# LIST OF EXPERIMENTS

* 1. Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Simulate the same in C/C++ language.
  2. (a) Write a C program to identify whether a given line is a comment or not

(b) Write a C program to test whether a given identifier is valid or not.

* 1. Write a C program to validate operators.
  2. (a) Write a program to convert Regular Expression to Non-Deterministic Finite Automata.

(b) Write program to simulate DFA, NFA for accept any input belongs to a particular language.

* 1. To implement Lexical Analyzer using LEX or FLEX (Fast Lexical Analyzer). The program should separate the tokens in the given C program and display with appropriate caption.
  2. Implement following programs using LEX.
     1. Write a LEX specification file to take input C program from a .c file and count the number of characters, number of lines & number of words.
     2. Write a LEX program to count the number of Macros defined and header files included in the C program.
  3. Implement following programs using LEX.
     1. Write a LEX program to print all the constants in the given C source program file.
     2. Write a LEX program to print all HTML tags in the input file.
     3. Write a LEX program which adds line numbers to the given C program file and display the same in the standard output.
  4. Write a LEX program to count the number of comment lines in a given C program and eliminate them and write into another file.
  5. Implement a C program to perform symbol table operations.
  6. Implement a C program to eliminate left recursion and left factoring from a given CFG.
  7. Write a C program to find FIRST and FOLLOW for predictive parser.
  8. Write a C program for constructing of LL (1) parsing.
  9. Write a C program to construct recursive descent parsing.
  10. Write a C program for stack implementation of Shift Reduce parser.
  11. Write a C program to implement operator precedence parsing.
  12. Create YACC (or BISON) and LEX specification files to implement a basic calculator which accepts variables and constants of integer and float type.
  13. Implement a simple intermediate code generator in C program, which produces three address code statements for a given input expression.

**Experiment 1:** Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Simulate the same in C/C++ language.

**Aim:** Design a lexical analyzer for given language.

# Algorithm:

1. Start
2. Declare an array of characters, an input file to store the input;
3. Read the character from the input file and put it into character type of variable, say ‘c’.
4. If ‘c’ is blank then do nothing.
5. If ‘c’ is new line character line=line+1.
6. If ‘c’ is digit, set token Val, the value assigned for a digit and return the ‘NUMBER’.
7. If ‘c’ is proper token then assign the token value.
8. Print the complete table with token entered by the user and associated token value.
9. Stop

# Program:

#include<string.h> #include<ctype.h> #include<stdio.h>

void keyword(char str[10])

{

if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||strcmp("int",str

)==0||strcmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strcmp("st atic",str)==0||strcmp("switch",str)==0||strcmp("case",str)==0)

printf("\n%s is a keyword",str); else

printf("\n%s is an identifier",str);

}

main()

*{*

FILE \*f1,\*f2,\*f3; char c, str[10], st1[10];

int num[100], lineno=0, tokenvalue=0,i=0,j=0,k=0; printf("\n Enter the c program : ");/\*gets(st1);\*/ f1=fopen("input","w");

while((c=getchar())!=EOF) putc(c,f1);

fclose(f1); f1=fopen("input","r"); f2=fopen("identifier","w");

f3=fopen("specialchar","w");

while((c=getc(f1))!=EOF)

{

if(isdigit(c))

{

tokenvalue=c-'0'; c=getc(f1); while(isdigit(c))

{

tokenvalue\*=10+c-'0'; c=getc(f1);

}

num[i++]=tokenvalue; ungetc(c,f1);

}

else if(isalpha(c))

{

putc(c,f2); c=getc(f1);

while(isdigit(c)||isalpha(c)||c=='\_'||c=='$')

{

putc(c,f2); c=getc(f1);

}

*putc(' ',f2);*

ungetc(c,f1);

}

else

if(c==' '||c=='\t')

printf(" "); else if(c=='\n') lineno++; else putc(c,f3);

}

fclose(f2); fclose(f3); fclose(f1);

printf("\n The no's in the program are :"); for(j=0; j<i; j++)

printf("%d", num[j]); printf("\n"); f2=fopen("identifier", "r"); k=0;

printf("The keywords and identifiers are:"); while((c=getc(f2))!=EOF)

{

if(c!=' ') str[k++]=c; else

{

str[k]='\0'; keyword(str); k=0;

}

}

fclose(f2); f3=fopen("specialchar","r"); printf("\n Special characters are : ");

*while((c=getc(f3))!=EOF)*

printf("%c",c);

printf("\n"); fclose(f3);

printf("Total no. of lines are:%d", lineno);

}

# Sample Input & Output:

**Input:**

Enter Program Ctrl+D for termination:

{

int a[3],t1,t2;

t1=2; a[0]=1; a[1]=2; a[t1]=3;

t2=-(a[2]+t1\*6)/(a[2]-t1);

if t2>5 then print(t2); else { int t3; t3=99; t2=-25;

print(-t1+t2\*t3); /\* this is a comment on 2 lines \*/

} endif

}

# Output:

Variables : a[3] t1 t2 t3 Operator : - + \* / > Constants : 2 1 3 6 5 99 -25

Keywords : int if then else endif Special Symbols : , ; ( ) { }

Comments : this is a comment on 2 lines

**Experiment 2(a):** Write a C program to identify whether a given line is a comment or not

**Aim:** C program to identify whether a given line is a comment or not

# Program:

#include<stdio.h> #include<conio.h> void main()

{

char com[30]; int i=2,a=0; clrscr();

printf("\n Enter comment:");

gets(com); if(com[0]=='/') {

if(com[1]=='/')

printf("\n It is a comment"); else if(com[1]=='\*') {

for(i=2;i<=30;i++)

{

if(com[i]=='\*'&&com[i+1]=='/')

{

printf("\n It is a comment"); a=1; break; }

else continue; } if(a==0)

printf("\n It is not a comment");

}

else

else

printf("\n It is not a comment");

}

printf("\n It is not a comment"); getch();

}

**Sample Input & Output: Input:** Enter comment: //hello **Output**: It is a comment **Input:** Enter comment: hello **Output**: It is not a comment

**Experiment 2(b):** Write a C program to test whether a given identifier is valid or not.

**Aim:** Write a C program to test whether a given identifier is valid or not.

# Algorithm:

* 1. Start
  2. Read the given input string.
  3. Check the initial character of the string is numerical or any special character except ‘\_’ then print it is not a valid identifier.
  4. Otherwise print it as valid identifier if remaining characters of string doesn’t contains any special characters except ‘\_’.
  5. Stop.

# Program:

#include<stdio.h> #include<conio.h> #include<ctype.h>

void main()

{

char a[10]; int flag, i=1; clrscr();

printf("\n Enter an identifier:"); gets(a);

if(isalpha(a[0]))

flag=1; else

printf("\n Not a valid identifier");

while(a[i]!='\0')

{

if(!isdigit(a[i])&&!isalpha(a[i]))

{

flag=0; break;

} i++;

}

if(flag==1)

printf("\n Valid identifier"); getch();

}

**Sample Input & Output: Input**: Enter an identifier: first ***Output:***

Valid identifier

Enter an identifier:1aqw Not a valid identifier

**Experiment 3:** Write a C program to validate operators.

**Aim:** Write a C program to validate operators.

# Algorithm:

1. Start
2. Read the given input.
3. If the given input matches with any operator symbol.
4. Then display in terms of words of the particular symbol. Else print not an operator.
5. Stop

# Program:

#include<stdio.h> #include<conio.h> void main()

{

char s[5];

clrscr();

printf("\n Enter any operator:"); gets(s);

switch(s[0])

{

case'>': if(s[1]=='=')

printf("\n Greater than or equal"); else

printf("\n Greater than"); break;

case'<': if(s[1]=='=')

printf("\n Less than or equal"); else

printf("\nLess than"); break;

case'=': if(s[1]=='=') printf("\nEqual to");

else printf("\nAssignment"); break; case'!': if(s[1]=='=')

printf("\nNot Equal"); else

printf("\n Bit Not"); break;

case'&': if(s[1]=='&')

printf("\nLogical AND"); else

printf("\n Bitwise AND"); break;

case'|': if(s[1]=='|') printf("\nLogical OR"); else

printf("\nBitwise OR"); break;

case'+': printf("\n Addition"); break;

case'-': printf("\nSubstraction"); break;

case'\*': printf("\nMultiplication"); break;

case'/': printf("\nDivision");

break;

case'%': printf("Modulus"); break;

default: printf("\n Not a operator");

}

getch();

}

# Sample Input & Output:

**Input**

Enter any operator: \* **Output** Multiplication

**Experiment 4(a):** Write a program to convert Regular Expression to Non-Deterministic Finite Automata.

**Aim:** C Program to convert Regular Expression to Non-Deterministic Finite Automata.

# Algorithm:

* 1. Start
  2. Create a menu for getting four regular expressions input as choice.
  3. To draw NFA for a, a/b ,ab ,a\* create a routine for each regular expression.
  4. For converting from regular expression to NFA, certain transition had been made based on choice of input at the runtime.
  5. Each of the NFA will be displayed is sequential order.
  6. Stop

# Program:

#include<iostream> #include<conio.h> #include<process.h> using namespace std; struct node

{

char start; char alp; node \*nstate;

}\*p,\*p1,\*p2,\*p3,\*p4,\*p5,\*p6,\*p7,\*p8; char e='e';

void disp();

void re1()

{

p1=new(node); p2=new(node); p1->start='0'; p1->alp='e';

p1->nstate=p2; p2->start='1'; p2->alp='a';

p2->nstate=NULL; disp();

getch();

}

void re2()

{

p1=new(node); p2=new(node); p3=new(node); p4=new(node); p5=new(node); p6=new(node);

p7=new(node); p8=new(node); p1->start='0'; p1->alp='e';

p1->nstate=p2; p2->start='1'; p2->alp='a';

p2->nstate=p3; p3->start='2'; p3->alp='e';

p3->nstate=p4; p4->start='5'; p4->alp=' ';

p4->nstate=p5; p5->start='0'; p5->alp='e';

p5->nstate=p6; p6->start='3'; p6->alp='b';

p6->nstate=p7; p7->start='4'; p7->alp='e';

p7->nstate=p8; p8->start='5'; p8->alp=' ';

p8->nstate=NULL; disp();

getch();

}

void re3()

{

p1=new(node); p2=new(node); p3=new(node); p1->start='0'; p1->alp='a';

p1->nstate=p2; p2->start='1'; p2->alp='b';

p2->nstate=p3; p3->start='2'; p3->alp=' ';

p3->nstate=NULL; disp();

getch();

}

void re4()

{

p1=new(node); p2=new(node); p3=new(node); p4=new(node); p5=new(node); p6=new(node); p7=new(node); p8=new(node); p1->start='0'; p1->alp='e';

p1->nstate=p2; p2->start='1'; p2->alp='a';

p2->nstate=p3; p3->start='2'; p3->alp='e';

p3->nstate=p4; p4->start='3'; p4->alp=' ';

p4->nstate=p5; p5->start='0'; p5->alp='e';

p5->nstate=p6; p6->start='3'; p6->alp=' ';

p6->nstate=p7; p7->start='2'; p7->alp='e';

p7->nstate=p8; p8->start='1'; p8->alp=' ';

p8->nstate=NULL; disp();

getch();

}

void disp()

{

p=p1; while(p!=NULL)

{

cout<<"\t"<<p->start<<"\t"<<p->alp; p=p->nstate;

}

}

int main()

{

p=new(node); int ch=1;

while(ch!=0)

{

cout<<"\nMenu"<<"\n1.a"<<"\n2.a/b"<<"\n3.ab"<<"\n4.a\*"; cout<<"\n Enter the choice:";

cin>>ch; switch(ch)

{

case 1:

{

re1(); break;

}

case 2:

{

re2(); break;

}

case 3:

{

re3(); break;

}

case 4:

{

re4(); break;

}

default:

{

exit(0);

}

}

//clrscr();

}

}

# Sample Output:

Menu

* + 1. a
    2. a/b
    3. ab
    4. a\*

Enter the choice: 1 0 a 1

Menu

1. a
2. a/b
3. ab
4. a\*

Enter the choice: 2

0 e 1 a 2 e 5 0 e 3 b 4 e 5

Menu

1. a
2. a/b
3. ab
4. a\*

Enter the choice: 3 0 a 1 b 2

Menu

1. a
2. a/b
3. ab
4. a\*

Enter the choice: 4

0 e 1 a 2 e 3 0 e 3 2 e 1

Menu

1. a
2. a/b
3. ab
4. a\*

Enter the choice: 5

**Experiment 4(b):** Write program to simulate DFA for accept any input belongs to a particular language.

**Aim:** C program for simulating Deterministic Finite Automata.

# Algorithm:

1. Start
2. Consider the DFA represented in this transition diagram.

[](https://1.bp.blogspot.com/-gL6cLSX2w_0/VQBLMoDvMyI/AAAAAAAAAjo/a_O1LJ3UVMo/s1600/unnamed.png)

1. The sample DFA with starting state " 0 ", and accepting state " 1" . "a" and "b" are the input symbols.
2. The final DFA will accept any string containing 'a's and 'b' and last symbol should be 'a'.
3. Stop

# Program:

#include <stdio.h>

#define TOTAL\_STATES 2

#define FINAL\_STATES 1

#define ALPHABET\_CHARCTERS 2

#define UNKNOWN\_SYMBOL\_ERR 0

#define NOT\_REACHED\_FINAL\_STATE 1

#define REACHED\_FINAL\_STATE 2

enum DFA\_STATES{q0,q1}; enum input{a,b};

int Accepted\_states[FINAL\_STATES]={q1};

char alphabet[ALPHABET\_CHARCTERS]={'a','b'};

int Transition\_Table[TOTAL\_STATES][ALPHABET\_CHARCTERS]; int Current\_state=q0;

void DefineDFA()

{

Transition\_Table[q0][a] = q1; Transition\_Table[q0][b] = q0; Transition\_Table[q1][a] = q1; Transition\_Table[q1][b] = q0;

}

int DFA(char current\_symbol)

{

int i,pos;

for(pos=0;pos<ALPHABET\_CHARCTERS; pos++)

if(current\_symbol==alphabet[pos]) break;//stops if any character other than a or b

if(pos==ALPHABET\_CHARCTERS)

return UNKNOWN\_SYMBOL\_ERR; for(i=0;i<FINAL\_STATES;i++)

if((Current\_state=Transition\_Table[Current\_state][pos])

==Accepted\_states[i])

return REACHED\_FINAL\_STATE; return NOT\_REACHED\_FINAL\_STATE;

}

int main(void)

{

char current\_symbol; int result;

DefineDFA(); //Fill transition table

printf("Enter a string with 'a' s and 'b's:\nPress Enter Key to stop\n");

while((current\_symbol=getchar())!= '\n')

if((result= DFA(current\_symbol))==UNKNOWN\_SYMBOL\_ERR) break;

switch (result) {

case UNKNOWN\_SYMBOL\_ERR:printf("Unknown Symbol %c", current\_symbol);

break;

case NOT\_REACHED\_FINAL\_STATE:printf("Not accepted"); break; case REACHED\_FINAL\_STATE:printf("Accepted");break;

default: printf("Unknown Error");

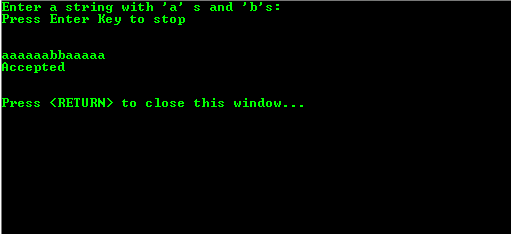
}

printf("\n\n\n"); return 0;

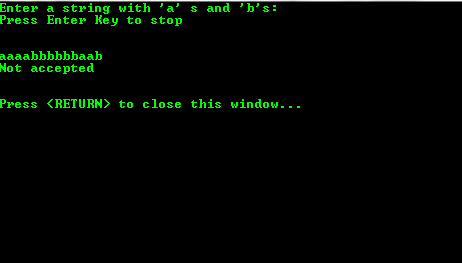
}

# Sample Output:

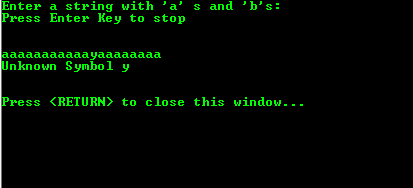
String with only "a"s and "b"s and last symbol is "a". ACCEPTED..!!



String with only "a"s and "b"s but last symbol is not "a". REJECTED..!!



Unknown symbol "y" in input. REJECTED.!!!!



**Experiment 5:** To implement Lexical Analyzer using LEX or FLEX (Fast Lexical Analyzer). The program should separate the tokens in the given C program and display with appropriate caption.

**Aim:** Design a Lexical Analyzer using LEX or FLEX to separate the tokens in the given C program.

# Algorithm:

1. Start
2. Open a file in text editor
3. Create a Lex specifications file to declare the variables, constants and inclusion of header files in the following format.
   1. %{

Definition of constant /header files

%}

* 1. Regular Expressions

%%

Transition rules

%%

* 1. Auxiliary Procedure (main( ) function)

1. Save file with .l extension e.g. token.l
2. Open Command prompt and switch to your working directory where you have stored your lex file (“.l“).
3. Compile the lex file named “token.l” using the command “flex token.l”. It will produce a ‘C’

version of lexical analyzer which is stored in “lex.yy.c”.

1. Compile the file lex.yy.c using C / C++ compiler. e.g. gcc lex.yy.c. After compilation the file lex.yy.c, the output file is in a.exe.
2. Run the file a.exe giving an input (text/file) e.g. a.exe sample.c
3. Upon processing, the sequence of tokens will be displayed as output.
4. Stop

# Lex Program: (token.l)

digit [0-9] letter [A-Za-z]

%{

int count\_id,count\_key;

%}

%%

(stdio.h|conio.h) { printf("%s is a standard library\n",yytext); } (include|void|main|printf|int) { printf("%s is a keyword\n",yytext); count\_key++; }

{letter}({letter}|{digit})\* { printf("%s is a identifier\n", yytext); count\_id++; }

{digit}+ { printf("%s is a number\n", yytext); }

\"(\\.|[^"\\])\*\" { printf("%s is a string literal\n", yytext); }

.|\n { }

%%

int yywrap(void) { return 1;

}

int main(int argc, char \*argv[]) {

yyin = fopen(argv[1], "r"); yylex();

printf("number of identifiers = %d\n", count\_id); printf("number of keywords = %d\n", count\_key); fclose(yyin);

}

# Input Source Program: (sample.c)

#include<stdio.h> void main()

{

int a,b,c = 30; printf("hello");

}

**Compilation & Execution of Lex Program:**

1. Open Command prompt and switch to your working directory where you have stored your lex file (“.l“).
2. Let lex file be “token.l”. Now, follow the preceding steps to compile and run lex program.
   1. For Compiling **Lex** file:
      1. flex token.l
      2. gcc lex.yy.c
   2. For **Executing** the Program a.exe sample.c

# Sample Output:

G:\lex>flex token.l

G:\lex>gcc lex.yy.c

G:\lex>a.exe sample.c include is a keyword stdio.h is a standard library void is a keyword

main is a keyword int is a keyword

a is a identifier b is a identifier c is a identifier 30 is a number

printf is a keyword "hello" is a string literal number of identifiers = 3 number of keywords = 5

G:\lex>

**Experiment 6(a):** Write a LEX specification file to take input C program from a .c file and count the number of characters, number of lines & number of words.

**Aim:** Design a Lexical Analyzer using LEX or FLEX to count the number of characters, number of lines & number of words.

# Algorithm:

1. Start
2. Open a file in text editor
3. Create a Lex specifications file and save file with .l extension.
4. Open Command prompt and switch to your working directory where you have stored your lex file.
5. Compile the lex file named “xxxxx.l” using the command “flex xxxxx.l”. It will produce a ‘C’

version of lexical analyzer which is stored in “lex.yy.c”.

1. Compile the file lex.yy.c using C / C++ compiler. e.g. gcc lex.yy.c. After compilation the file lex.yy.c, the output file is in a.exe.
2. Run the file a.exe giving an input (text/file) e.g. a.exe sample.c
3. Upon processing, the sequence of tokens will be displayed as output.
4. Stop

# Lex Program: (count\_lines.l)

%{

int nchar, nword, nline;

%}

%%

\n { nline++; nchar++; }

[^ \t\n]+ { nword++, nchar += yyleng; }

. { nchar++; }

%%

int yywrap(void) { return 1;

}

int main(int argc, char \*argv[]) { yyin = fopen(argv[1], "r"); yylex();

printf("Number of characters = %d\n", nchar); printf("Number of words = %d\n", nword); printf("Number of lines = %d\n", nline); fclose(yyin);

}

# Input Source Program: (sample.c)

#include<stdio.h> void main()

{

int a,b,c = 30;

printf("hello");

}

**Compilation & Execution of Lex Program:**

1. Open Command prompt and switch to your working directory where you have stored your lex file (“.l“).
2. Let lex file be “count\_lines.l”. Now, follow the preceding steps to compile and run lex program.
   1. For Compiling **Lex** file:
      1. flex count\_lines.l
      2. gcc lex.yy.c
   2. For **Executing** the Program a.exe sample.c

# Sample Output:

G:\lex>flex count\_line.l G:\lex>gcc lex.yy.c

G:\lex>a.exe sample.c Number of characters = 71 Number of words = 10 Number of lines = 6

G:\lex>

**Experiment 6(b):** Write a LEX program to count the number of Macros defined and header files included in the C program.

**Aim:** Design a Lexical Analyzer using LEX or FLEX to count the number of Macros defined and header files.

# Algorithm:

1. Start
2. Open a file in text editor
3. Create a Lex specifications file and save file with .l extension.
4. Open Command prompt and switch to your working directory where you have stored your lex file.
5. Compile the lex file named “xxxxx.l” using the command “flex xxxxx.l”. It will produce a ‘C’

version of lexical analyzer which is stored in “lex.yy.c”.

1. Compile the file lex.yy.c using C / C++ compiler. e.g. gcc lex.yy.c. After compilation the file lex.yy.c, the output file is in a.exe.
2. Run the file a.exe giving an input (text/file) e.g. a.exe sample.c
3. Upon processing, the sequence of tokens will be displayed as output.
4. Stop

# Lex Program: (count\_macro.l)

%{

int nmacro, nheader;

%}

%%

^#define { nmacro++; }

^#include { nheader++; }

.|\n { }

%%

int yywrap(void) { return 1;

}

int main(int argc, char \*argv[]) { yyin = fopen(argv[1], "r"); yylex();

printf("Number of macros defined = %d\n", nmacro); printf("Number of header files included = %d\n", nheader); fclose(yyin);

}

# Input Source Program: (sample.c)

#define PI 3.14 #include<stdio.h> #include<conio.h> void main()

{

int a,b,c = 30;

printf("hello");

}

# Sample Output:

G:\lex>flex count\_macro.l G:\lex>gcc lex.yy.c

G:\lex>a.exe sample.c Number of macros defined = 1

Number of header files included = 2 G:\lex>

**Experiment 7(a):** Write a LEX program to print all the constants in the given C source program file.

**Aim:** Design a Lexical Analyzer using LEX or FLEX to print all the constants.

# Algorithm:

1. Start
2. Open a file in text editor
3. Create a Lex specifications file and save file with .l extension.
4. Open Command prompt and switch to your working directory where you have stored your lex file.
5. Compile the lex file named “xxxxx.l” using the command “flex xxxxx.l”. It will produce a ‘C’

version of lexical analyzer which is stored in “lex.yy.c”.

1. Compile the file lex.yy.c using C / C++ compiler. e.g. gcc lex.yy.c. After compilation the file lex.yy.c, the output file is in a.exe.
2. Run the file a.exe giving an input (text/file) e.g. a.exe sample.c
3. Upon processing, the sequence of tokens will be displayed as output.
4. Stop

# Lex Program: (7a\_countconstants.l)

digit [0-9]

%{

int cons=0;

%}

%%

{digit}+ { cons++; printf("%s is a constant\n", yytext); }

.|\n { }

%%

int yywrap(void) { return 1; }

int main(void)

{

FILE \*f;

char file[10];

printf("Enter File Name : "); scanf("%s",file);

f = fopen(file,"r"); yyin = f;

yylex();

printf("Number of Constants : %d\n", cons); fclose(yyin);

}

# Input Source Program: (sample.c)

#include<stdio.h> void main()

{

int a,b = 20, c = 30;

printf("hello");

}

# Sample Output:

G:\lex>flex 7a\_countconstants.l G:\lex>gcc lex.yy.c

G:\lex>a.exe

Enter File Name : sample.c 20 is a constant

30 is a constant

Number of Constants : 2 G:\lex>

**Experiment 7(b):** Write a LEX program to print all HTML tags in the input file.

**Aim:** Design a Lexical Analyzer using LEX or FLEX to print all HTML tags in the input file.

# Algorithm:

1. Start
2. Open a file in text editor
3. Create a Lex specifications file and save file with .l extension.
4. Open Command prompt and switch to your working directory where you have stored your lex file.
5. Compile the lex file named “xxxxx.l” using the command “flex xxxxx.l”. It will produce a ‘C’

version of lexical analyzer which is stored in “lex.yy.c”.

1. Compile the file lex.yy.c using C / C++ compiler. e.g. gcc lex.yy.c. After compilation the file lex.yy.c, the output file is in a.exe.
2. Run the file a.exe giving an input (text/file) e.g. a.exe sample.c
3. Upon processing, the sequence of tokens will be displayed as output.
4. Stop

# Lex Program: (7b\_html.l)

%{

int tags;

%}

%%

"<"[^>]\*> { tags++; printf("%s \n", yytext); }

.|\n { }

%%

int yywrap(void) { return 1; }

int main(void)

{

FILE \*f;

char file[10];

printf("Enter File Name : "); scanf("%s",file);

f = fopen(file,"r"); yyin = f;

yylex();

printf("\n Number of html tags: %d",tags); fclose(yyin);

}

# Input Source Program: (sample.html)

<html>

<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>

</html>

# Sample Output:

G:\lex>flex 7b\_html.l G:\lex>gcc lex.yy.c

G:\lex>a.exe

Enter File Name : sample.html

<html>

<body>

<h1>

</h1>

<p>

</p>

</body>

</html>

Number of html tags: 8 G:\lex>

**Experiment 7(c):** Write a LEX program which adds line numbers to the given C program file and display the same in the standard output.

**Aim:** Design a Lexical Analyzer using LEX or FLEX to add line numbers to the given C program file.

# Algorithm:

1. Start
2. Open a file in text editor
3. Create a Lex specifications file and save file with .l extension.
4. Open Command prompt and switch to your working directory where you have stored your lex file.
5. Compile the lex file named “xxxxx.l” using the command “flex xxxxx.l”. It will produce a ‘C’

version of lexical analyzer which is stored in “lex.yy.c”.

1. Compile the file lex.yy.c using C / C++ compiler. e.g. gcc lex.yy.c. After compilation the file lex.yy.c, the output file is in a.exe.
2. Run the file a.exe giving an input (text/file) e.g. a.exe sample.c
3. Upon processing, the sequence of tokens will be displayed as output.
4. Stop

# Lex Program: (7c\_addlinenos.l)

%{

int yylineno;

%}

%%

^(.\*)\n printf("%4d\t%s", ++yylineno, yytext);

%%

int yywrap(void) { return 1;

}

int main(int argc, char \*argv[]) { yyin = fopen(argv[1], "r"); yylex();

fclose(yyin);

}

# Input Source Program: (sample.c)

#define PI 3.14 #include<stdio.h> #include<conio.h> void main()

{

int a,b,c = 30; printf("hello");

}

# Sample Output:

G:\lex>flex 7c\_addlinenos.l G:\lex>gcc lex.yy.c

G:\lex>a.exe sample.c

1. #define PI 3.14
2. #include<stdio.h>
3. #include<conio.h>
4. void main() 5 {
5. int a,b,c = 30;
6. printf("hello"); 8 }

G:\lex>

**Experiment 8:** Write a LEX program to count the number of comment lines in a given C program and eliminate them and write into another file.

**Aim:** Design a Lexical Analyzer using LEX or FLEX to count the number of comment lines in a given C program and eliminate them and write into another file.

# Algorithm:

* 1. Start
  2. Open a file in text editor
  3. Create a Lex specifications file and save file with .l extension.
  4. Open Command prompt and switch to your working directory where you have stored your lex file.
  5. Compile the lex file named “xxxxx.l” using the command “flex xxxxx.l”. It will produce a ‘C’

version of lexical analyzer which is stored in “lex.yy.c”.

* 1. Compile the file lex.yy.c using C / C++ compiler. e.g. gcc lex.yy.c. After compilation the file lex.yy.c, the output file is in a.exe.
  2. Run the file a.exe giving an input (text/file) e.g. a.exe sample.c
  3. Upon processing, the sequence of tokens will be displayed as output.
  4. Stop

# Lex Program: (comment.l)

%{

int com=0;

%}

%s COMMENT

%%

"/\*" {BEGIN COMMENT;}

<COMMENT>"\*/" {BEGIN 0; com++;}

<COMMENT>\n {com++;}

<COMMENT>. {;}

\/\/.\* {; com++;}

.|\n {fprintf(yyout,"%s",yytext);}

%%

void main(int argc, char \*argv[])

{

if(argc!=3)

{

printf("usage : ./a.exe input.c output.c\n"); exit(0);

}

yyin=fopen(argv[1],"r");

yyout=fopen(argv[2],"w"); yylex();

printf("\n number of comments are = %d\n",com);

}

int yywrap()

{

return 1;

}

# Input Source File: (input.c)

#include<stdio.h> int main()

{

int a,b,c; /\*varible declaration\*/ printf(“enter two numbers”); scanf(“%d %d”,&a,&b); c=a+b;//adding two numbers printf(“sum is %d”,c);

return 0;

}

# Output Destination File: (output.c)

include<stdio.h> int main()

{

int a,b,c;

printf(“enter two numbers”); scanf(“%d %d”,&a,&b); c=a+b;

printf(“sum is %d”,c); return 0;

}

# Sample Output:

G:\lex>flex comment.l G:\lex>gcc lex.yy.c G:\lex>a.exe input.c output.c number of comments are = 2 G:\lex>

**Experiment 9:** Implement a C program to perform symbol table operations.

**Aim:** Implement a C program to perform symbol table operations.

# Algorithm:

1. Start the program for performing insert, display, delete, search and modify option in symbol table
2. Define the structure of the Symbol Table
3. Enter the choice for performing the operations in the symbol Table
4. If the entered choice is 1, search the symbol table for the symbol to be inserted. If the symbol is already present, it displays “Duplicate Symbol”. Else, insert the symbol and the corresponding address in the symbol table.
5. If the entered choice is 2, the symbols present in the symbol table are displayed.
6. If the entered choice is 3, the symbol to be deleted is searched in the symbol table.
7. If it is not found in the symbol table it displays “Label Not found”. Else, the symbol is deleted.
8. Stop

# Program:

#include<stdio.h> #include<conio.h> #include<malloc.h> #include<string.h> #include<stdlib.h> #define NULL 0

int size=0; void Insert(); void Display(); void Delete();

int Search(char lab[]); void Modify();

struct SymbTab

{

char label[10],symbol[10]; int addr;

struct SymbTab \*next;}; struct SymbTab \*first,\*last; int main()

{

int op,y; char la[10];

//clrscr(); do

{

printf("\n\tSYMBOL TABLE IMPLEMENTATION\n");

printf("\n\t1.INSERT\n\t2.DISPLAY\n\t3.DELETE\n\t4.SEARCH\n\t5.MODIFY\n\t6.END\n"); printf("\n\tEnter your option : ");

scanf("%d",&op); switch(op)

{

case 1:

Insert(); break;

case 2:

Display(); break;

case 3:

Delete(); break;

case 4:

printf("\n\tEnter the label to be searched : "); scanf("%s",la);

y=Search(la); printf("\n\tSearch Result:"); if(y==1)

printf("\n\tThe label is present in the symbol table\n"); else

printf("\n\tThe label is not present in the symbol table\n"); break;

case 5: Modify(); break;

case 6: exit(0);

}

}while(op<6);

getch();

}

void Insert()

{

int n;

char l[10];

printf("\n\tEnter the label : "); scanf("%s",l);

n=Search(l); if(n==1)

printf("\n\tThe label exists already in the symbol table\n\tDuplicate can't be inserted"); else

{

struct SymbTab \*p; p=malloc(sizeof(struct SymbTab)); strcpy(p->label,l);

printf("\n\tEnter the symbol : "); scanf("%s",p->symbol); printf("\n\tEnter the address : ");

scanf("%d",&p->addr); p->next=NULL; if(size==0)

{

first=p; last=p;

}

else

{

last->next=p; last=p;

}

size++;

}

printf("\n\tLabel inserted\n");

}

void Display()

{

int i;

struct SymbTab \*p; p=first;

printf("\n\tLABEL\t\tSYMBOL\t\tADDRESS\n"); for(i=0;i<size;i++)

{

printf("\t%s\t\t%s\t\t%d\n",p->label,p->symbol,p->addr); p=p->next;

}

}

int Search(char lab[])

{

int i,flag=0;

struct SymbTab \*p; p=first; for(i=0;i<size;i++)

{

if(strcmp(p->label,lab)==0) flag=1;

p=p->next;

}

return flag;

}

void Modify()

{

char l[10],nl[10]; int add,choice,i,s; struct SymbTab \*p; p=first;

printf("\n\tWhat do you want to modify?\n");

printf("\n\t1.Only the label\n\t2.Only the address\n\t3.Both the label and address\n"); printf("\tEnter your choice : ");

scanf("%d",&choice); switch(choice)

{

case 1:

printf("\n\tEnter the old label : "); scanf("%s",l);

s=Search(l); if(s==0)

printf("\n\tLabel not found\n"); else

{

printf("\n\tEnter the new label : "); scanf("%s",nl); for(i=0;i<size;i++)

{

if(strcmp(p->label,l)==0) strcpy(p->label,nl);

p=p->next;

}

printf("\n\tAfter Modification:\n"); Display();

}

break; case 2:

printf("\n\tEnter the label where the address is to be modified : "); scanf("%s",l);

s=Search(l); if(s==0)

printf("\n\tLabel not found\n"); else

{

printf("\n\tEnter the new address : "); scanf("%d",&add); for(i=0;i<size;i++)

{

if(strcmp(p->label,l)==0) p->addr=add;

p=p->next;

}

printf("\n\tAfter Modification:\n"); Display();

}

break; case 3:

printf("\n\tEnter the old label : "); scanf("%s",l);

s=Search(l); if(s==0)

printf("\n\tLabel not found\n"); else

{

printf("\n\tEnter the new label : "); scanf("%s",nl);

printf("\n\tEnter the new address : "); scanf("%d",&add); for(i=0;i<size;i++)

{

if(strcmp(p->label,l)==0)

{

strcpy(p->label,nl); p->addr=add;

}

p=p->next;

}

printf("\n\tAfter Modification:\n"); Display();

}

break;

}

}

void Delete()

{

int a;

char l[10];

struct SymbTab \*p,\*q; p=first;

printf("\n\tEnter the label to be deleted : "); scanf("%s",l);

a=Search(l); if(a==0)

printf("\n\tLabel not found\n"); else

{

if(strcmp(first->label,l)==0) first=first->next;

else if(strcmp(last->label,l)==0)

{

q=p->next;

while(strcmp(q->label,l)!=0)

{

p=p->next; q=q->next;

}

p->next=NULL;

last=p;

}

else

{

q=p->next;

while(strcmp(q->label,l)!=0)

{

p=p->next; q=q->next;

}

p->next=q->next;

}

size--;

printf("\n\tAfter Deletion:\n"); Display();

}

}

# Sample Output:

SYMBOL TABLE IMPLEMENTATION

* 1. INSERT
  2. DISPLAY
  3. DELETE
  4. SEARCH
  5. MODIFY
  6. END

Enter your option : 1 Enter the label : plus Enter the symbol : + Enter the address : 100 Label inserted

SYMBOL TABLE IMPLEMENTATION

1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END

Enter your option : 2

LABEL SYMBOL ADDRESS

plus + 100

SYMBOL TABLE IMPLEMENTATION 1.INSERT

1. DISPLAY
2. DELETE
3. SEARCH
4. MODIFY
5. END

Enter your option : 3

Enter the label to be deleted : plus After Deletion:

LABEL SYMBOL ADDRESS SYMBOL TABLE IMPLEMENTATION

1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END

Enter your option : 2

LABEL SYMBOL ADDRESS SYMBOL TABLE IMPLEMENTATION

1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END

Enter your option : 1 Enter the label : minus

Enter the symbol : - Enter the address : 200 Label inserted

SYMBOL TABLE IMPLEMENTATION

1. INSERT
2. DISPLAY
3. DELETE
4. SEARCH
5. MODIFY
6. END

Enter your option : 4

Enter the label to be searched : minus Search Result:

The label is present in the symbol table

SYMBOL TABLE IMPLEMENTATION 1.INSERT

1. DISPLAY
2. DELETE
3. SEARCH
4. MODIFY
5. END

Enter your option : 5

What do you want to modify? 1.Only the label

1. Only the address
2. Both the label and address Enter your choice : 1

Enter the old label : minus

Enter the new label : minus\_new

After Modification:

LABEL SYMBOL ADDRESS

minus\_new - 200

SYMBOL TABLE IMPLEMENTATION 1.INSERT

1. DISPLAY
2. DELETE
3. SEARCH
4. MODIFY
5. END

Enter your option : 6

**Experiment 10(a):** Implement a C program to eliminate left recursion from a given CFG.

**Aim:** Implement a C program to eliminate left recursion from a given CFG.

# Algorithm:

1. Start the program.
2. Define the structure of the CFG.
3. Left recursion is eliminated by converting the grammar into a right recursive grammar.
4. If we have the left-recursive pair of productions- **A** → **Aα / β** (Left Recursive Grammar) where β does not begin with an A.
5. Then, we can eliminate left recursion by replacing the pair of productions with-

**A** → **βA’**

**A’** → **αA’ / ∈** (Right Recursive Grammar)

1. This right recursive grammar functions same as left recursive grammar.
2. Stop

# Program:

#include<stdio.h> #include<string.h> #define SIZE 10 int main () {

char non\_terminal; char beta,alpha; int num;

char production[10][SIZE];

int index=3; /\* starting of the string following "->" \*/ printf("Enter Number of Production : "); scanf("%d",&num);

printf("Enter the grammar as E->E-A :\n"); for(int i=0;i<num;i++){

scanf("%s",production[i]);

}

for(int i=0;i<num;i++){

printf("\nGRAMMAR : : : %s",production[i]); non\_terminal=production[i][0]; if(non\_terminal==production[i][index]) {

alpha=production[i][index+1]; printf(" is left recursive.\n");

while(production[i][index]!=0 && production[i][index]!='|') index++;

if(production[i][index]!=0) { beta=production[i][index+1]; printf("Grammar without left recursion:\n");

printf("%c->%c%c\'",non\_terminal,beta,non\_terminal); printf("\n%c\'->%c%c\'|E\n",non\_terminal,alpha,non\_terminal);

}

else

printf(" can't be reduced\n");

}

else

printf(" is not left recursive.\n"); index=3;

}

}

# Sample Output:

Enter Number of Production : 4 Enter the grammar as E->E-A : E->EA|A

A->AT|a T->a

E->i

GRAMMAR : : : E->EA|A is left recursive. Grammar without left recursion:

E->AE' E'->AE'|E

GRAMMAR : : : A->AT|a is left recursive. Grammar without left recursion:

A->aA' A'->TA'|E

GRAMMAR : : : T->a is not left recursive. GRAMMAR : : : E->i is not left recursive.

**Experiment 10(b):** Implement a C program to eliminate left factoring from a given CFG.

**Aim:** Implement a C program to eliminate left factoring from a given CFG.

# Algorithm:

1. Start the program.
2. Define the structure of the CFG.
3. Make one production for each common prefixes.
4. The common prefix may be a terminal or a non-terminal or a combination of both.
5. Rest of the derivation is added by new productions.
6. Stop

**Program:** #include<stdio.h> #include<string.h>

int main()

{

char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20]; int i,j=0,k=0,l=0,pos;

printf("Enter Production : A->"); gets(gram); for(i=0;gram[i]!='|';i++,j++)

part1[j]=gram[i]; part1[j]='\0';

for(j=++i,i=0;gram[j]!='\0';j++,i++) part2[i]=gram[j];

part2[i]='\0'; for(i=0;i<strlen(part1)||i<strlen(part2);i++)

{

if(part1[i]==part2[i])

{

modifiedGram[k]=part1[i]; k++;

pos=i+1;

}

}

for(i=pos,j=0;part1[i]!='\0';i++,j++){ newGram[j]=part1[i];

}

newGram[j++]='|'; for(i=pos;part2[i]!='\0';i++,j++){

newGram[j]=part2[i];

}

modifiedGram[k]='X'; modifiedGram[++k]='\0';

newGram[j]='\0';

printf("\n A->%s",modifiedGram); printf("\n X->%s\n",newGram);

}

# Sample Output:

Enter Production : A->abCDE|abCFG

A->abCX X->DE|FG

**Experiment 11(a):** Write a C program to find FIRST set for predictive parser.

**Aim:** Implement a C program to find FIRST set for predictive parser.

# Algorithm:

* 1. Start
  2. First(α) is a set of terminal symbols that begin in strings derived from α
  3. Rules for Calculating First Function

Rule-01: For a production rule X → ∈, First(X) = { ∈ } Rule-02: For any terminal symbol ‘a’, First(a) = { a } Rule-03: For a production rule X → Y1Y2Y3.

* + 1. For calculating First(X), If ∈ ∉ First(Y1), then First(X) = First(Y1) If ∈ ∈ First(Y1), then First(X) = { First(Y1) – ∈ } ∪ First(Y2Y3).
    2. Calculating First(Y2Y3) If ∈ ∉ First(Y2), then First(Y2Y3) = First(Y2) If ∈ ∈ First(Y2), then First(Y2Y3) = { First(Y2) – ∈ } ∪ First(Y3) Similarly, we can make expansion for any production rule X → Y1Y2Y3…..Yn.
  1. Stop

**Program:** #include<stdio.h> #include<ctype.h>

void FIRST(char[],char );

void addToResultSet(char[],char); int numOfProductions;

char productionSet[10][10]; int main()

{

int i;

char choice; char c;

char result[20];

printf("How many number of productions ? :"); scanf(" %d",&numOfProductions);

for(i=0;i<numOfProductions;i++)//read production string eg: E=E+T

{

printf("Enter productions Number %d : ",i+1); scanf(" %s",productionSet[i]);

}

do

{

printf("\n Find the FIRST of :"); scanf(" %c",&c);

FIRST(result,c); //Compute FIRST; Get Answer in 'result' array printf("\n FIRST(%c)= { ",c);

for(i=0;result[i]!='\0';i++)

printf(" %c ",result[i]); //Display result printf("}\n");

printf("press 'y' to continue : "); scanf(" %c",&choice);

}

while(choice=='y'||choice =='Y');

}

/\*

\*Function FIRST:

\*Compute the elements in FIRST(c) and write them

\*in Result Array.

\*/

void FIRST(char\* Result,char c)

{

int i,j,k;

char subResult[20]; int foundEpsilon; subResult[0]='\0'; Result[0]='\0';

//If X is terminal, FIRST(X) = {X}. if(!(isupper(c)))

{

addToResultSet(Result,c); return ;

}

//If X is non terminal

//Read each production for(i=0;i<numOfProductions;i++)

{

//Find production with X as LHS if(productionSet[i][0]==c)

{

//If X → ε is a production, then add ε to FIRST(X). if(productionSet[i][2]=='$') addToResultSet(Result,'$');

//If X is a non-terminal, and X → Y1 Y2 … Yk

//is a production, then add a to FIRST(X)

//if for some i, a is in FIRST(Yi),

//and ε is in all of FIRST(Y1), …, FIRST(Yi-1). else

{

j=2;

while(productionSet[i][j]!='\0')

{

foundEpsilon=0; FIRST(subResult,productionSet[i][j]); for(k=0;subResult[k]!='\0';k++)

addToResultSet(Result,subResult[k]); for(k=0;subResult[k]!='\0';k++)

if(subResult[k]=='$')

{

foundEpsilon=1; break;

}

//No ε found, no need to check next element if(!foundEpsilon)

break; j++;

}

}

}

}

return ;

}

/\* addToResultSet adds the computed

\*element to result set.

\*This code avoids multiple inclusion of elements

\*/

void addToResultSet(char Result[],char val)

{

int k;

for(k=0 ;Result[k]!='\0';k++) if(Result[k]==val)

return;

Result[k]=val;

Result[k+1]='\0';

}

# Sample Output:

How many number of productions ? :8 Enter productions Number 1 : E=TD Enter productions Number 2 : D=+TD Enter productions Number 3 : D=$ Enter productions Number 4 : T=FS Enter productions Number 5 : S=\*FS Enter productions Number 6 : S=$ Enter productions Number 7 : F=(E) Enter productions Number 8 : F=a

Find the FIRST of :E FIRST(E)= { ( a }

press 'y' to continue : y

Find the FIRST of :D FIRST(D)= { + $ }

press 'y' to continue : y

Find the FIRST of :T FIRST(T)= { ( a }

press 'y' to continue : n

**Experiment 11(b):** Write a C program to find FOLLOW set for predictive parser.

**Aim:** Implement a C program to find FOLLOW set for predictive parser.

# Algorithm:

1. Start the program
2. Follow(α) is a set of terminal symbols that appear immediately to the right of α. Rules. For Calculating Follow Function
3. Rule-01: For the start symbol S, place $ in Follow(S).
4. Rule-02: For any production rule A → αB, Follow(B) = Follow(A)
5. Rule-03: For any production rule A → αBβ, If ∈ ∉ First(β), then Follow(B) = First(β). If ∈ ∈

First(β), then Follow(B) = { First(β) – ∈ } ∪ Follow(A).

1. Stop the program

**Program:** #include<stdio.h> #include<ctype.h> #include<string.h> int limit, x = 0;

char production[10][10], array[10];

void find\_first(char ch); void find\_follow(char ch);

void Array\_Manipulation(char ch);

int main()

{

int count;

char option, ch;

printf("\nEnter Total Number of Productions:\t"); scanf("%d", &limit);

for(count = 0; count < limit; count++)

{

printf("\nValue of Production Number [%d]:\t", count + 1); scanf("%s", production[count]);

}

do

{

x = 0;

printf("\nEnter production Value to Find Follow:\t"); scanf(" %c", &ch);

find\_follow(ch);

printf("\nFollow Value of %c:\t{ ", ch); for(count = 0; count < x; count++)

{

printf("%c ", array[count]);

}

printf("}\n");

printf("To Continue, Press Y:\t"); scanf(" %c", &option);

}while(option == 'y' || option == 'Y'); return 0;

}

void find\_follow(char ch)

{

int i, j;

int length = strlen(production[i]); if(production[0][0] == ch)

{

Array\_Manipulation('$');

}

for(i = 0; i < limit; i++)

{

for(j = 2; j < length; j++)

{

if(production[i][j] == ch)

{

if(production[i][j + 1] != '\0')

{

find\_first(production[i][j + 1]);

}

if(production[i][j + 1] == '\0' && ch != production[i][0])

{

find\_follow(production[i][0]);

}

}

}

}

}

void find\_first(char ch)

{

int i, k; if(!(isupper(ch)))

{

Array\_Manipulation(ch);

}

for(k = 0; k < limit; k++)

{

if(production[k][0] == ch)

{

if(production[k][2] == '$')

{

find\_follow(production[i][0]);

}

else if(islower(production[k][2]))

{

Array\_Manipulation(production[k][2]);

}

else

{

find\_first(production[k][2]);

}

}

}

}

void Array\_Manipulation(char ch)

{

int count;

for(count = 0; count <= x; count++)

{

if(array[count] == ch)

{

return;

}

}

array[x++] = ch;

}

# Sample Output:

Enter Total Number of Productions: 8 Value of Production Number [1]: E=TD Value of Production Number [2]: D=+TD Value of Production Number [3]: D=$ Value of Production Number [4]: T=FS Value of Production Number [5]: S=\*FS Value of Production Number [6]: S=$ Value of Production Number [7]: F=(E) Value of Production Number [8]: F=a

Enter production Value to Find Follow: E

Follow Value of E: { $ ) } To Continue, Press Y: y

Enter production Value to Find Follow: D Follow Value of D: { ) }

To Continue, Press Y: n

**Experiment 12:** Write a C program for constructing of LL (1) parsing.

**Aim:** Implement a C program to construct LL (1) parsing.

# Algorithm:

1. Start the program
2. For constructing of LL(1) parsing for the given grammar. b->tb

t->cf f->i

i->@

c->f|@ b->+tb f->id

t->\*ft

1. First check for [left recursion](https://www.geeksforgeeks.org/removing-direct-and-indirect-left-recursion-in-a-grammar/) in the grammar, if there is left recursion in the grammar remove that and go to step 4.
2. Calculate First() and Follow() for all non-terminals.
   1. [First](https://www.geeksforgeeks.org/first-set-in-syntax-analysis/)(): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
   2. [Follow](https://www.geeksforgeeks.org/follow-set-in-syntax-analysis/)(): What is the Terminal Symbol which follows a variable in the process of derivation.
3. For each production A –> α. (A tends to alpha)
   1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.
   2. If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A –> α in the table.
   3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> α in

the table for the $.

1. Stop the program

**Program:** #include<stdio.h> #include<conio.h> #include<string.h> char s[20],stack[20]; int main()

{

char m[5][6][3]={"tb"," "," ","tb"," "," "," ","+tb"," "," ","n","n","fc"," "," ","fc"," "," "," ","n","\*fc"," a ","n","n","i"," "," ","(e)"," "," "};

int size[5][6]={2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0,0,3,0,0};

int i,j,k,n,str1,str2;

// clrscr();

printf("\n Enter the input string: "); scanf("%s",s);

strcat(s,"$"); n=strlen(s); stack[0]='$';

stack[1]='e';

i=1; j=0;

printf("\nStack Input\n");

printf(" \n"); while((stack[i]!='$')&&(s[j]!='$'))

{

if(stack[i]==s[j])

{

i--; j++;

}

switch(stack[i])

{

case 'e':

str1=0; break; case 'b':

str1=1; break; case 't':

str1=2; break; case 'c':

str1=3; break; case 'f':

str1=4; break;

}

switch(s[j])

{ case 'i':

str2=0; break; case '+':

str2=1; break;

case '\*': str2=2; break;

case '(': str2=3; break;

case ')': str2=4; break;

case '$': str2=5; break;

}

if(m[str1][str2][0]=='\0')

{

printf("\nERROR"); exit(0);

}

else if(m[str1][str2][0]=='n') i--;

else if(m[str1][str2][0]=='i') stack[i]='i';

else

{

for(k=size[str1][str2]-1;k>=0;k--)

{

stack[i]=m[str1][str2][k]; i++;

}

i--;

}

for(k=0;k<=i;k++) printf(" %c",stack[k]); printf(" "); for(k=j;k<=n;k++) printf("%c",s[k]); printf(" \n ");

}

printf("\n SUCCESS"); getch();

}

# Sample Output:

Enter the input string: i\*i+i

Stack Input

$ b t i\*i+i$

$ b c f i\*i+i$

$ b c i i\*i+i$

$ b c f \* \*i+i$

$ b c i i+i$

$ b +i$

$ b t + +i$

$ b c f i$

$ b c i i$

$ b $ SUCCESS

**Experiment 13:** Write a C program to construct recursive descent parsing.

**Aim:** Implement a C program to construct recursive descent parsing.

# Algorithm:

1. Write procedures for the non-terminals. Consider the grammar for arithmetic expressions E → E+T | T T → T\*F | F F → (E) | id

After eliminating the left-recursion the grammar becomes,

E → TE’

E’ → +TE’ | ε T → FT’

T’ → \*FT’ | ε

F → (E) | id

# Recursive procedure:

Procedure E() begin T( );

EPRIME( );

End

Procedure EPRIME( ) begin

If input\_symbol=’+’ then ADVANCE( ); T( ); EPRIME( );

end

Procedure T( ) begin F( );

TPRIME( );

End Procedure

TPRIME( )

begin

If input\_symbol=’\*’ then ADVANCE( ); F( ); TPRIME( );

End

Procedure F( ) begin

If input-symbol=’id’ then ADVANCE( );

else if input-symbol=’(‘ then ADVANCE( ); E( );

else if input-symbol=’)’ then

ADVANCE( );

end

else ERROR( );

1. Read the input string.
2. Verify the next token equals to non-terminals if it satisfies match for the non-terminal.
3. If the input string does not match print error.

# Program:

#include<stdio.h> #include<conio.h> #include<string.h> char input[100]; int i,l;

void main()

{

//clrscr();

printf("\nRecursive descent parsing for the following grammar\n"); printf("\nE->TE'\nE'->+TE'/@\nT->FT'\nT'->\*FT'/@\nF->(E)/ID\n"); printf("\nEnter the string to be checked:");

gets(input); if(E())

{

if(input[i+1]=='\0') printf("\nString is accepted"); else

printf("\nString is not accepted");

}

else

printf("\nString not accepted"); getch();

} E()

{ if(T())

{ if(EP())

return(1); else return(0);

}

else return(0);

} EP()

{

if(input[i]=='+')

{

i++; if(T())

{ if(EP())

return(1); else return(0);

}

else return(0);

}

else return(1);

} T()

{ if(F())

{ if(TP())

return(1); else return(0);

}

else return(0);

} TP()

{

if(input[i]=='\*')

{ i++;

if(F())

{ if(TP())

return(1); else return(0);

}

else return(0);

}

else return(1);

} F()

{

if(input[i]=='(')

{

i++; if(E())

{

if(input[i]==')')

{ i++;

return(1);

}

else return(0);

}

else return(0);

}

else if(input[i]>='a'&&input[i]<='z'||input[i]>='A'&&input[i]<='Z')

{ i++;

return(1);

}

else return(0);

}

# Sample Output:

Recursive descent parsing for the following grammar

E->TE'

E'->+TE'/@ T->FT'

T'->\*FT'/@ F->(E)/ID

Enter the string to be checked: (a+b)\*c String is accepted

Enter the string to be checked: a/c+d String is not accepted

**Experiment 14:** Write a C program for stack implementation of Shift Reduce parser.

**Aim:** Implement a C program for stack implementation of Shift Reduce parser.

# Algorithm:

1. Start the program
2. Get the input expression and store it in the input buffer.
3. Read the data from the input buffer one at the time.
4. Using stack and push & pop operation shift and reduce symbols with respect to production rules available.
5. Continue the process till symbol shift and production rule reduce reaches the start symbol.
6. Display the Stack Implementation table with corresponding Stack actions with input symbols.
7. Stop

# Program:

#include<stdio.h> #include<stdlib.h> #include<conio.h> #include<string.h>

char ip\_sym[15],stack[15]; int ip\_ptr=0,st\_ptr=0,len,i; char temp[2],temp2[2]; char act[15];

void check(); int main()

{

//clrscr();

printf("\n\t\t SHIFT REDUCE PARSER\n"); printf("\n GRAMMER\n");

printf("\n E->E+E\n E->E/E"); printf("\n E->E\*E\n E->a/b"); printf("\n enter the input symbol:\t"); gets(ip\_sym);

printf("\n\t stack implementation table"); printf("\n stack \t\t input symbol\t\t action");

printf("\n \t\t \t\t \n");

printf("\n $\t\t%s$\t\t\t--",ip\_sym); strcpy(act,"shift "); temp[0]=ip\_sym[ip\_ptr]; temp[1]='\0';

strcat(act,temp); len=strlen(ip\_sym); for(i=0;i<=len-1;i++)

{

stack[st\_ptr]=ip\_sym[ip\_ptr];

stack[st\_ptr+1]='\0'; ip\_sym[ip\_ptr]=' '; ip\_ptr++;

printf("\n $%s\t\t%s$\t\t\t%s",stack,ip\_sym,act); strcpy(act,"shift");

temp[0]=ip\_sym[ip\_ptr]; temp[1]='\0'; strcat(act,temp); check();

st\_ptr++;

}

st\_ptr++; check();

}

void check()

{

int flag=0; temp2[0]=stack[st\_ptr]; temp2[1]='\0';

if((!strcmpi(temp2,"a"))||(!strcmpi(temp2,"b")))

{

stack[st\_ptr]='E'; if(!strcmpi(temp2,"a"))

printf("\n $%s\t\t%s$\t\t\tE->a",stack,ip\_sym); else

printf("\n $%s\t\t%s$\t\t\tE->b",stack,ip\_sym); flag=1;

}

if((!strcmpi(temp2,"+"))||(strcmpi(temp2,"\*"))||(!strcmpi(temp2,"/")))

{

flag=1;

}

if((!strcmpi(stack,"E+E"))||(!strcmpi(stack,"E\E"))||(!strcmpi(stack,"E\*E")))

{

strcpy(stack,"E"); st\_ptr=0; if(!strcmpi(stack,"E+E"))

printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip\_sym); else

if(!strcmpi(stack,"E\E"))

printf("\n $%s\t\t%s$\t\t\tE->E\E",stack,ip\_sym); else

if(!strcmpi(stack,"E\*E"))

printf("\n $%s\t\t%s$\t\t\tE->E\*E",stack,ip\_sym); else

printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip\_sym); flag=1;

}

if(!strcmpi(stack,"E")&&ip\_ptr==len)

{

printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip\_sym); getch();

exit(0);

}

if(flag==0)

{

printf("\n%s\t\t\t%s\t\t reject",stack,ip\_sym); exit(0);

}

return;

}

# Sample Output:

SHIFT REDUCE PARSER

GRAMMER E->E+E

E->E/E E->E\*E

E->a/b

enter the input symbol: a+b

stack implementation table

stack input symbol action

$ a+b$ --

$a +b$ shift a

$E +b$ E->a

$E+ b$ shift +

$E+b $ shift b

$E+E $ E->b

$E $ E->E+E

$E $ ACCEPT

**Experiment 15:** Write a C program to implement operator precedence parsing.

**Aim:** Implement a C program for operator precedence parsing.

# Algorithm:

1. Start the program
2. Read the arithmetic input string.
3. Verify the precedence between terminals and symbols
4. Find the handle enclosed in < . > and reduce it to production symbol.
5. Repeat the process till we reach the start node.
6. Stop

# Program:

#include<stdio.h> #include<string.h>

char \*input; int i=0;

char lasthandle[6],stack[50],handles[][5]={")E(","E\*E","E+E","i","E^E"};

//(E) becomes )E( when pushed to stack

int top=0,l;

char prec[9][9]={

/\*input\*/

/\*stack + - \* / ^ i ( ) $ \*/

/\* + \*/ '>', '>','<','<','<','<','<','>','>',

/\* - \*/ '>', '>','<','<','<','<','<','>','>',

/\* \* \*/ '>', '>','>','>','<','<','<','>','>',

/\* / \*/ '>', '>','>','>','<','<','<','>','>',

/\* ^ \*/ '>', '>','>','>','<','<','<','>','>',

/\* i \*/ '>', '>','>','>','>','e','e','>','>',

/\* ( \*/ '<', '<','<','<','<','<','<','>','e',

/\* ) \*/ '>', '>','>','>','>','e','e','>','>',

/\* $ \*/ '<', '<','<','<','<','<','<','<','>',

};

int getindex(char c)

{

switch(c)

{

case '+':return 0;

case '-':return 1;

case '\*':return 2;

case '/':return 3;

case '^':return 4;

case 'i':return 5;

case '(':return 6;

case ')':return 7;

case '$':return 8;

}

}

int shift()

{

stack[++top]=\*(input+i++); stack[top+1]='\0';

}

int reduce()

{

int i,len,found,t; for(i=0;i<5;i++)//selecting handles

{

len=strlen(handles[i]); if(stack[top]==handles[i][0]&&top+1>=len)

{

found=1; for(t=0;t<len;t++)

{

if(stack[top-t]!=handles[i][t])

{

found=0; break;

}

}

if(found==1)

{

stack[top-t+1]='E'; top=top-t+1;

strcpy(lasthandle,handles[i]);

stack[top+1]='\0';

return 1;//successful reduction

}

}

}

return 0;

}

void dispstack()

{

int j; for(j=0;j<=top;j++)

printf("%c",stack[j]);

}

void dispinput()

{

int j; for(j=i;j<l;j++)

printf("%c",\*(input+j));

}

void main()

{

int j;

input=(char\*)malloc(50\*sizeof(char)); printf("\nEnter the string\n"); scanf("%s",input); input=strcat(input,"$"); l=strlen(input);

strcpy(stack,"$"); printf("\nSTACK\tINPUT\tACTION"); while(i<=l)

{

shift(); printf("\n"); dispstack(); printf("\t"); dispinput(); printf("\tShift");

if(prec[getindex(stack[top])][getindex(input[i])]=='>')

{

while(reduce())

{

printf("\n");

dispstack(); printf("\t"); dispinput();

printf("\tReduced: E->%s",lasthandle);

}

}

}

if(strcmp(stack,"$E$")==0) printf("\nAccepted;");

else

printf("\nNot Accepted;");

}

# Sample Output:

Enter the string i\*(i+i)\*i

STACK INPUT ACTION

$i \*(i+i)\*i$ Shift

$E \*(i+i)\*i$ Reduced: E->i

$E\* (i+i)\*i$ Shift

$E\*( i+i)\*i$ Shift

$E\*(i +i)\*i$ Shift

$E\*(E +i)\*i$ Reduced: E->i

$E\*(E+ i)\*i$ Shift

$E\*(E+i )\*i$ Shift

$E\*(E+E )\*i$ Reduced: E->i

$E\*(E )\*i$ Reduced: E->E+E

$E\*(E) \*i$ Shift

$E\*E \*i$ Reduced: E->)E(

$E \*i$ Reduced: E->E\*E

$E\* i$ Shift

$E\*i $ Shift

$E\*E $ Reduced: E->i

$E $ Reduced: E->E\*E

$E$ Shift

$E$ Shift

Accepted;

**Experiment 16:** Create YACC (or BISON) and LEX specification files to implement a basic calculator which accepts variables and constants of integer and float type.

**Aim:** Design a lexical analyzer and parser using Lex/Flex and YACC/Bison for a basic calculator.

# Algorithm:

1. Get the input from the user and parse it token by token.
2. First identify the valid inputs that can be given for a program.
3. The inputs include numbers, variables and operators.
4. Define the precedence and the associativity of various operators like +,-,/,\* etc.
5. Write codes for saving the answer into memory and displaying the result on the screen.
6. Write codes for performing various arithmetic operations.
7. Display the possible Error message that can be associated with this calculation.
8. Display the output on the screen else display the error message on the screen.

# Program:

**calc. l (flex source program)**

%{

#include <stdlib.h> void yyerror(char \*); #include "calc.tab.h"

%}

%%

/\* variables \*/ [a-z] {

yylval = \*yytext - 'a'; return VARIABLE;

}

/\* integers \*/ [0-9]+ {

yylval = atoi(yytext); return INTEGER;

}

/\* operators \*/

[-+()=/\*\n] { return \*yytext; }

/\* skip whitespace \*/ [ \t] ;

/\* anything else is an error \*/

. yyerror("invalid character");

%%

int yywrap(void) { return 1;

}

# calc.y (bison source program)

%token INTEGER VARIABLE

%left '+' '-'

%left '\*' '/'

%{

#include <stdio.h> void yyerror(char \*); int yylex(void);

int sym[26];

%}

%%

program:

statement:

program statement '\n'

|

;

expr:

%%

expr { printf("%d\n", $1); }

| VARIABLE '=' expr { sym[$1] = $3; }

;

INTEGER

| VARIABLE { $$ = sym[$1]; }

| expr '+' expr { $$ = $1 + $3; }

| expr '-' expr { $$ = $1 - $3; }

| expr '\*' expr { $$ = $1 \* $3; }

| expr '/' expr { $$ = $1 / $3; }

| '(' expr ')' { $$ = $2; }

;

void yyerror(char \*s) { fprintf(stderr, "%s\n", s);

}

int main(void) { yyparse(); return 1;

}

# Sample Output:

G:\lex>flex calc.l G:\lex>bison calc.y

G:\lex>gcc calc.tab.c lex.yy.c -o calc.exe

G:\lex>calc.exe 3\*(5+6)

33

x = 3 \* (4 + 5)

y = 5 x

27

y 5

x + 2\*y 37

G:\lex>

**Experiment 17:** Implement a simple intermediate code generator in C program, which produces three address code statements for a given input expression.

**Aim:** Design a simple intermediate code generator in C program, which produces three address code statements.

# Algorithm:

1. Begin the program.
2. The expression is read from the file using a file pointer.
3. Each string is read and the total no. of strings in the file is calculated.
4. Each string is compared with an operator, if any operator is seen then the previous string and next string are concatenated and stored in a first temporary value and the three address code expression is printed
5. Suppose if another operand is seen then the first temporary value is concatenated to the next string using the operator and the expression is printed.
6. The final temporary value is replaced to the left operand value.
7. End the program.

# Program:

#include<stdio.h> #include<conio.h> #include<stdlib.h> #include<string.h> struct three

{

char data[10],temp[7];

}s[30];

int main()

{

char d1[7],d2[7]="t"; int i=0,j=1,len=0; FILE \*f1,\*f2;

//clrscr(); f1=fopen("sum.txt","r");

f2=fopen("out.txt","w");

while(fscanf(f1,"%s",s[len].data)!=EOF) len++;

itoa(j,d1,7); strcat(d2,d1); strcpy(s[j].temp,d2);

strcpy(d1,"");

strcpy(d2,"t"); if(!strcmp(s[3].data,"+"))

{

fprintf(f2,"%s=%s+%s",s[j].temp,s[i+2].data,s[i+4].data);

j++;

}

else if(!strcmp(s[3].data,"-"))

{

fprintf(f2,"%s=%s-%s",s[j].temp,s[i+2].data,s[i+4].data); j++;

}

for(i=4;i<len-2;i+=2)

{

itoa(j,d1,7); strcat(d2,d1); strcpy(s[j].temp,d2);

if(!strcmp(s[i+1].data,"+")) fprintf(f2,"\n%s=%s+%s",s[j].temp,s[j-1].temp,s[i+2].data); else if(!strcmp(s[i+1].data,"-"))

fprintf(f2,"\n%s=%s-%s",s[j].temp,s[j-1].temp,s[i+2].data); strcpy(d1,"");

strcpy(d2,"t"); j++;

}

fprintf(f2,"\n%s=%s",s[0].data,s[j-1].temp); fclose(f1);

fclose(f2); getch();

}

# Sample Output:

**Input:** sum.txt

out = in1 + in2 + in3 - in4

**Output:** out.txt t1=in1+in2 t2=t1+in3

t3=t2-in4 out=t3