *yyin* for scanning from that file.

PROGRAM 3.18

```
-----------------------------------------------------------------------
%{
//*********************************************************************
//FLEX program that implements the yyrestart()
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*********************************************************************
%}
%%
[a-zA-Z]+ { printf("\nIts a lower case token =");
        ECHO;return;
}
%%
main()
{
  yylex();
  printf("\nENDING 1st yylex");
  FILE * fpt; /* Opening the input file using the function fopen */
  fpt=fopen("input.txt","r+");
  yyrestart(fpt);  /* Calling to restart the input file scanning */
  yylex();
  printf("\nENDING 2nd yylex");
}
-------------------------------- input.txt --------------------------------
Morning
```

```
---------------------------------- output ----------------------------------
[root@Zion restartYY]# ./a.out
Good

Its a lower case token = Good
ENDING 1st yylex
Its a lower case token = Morning
ENDING 2nd yylex
[root@Zion restartYY]#
----------------------------------------------------------------------------
```

Essentially there is no difference between just assigning a new input file to *yyin* and using *yyrestart()*, such as *yyin* = *fopen*("input.txt", "r+") directly available for compatibility with previous version of Flex and it can be used to switch input file in the middle of scanning.

By default (and for the purpose of the efficient Flex programming), the scanner uses block-reads rather than simple *getch()*; to read characters from *yyin*. The way it gets its input can be controlled by YY_INPUT macro. YY_INPUT's calling sequence is YY_INPUT(buf, result, max-size), its action is to place upto max-size character read or the constant YY_INPUT (0 on unix systems) to indicate EOF. The default YY_INPUT reads from the global file pointer *yyin*.

A sample definition of YY_INPUT is as follows. You may redefine it in the definition section of the flex input file.

```
%{
#define YY_INPUT (buf, result, max_size)
  {
    int c = getchar();
    result = (c == EOF) ? YY_NULL : (buf[0] = c, 1)
  }
%}
```

This definition will change the input processing to occur one character at a time. When the scanner receives an end-of-file file indication from YY_INPUT, it checks the *yywrap()* function. If *yywrap()* returns false (i.e. zero), then it is assumed that the function has gone ahead and set the *yyin* to point to another input file and scanning continues. If it (i.e. *yywrap()*) returns true (non-zero) then the scanner terminates, returning 0 to its caller. Note that in either case, the start condition remains unchanged; it does not revert to INITIAL.

The default *yywrap()* always returns 1. But we can provide our own version of *yywrap()* by specifying program files. If we do not supply our own version of *yywrap()*, then we must either use *%option noyywrap* (i.e. in this case the scanner behaves as though *yywrap()* returned 1) or we must link with *-lfl* to obtain the default version of the routine, which always returns 1.

Program 3.19 provides user defined version of the *yywrap()* that counts the number of links, words and characters of multiple files.

PROGRAM 3.19

```
----------------------------------------------------------------------
%{
//*********************************************************************
//FLEX program to count the line/word/char of multiple files
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*********************************************************************

/*declaring variables to calculate the statics of individual files*/
unsigned long charCount=0, wordCount=0, lineCount=0;

/*Undefining the default yywrap definition in the FLEX tool*/
#undef yywrap

%}

WORD [^ \t\n]+
EOL \n

%%

{WORD} { wordCount++;
     /*Adding the present token length to previous char count*/
     charCount=charCount+yyleng;
     }
{EOL}    { charCount++;
         lineCount++;
         }
. { charCount++;
         }
%%

char **fileList;
unsigned currentFile=0, noFiles;
/*declaring variables to calculate the statics of all files*/
unsigned long totalCC=0, totalWC=0, totalLC=0;

/*main() function takes two arguments; argv is a list of files names
and argc gives number of input (or file names) that we give to argv*/
main(int argc,char **argv)
{
  FILE *fpt;
  fileList = argv+1;
  noFiles=argc-1;
```

```
  /*we handle single file case differently from the multiple
  file case since we don't need to print a total summary details*/
  if(argc == 2)
  {
    currentFile=1;
    fpt=fopen(argv[1], "r");
    if(!fpt)
    {
       /*printing the automatically generated error
       message with our error message*/
       fprintf(stderr, "\nCould not open %s\n",argv[1]);
       exit(1);
    }
    yyin=fpt;
  }
  /*here user defined yywrap function is called only
  when there is more than one files as input arguments*/
  if(argc > 2)
    yywrap();
  yylex();
  /*printing the details of the last file*/
  printf("%8lu %8lu %8lu %s\n", lineCount, wordCount, charCount, fileList
[currentFile-1]);
  /*calculating the total statics of all file
  and displaying the same*/
  totalCC=totalCC+charCount;
  totalWC=totalWC+wordCount;
  totalLC=totalLC+lineCount;
  printf("%8lu %8lu %8lu\n", totalLC, totalWC, totalCC);
  return 0;
}
/* The lexer calls yywrap to check EOF condition*/
yywrap()
{
  FILE *fp=NULL;
  if((currentFile != 0) && (noFiles > 1) && (currentFile < noFiles))
  {
    /*printing the statics of previous file*/
    printf("%8lu %8lu %8lu %s\n", lineCount, wordCount, charCount, fileList
[currentFile-1]);
    /*Calculating the statics of the present file*/
    totalCC=totalCC+charCount;
    totalWC=totalWC+wordCount;
```

```
    totalLC=totalLC+lineCount;
    /*initialise the variable to calculate
    the statics of next file*/
    charCount=wordCount=lineCount=0;
    fclose(yyin);
  }
  while(fileList[currentFile] != NULL)
  {
    fp=fopen(fileList[currentFile++], "r");
    if(fp != NULL)
    {
      yyin=fp;
      break;
    }
    fprintf(stderr, "\nCould not open %s\n",fileList[currentFile-1]);
  }
  return(fp ? 0 : 1);
}
```

```
-------------------------------- input1.txt --------------------------------
Vinu V Das
Author of a book
Principles of Data Structure Using C and C++
New Age International
-------------------------------- input2.txt --------------------------------
Compiler Design
Using
FLEX & YACC
-------------------------------- input3.txt --------------------------------
This is a word count program
implemented using FLEC and YACC
with the help of undef yywrap()
-------------------------------- output --------------------------------
[root@localhost wrapYY]# ./a.out input1.txt input2.txt input3.txt
 4 18 95 input1.txt
 3 6 34 input2.txt
 3 17 93 input3.txt
 10 41 222
[root@localhost wrapYY]#
-----------------------------------------------------------------------
```

Our example reports both the size of the individual files and a cumulative total of the entire set of files at the end.

As we have discussed earlier, any macros can be redefined as C (such as *yywrap()*) function. In fact, you can even change the actual purpose of any macro. For example, Program 3.20 illustrates the Flex program that redefined the macro *yymore()* as C function to add two numbers.

PROGRAM 3.20

```
----------------------------------------------------------------------
%{
//*********************************************************************
//FLEX program that implements a user defined yymore by
//un-defining the already declared yymore()

//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*********************************************************************

#undef yymore /* undefinif the already declared yymore() */
int a,b,c;

%}

%%

add {/* First Rule that matches the pattern add */
printf("\nEnter any two numbers =");
   scanf("%d%d",&a,&b);yymore();
   printf("\nThe sum is = %d",c);
   }

.+ { /* Second rule */
     printf("\nEnter 'add' to add two numbers...");
}

%%

main()
{
  yylex();
}

/* User defined yymore() function */
yymore()
{
  c=a+b;
}
```

```
------------------------------------ output ------------------------------

[root@vinu yymoreTest]# ./a.out
12
```

```
Enter 'add' to add two numbers...
add

Enter the two munbers = 10
11

The output is = 21

[2]+ Stopped ./a.out
[root@vinu yymoreTest]#
```

In the above program, *yymore()* function is redefined to add two numbers, which will get into the program whenever we input "add". Then the user defined *yymore()* function is called to add two numbers, and to show the sum of input numbers.

PROGRAM 3.21

Program 3.21 checks whether the parenthesis in a statement is missing or not. The input text file *input.txt* contains the number of expressions. The Flex program will scan through the given file to check whether the expression has the correct number of right and left parenthesis or not.

```
%{
//*****************************************************************
//FLEX program checks whether the
//parenthesis in a statement is missing or not

//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*****************************************************************

int flag=0,ln=1;

%}

%%

"(" { flag++; /* First Rule that matches ( */
      }

")" { flag--; /* Second Rule that matches ) */
      }

[\n] { /* Third Rule - If the flag is equal to zero, the right and left
          parenthesis are of correct order, otherwise ERROR */
       if(flag == 0)
       printf("\n\nThe statement in the line %d has NO parenthesis missing\n",ln);
  else
```

```
   printf("\n\nERROR.. in the line : %d",ln);

   if(flag < 0)

   printf("\nIt has missed ( parenthesis or extra ) parenthesis\n");
   else if(flag > 0)
   printf("\nIt has missed ) parenthesis or extra ( parenthesis\n");

   flag = 0;ln++;
   }

.+ { /* Fourth rule to do nothing */
     }

%%

main()
{
   char fileName[20];
   printf("\nEnter the file name =");
   scanf("%s",fileName);
   yyin=fopen(fileName, "r+");
   yylex();
}
```

---------- input.txt ----------

```
((a+b*(b+c))
(a*(b+c))
((a/c)*b)+(b-(-c)))
(a+b)-(c*(d-e))
```

---------- output ----------

```
[root@Zion parenth]# ./a.out

Enter the file name = input.txt

ERROR.. in the line : 1
It has missed) parenthesis or extra (parenthesis

The statement in the line 2 has NO parenthesis missing

ERROR.. in the line : 3
It has missed (parenthesis or extra) parenthesis

The statement in the line 4 has NO parenthesis missing
[root@Zion parenth]#
```

A variable *flag* is set to 0, initial condition, and line number (ln) to one. As the scanner gets the token "(", the *flag* is incremented by expecting a closing parenthesis. At the end of every expression, that is, when the scanner gets the new line (i.e. \n), the *flag* must

be zero if it (expressions) has correct number of right and left parenthesis; otherwise it shows an error in the appropriate line. The output given above is the testimony of the same. Note that the fourth rule does nothing whenever the scanner gets a token that matches other than the above three rules patterns.

PROGRAM 3.22

Program 3.22 is a Flex program that simulates a simple desktop calculator to calculate the expressions with basic operators.

```
----------------------------------------------------------------------
%{
//*********************************************************************
//FLEX program that implements the simple desktop calculator

//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*********************************************************************

float op1=0,op2=0,ans=0;
char oper;
int f1=0,f2=0;

void eval();

%}

DIGIT [0-9]
NUM {DIGIT}+(\.{DIGIT}+)?
OP [+\-*/]

%%

{NUM} { /* First Rule that checks for digits */
    if(f1 == 0)
    {
       op1=atof(yytext);
       f1=1;
    }
    else if(f2 == -1)
    {
       op2=atof(yytext);
       f2=1;
    }

    if((f1==1) && (f2 == 1))
       eval();
  }
```

```
{OP} {      /* Second rule that checks for operators */
            oper=(char) *yytext;
            f2=-1;
        }

[\n]{       /* Third rule that checks for new line */
        if((f1 == 1) && (f2 == 1))
            eval();
        f1=0;f2=0;
        }

[\t] {      /* Fourth rule that checks against tab space and does nothing */
        }

. {     /* Fifth Rule */
    printf("\n");
  }

%%

main()
{
  yylex();
}

/* Function definition to evaluate the operations */
void eval()
{
  f1=0;f2=0;
  switch(oper)
  {
    case '+':ans=op1+op2;       /* Addition */
       break;
    case '-':ans=op1-op2;       /* Substation */
       break;
    case '*':ans=op1*op2;       /* Multiplication */
       break;
    case '/':if(op2==0)      /* Division */
       {
         printf("\nDivision by ZERO.....ERROR");
         return;
       }
       else
       {
         ans=op1/op2;
         break;
       }
```

```
    default:
       printf("\Program is NOT supporting the %c", oper);
       break;
  }
  printf("\nThe answer is = %lf",ans);
}
---------------------------------- output --------------------------------

[root@Zion calcuLex]# ./a.out
2+5

The answer is = 7.000000

5*2

The answer is = 10.000000

[1]+ Stopped ./a.out
[root@Zion calcuLex]#
-------------------------------------------------------------------------
```

The above program gets a simple expression as input with basic operators to evaluate the same. The first rule is the rule section of the Flex program that assigns the first operand to *op1* and the second to *op2*. The second rule assigns the operator to the variable *oper* to evaluate the expression by calling the function *eval()*. Note that the function *eval()* is called only after assigning the first operand to *op1* the operator to *oper*, and the second operator to *op2*. The *eval()* function does the appropriate calculations by checking the operator.

PROGRAM 3.23

Program 3.23 is a Flex program that implements the positive closure. The program will accept all strings (i.e. valid tokens) that accept the language (10)+.

```
-------------------------------------------------------------------------
%{
//*************************************************************************
//FLEX program to implement the positive closure.
// For eg. (10)+ That is 10 and its any combinations
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************
%}

LANGUAGE "10"

%%
```

```
{LANGUAGE}+ { /* First rule that matches all the combinations of 1's and 0's */
     printf("\nIts a positive closure....\n");
     return;
   }
.+{ /* Second rule is matched when the input is NOT 1's and 0's*/
          printf("\nSORRY.....It is NOT a positive closure\n");
          return;
     }
%%
main()
{
  yylex();
}
```

The first rule is matched against the input streams only when there is one or more numbers of '10' combinations. In any other case, the second rule is matched and the following action statements are executed.

EXERCISES

3.1 Flex program to simulate at least 7 operations in a desktop calculator.

3.2 Flex program that accepts the language

$$L = \{a^n b^n\}; \quad \text{where } n >= 0 \; m >= 1$$

(**Ans:** see Appendix A)

3.3 Flex program that accepts the language

$$L = \{1^{n-1} 0^n\}; \quad \text{where } n >= 1$$

(**Ans:** see Appendix A)

3.4 Flex program that accepts the language

$$L = \{1^{n-1} b^{n+m}\}; \quad \text{where } n >= 1 \; m >= 0$$

3.5 Flex program that identifies no of {, }, (and) in C program.