EXPERIMENT NO. 1

AIM :- To write a C program to simulate a DFA.

#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:\n Press 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;

}

EXPERIMENT NO. 2

AIM :- To write a C program to minimize the number of states of a DFA.

#include <stdio.h>

#include <string.h>

#define STATES 99

#define SYMBOLS 20

int N\_symbols; /\* number of input symbols \*/

int N\_DFA\_states; /\* number of DFA states \*/

char \*DFA\_finals; /\* final-state string \*/

int DFAtab[STATES][SYMBOLS];

char StateName[STATES][STATES+1]; /\* state-name table \*/

int N\_optDFA\_states; /\* number of optimized DFA states \*/

int OptDFA[STATES][SYMBOLS];

char NEW\_finals[STATES+1];

/\*

Print state-transition table.

State names: 'A', 'B', 'C', ...

\*/

void print\_dfa\_table(

int tab[][SYMBOLS], /\* DFA table \*/

int nstates, /\* number of states \*/

int nsymbols, /\* number of input symbols \*/

char \*finals)

{

int i, j;

puts("\nDFA: STATE TRANSITION TABLE");

/\* input symbols: '0', '1', ... \*/

printf(" | ");

for (i = 0; i < nsymbols; i++) printf(" %c ", '0'+i);

printf("\n-----+--");

for (i = 0; i < nsymbols; i++) printf("-----");

printf("\n");

for (i = 0; i < nstates; i++) {

printf(" %c | ", 'A'+i); /\* state \*/

for (j = 0; j < nsymbols; j++)

printf(" %c ", tab[i][j]); /\* next state \*/

printf("\n");

}

printf("Final states = %s\n", finals);

}

/\*

Initialize NFA table.

\*/

void load\_DFA\_table()

{

DFAtab[0][0] = 'B'; DFAtab[0][1] = 'C';

DFAtab[1][0] = 'E'; DFAtab[1][1] = 'F';

DFAtab[2][0] = 'A'; DFAtab[2][1] = 'A';

DFAtab[3][0] = 'F'; DFAtab[3][1] = 'E';

DFAtab[4][0] = 'D'; DFAtab[4][1] = 'F';

DFAtab[5][0] = 'D'; DFAtab[5][1] = 'E';

DFA\_finals = "EF";

N\_DFA\_states = 6;

N\_symbols = 2;

}

/\*

Get next-state string for current-state string.

\*/

void get\_next\_state(char \*nextstates, char \*cur\_states,

int dfa[STATES][SYMBOLS], int symbol)

{

int i, ch;

for (i = 0; i < strlen(cur\_states); i++)

\*nextstates++ = dfa[cur\_states[i]-'A'][symbol];

\*nextstates = '\0';

}

/\*

Get index of the equivalence states for state 'ch'.

Equiv. class id's are '0', '1', '2', ...

\*/

char equiv\_class\_ndx(char ch, char stnt[][STATES+1], int n)

{

int i;

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

if (strchr(stnt[i], ch)) return i+'0';

return -1; /\* next state is NOT defined \*/

}

/\*

Check if all the next states belongs to same equivalence class.

Return value:

If next state is NOT unique, return 0.

If next state is unique, return next state --> 'A/B/C/...'

's' is a '0/1' string: state-id's

\*/

char is\_one\_nextstate(char \*s)

{

char equiv\_class; /\* first equiv. class \*/

while (\*s == '@') s++;

equiv\_class = \*s++; /\* index of equiv. class \*/

while (\*s) {

if (\*s != '@' && \*s != equiv\_class) return 0;

s++;

}

return equiv\_class; /\* next state: char type \*/

}

int state\_index(char \*state, char stnt[][STATES+1], int n, int \*pn,

int cur) /\* 'cur' is added only for 'printf()' \*/

{

int i;

char state\_flags[STATES+1]; /\* next state info. \*/

if (!\*state) return -1; /\* no next state \*/

for (i = 0; i < strlen(state); i++)

state\_flags[i] = equiv\_class\_ndx(state[i], stnt, n);

state\_flags[i] = '\0';

printf(" %d:[%s]\t--> [%s] (%s)\n",

cur, stnt[cur], state, state\_flags);

if (i=is\_one\_nextstate(state\_flags))

return i-'0'; /\* deterministic next states \*/

else {

strcpy(stnt[\*pn], state\_flags); /\* state-division info \*/

return (\*pn)++;

}

}

/\*

Divide DFA states into finals and non-finals.

\*/

int init\_equiv\_class(char statename[][STATES+1], int n, char \*finals)

{

int i, j;

if (strlen(finals) == n) { /\* all states are final states \*/

strcpy(statename[0], finals);

return 1;

}

strcpy(statename[1], finals); /\* final state group \*/

for (i=j=0; i < n; i++) {

if (i == \*finals-'A') {

finals++;

} else statename[0][j++] = i+'A';

}

statename[0][j] = '\0';

return 2;

}

/\*

Get optimized DFA 'newdfa' for equiv. class 'stnt'.

\*/

int get\_optimized\_DFA(char stnt[][STATES+1], int n,

int dfa[][SYMBOLS], int n\_sym, int newdfa[][SYMBOLS])

{

int n2=n; /\* 'n' + <num. of state-division info> \*/

int i, j;

char nextstate[STATES+1];

for (i = 0; i < n; i++) { /\* for each pseudo-DFA state \*/

for (j = 0; j < n\_sym; j++) { /\* for each input symbol \*/

get\_next\_state(nextstate, stnt[i], dfa, j);

newdfa[i][j] = state\_index(nextstate, stnt, n, &n2, i)+'A';

}

}

return n2;

}

/\*

char 'ch' is appended at the end of 's'.

\*/

void chr\_append(char \*s, char ch)

{

int n=strlen(s);

\*(s+n) = ch;

\*(s+n+1) = '\0';

}

void sort(char stnt[][STATES+1], int n)

{

int i, j;

char temp[STATES+1];

for (i = 0; i < n-1; i++)

for (j = i+1; j < n; j++)

if (stnt[i][0] > stnt[j][0]) {

strcpy(temp, stnt[i]);

strcpy(stnt[i], stnt[j]);

strcpy(stnt[j], temp);

}

}

/\*

Divide first equivalent class into subclasses.

stnt[i1] : equiv. class to be segmented

stnt[i2] : equiv. vector for next state of stnt[i1]

Algorithm:

- stnt[i1] is splitted into 2 or more classes 's1/s2/...'

- old equiv. classes are NOT changed, except stnt[i1]

- stnt[i1]=s1, stnt[n]=s2, stnt[n+1]=s3, ...

Return value: number of NEW equiv. classses in 'stnt'.

\*/

int split\_equiv\_class(char stnt[][STATES+1],

int i1, /\* index of 'i1'-th equiv. class \*/

int i2, /\* index of equiv. vector for 'i1'-th class \*/

int n, /\* number of entries in 'stnt' \*/

int n\_dfa) /\* number of source DFA entries \*/

{

char \*old=stnt[i1], \*vec=stnt[i2];

int i, n2, flag=0;

char newstates[STATES][STATES+1]; /\* max. 'n' subclasses \*/

for (i=0; i < STATES; i++) newstates[i][0] = '\0';

for (i=0; vec[i]; i++)

chr\_append(newstates[vec[i]-'0'], old[i]);

for (i=0, n2=n; i < n\_dfa; i++) {

if (newstates[i][0]) {

if (!flag) { /\* stnt[i1] = s1 \*/

strcpy(stnt[i1], newstates[i]);

flag = 1; /\* overwrite parent class \*/

} else /\* newstate is appended in 'stnt' \*/

strcpy(stnt[n2++], newstates[i]);

}

}

sort(stnt, n2); /\* sort equiv. classes \*/

return n2; /\* number of NEW states(equiv. classes) \*/

}

/\*

Equiv. classes are segmented and get NEW equiv. classes.

\*/

int set\_new\_equiv\_class(char stnt[][STATES+1], int n,

int newdfa[][SYMBOLS], int n\_sym, int n\_dfa)

{

int i, j, k;

for (i = 0; i < n; i++) {

for (j = 0; j < n\_sym; j++) {

k = newdfa[i][j]-'A'; /\* index of equiv. vector \*/

if (k >= n) /\* equiv. class 'i' should be segmented \*/

return split\_equiv\_class(stnt, i, k, n, n\_dfa);

}

}

return n;

}

void print\_equiv\_classes(char stnt[][STATES+1], int n)

{

int i;

printf("\nEQUIV. CLASS CANDIDATE ==>");

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

printf(" %d:[%s]", i, stnt[i]);

printf("\n");

}

/\*

State-minimization of DFA: 'dfa' --> 'newdfa'

Return value: number of DFA states.

\*/

int optimize\_DFA(

int dfa[][SYMBOLS], /\* DFA state-transition table \*/

int n\_dfa, /\* number of DFA states \*/

int n\_sym, /\* number of input symbols \*/

char \*finals, /\* final states of DFA \*/

char stnt[][STATES+1], /\* state name table \*/

int newdfa[][SYMBOLS]) /\* reduced DFA table \*/

{

char nextstate[STATES+1];

int n; /\* number of new DFA states \*/

int n2; /\* 'n' + <num. of state-dividing info> \*/

n = init\_equiv\_class(stnt, n\_dfa, finals);

while (1) {

print\_equiv\_classes(stnt, n);

n2 = get\_optimized\_DFA(stnt, n, dfa, n\_sym, newdfa);

if (n != n2)

n = set\_new\_equiv\_class(stnt, n, newdfa, n\_sym, n\_dfa);

else break; /\* equiv. class segmentation ended!!! \*/

}

return n; /\* number of DFA states \*/

}

/\*

Check if 't' is a subset of 's'.

\*/

int is\_subset(char \*s, char \*t)

{

int i;

for (i = 0; \*t; i++)

if (!strchr(s, \*t++)) return 0;

return 1;

}

/\*

New finals states of reduced DFA.

\*/

void get\_NEW\_finals(

char \*newfinals, /\* new DFA finals \*/

char \*oldfinals, /\* source DFA finals \*/

char stnt[][STATES+1], /\* state name table \*/

int n) /\* number of states in 'stnt' \*/

{

int i;

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

if (is\_subset(oldfinals, stnt[i])) \*newfinals++ = i+'A';

\*newfinals++ = '\0';

}

void main()

{

load\_DFA\_table();

print\_dfa\_table(DFAtab, N\_DFA\_states, N\_symbols, DFA\_finals);

N\_optDFA\_states = optimize\_DFA(DFAtab, N\_DFA\_states,

N\_symbols, DFA\_finals, StateName, OptDFA);

get\_NEW\_finals(NEW\_finals, DFA\_finals, StateName, N\_optDFA\_states);

print\_dfa\_table(OptDFA, N\_optDFA\_states, N\_symbols, NEW\_finals);

}

Experiment No. 3

Aim :- To write a program using Lex to count the number of characters, words, spaces and lines in a given input file.

%{

#include<stdio.h>

int cc=0,bc=0,wc=0,lc=0;

%}

%%

[^ \t\n]+ { wc++;

cc=cc+yyleng;

}

\n lc++;

" " bc++;

\t bc=bc+5;

%%

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

{

if (argc!=2) {

printf("\nusage:./a.out filename\n");

return 0;

}

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

yylex();

printf("\n no. of lines are %d\n",lc);

printf("\n no. of words are %d\n",wc);

printf("\n no. of blanks are %d\n",bc);

printf("\n no. of character are %d\n",cc);

}

int yywrap()

{

return 1;

}

EXPERIMENT NO. 4

AIM :- Write a C program for dividing the given input program into lexemes. and also simulate lexical analyzer for validating operators

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<ctype.h>

int isKeyword(char buffer[])

{

char keywords[32][10] = {"auto","break","case","char","const","continue","default","do","double","else","enum","extern","float","for","goto","if","int","long","register","return","short","signed","sizeof","static","struct","switch","typedef","union","unsigned","void","volatile", "while"};

int i, flag = 0;

for(i = 0; i < 32; ++i)

{

 if(strcmp(keywords[i], buffer) == 0)

{

  flag = 1;

break;

 }

}

return flag;

}

int main()

{

char ch, buffer[15], operators[] = "+-\*/%=";

FILE \*fp;

int i,j=0;

fp = fopen("program.txt","r");

if(fp == NULL)

{

printf("error while opening the file\n");

exit(0);

}

while((ch = fgetc(fp)) != EOF)

{

for(i = 0; i < 6; ++i)

{

if(ch == operators[i])

printf("%c is operator\n", ch);}

    if(isalnum(ch))

{

     buffer[j++] = ch;

  }

else if((ch == ' ' || ch == '\n') && (j != 0))

{

buffer[j] = '\0';

j = 0;

if(isKeyword(buffer) == 1)

printf("%s is keyword\n", buffer);

else

printf("%s is indentifier\n", buffer);

}

}

fclose(fp);

return 0;

}

EXPERIMENT NO. 6

AIM :- To Study the structural and implementation process of lex and YACC tools and prepare a detail analysis report.

%{  
#include"y.tab.h"  
#include<math.h>  
extern yylval;  
%}  
%%  
[0-9]+ {yylval=atoi(yytext);return NUM;}  
[+] {return '+';}  
[-] {return '-';}  
[\*] {return '\*';}  
[/] {return '/';}  
[\t]+;  
[\n] {return 0;}  
.{return yytext[0];  
}  
%%

YACC PROGRAM:-  
  
%{

#include<stdio.h>  
%}  
%token NUM  
%left '-''+'  
%right '\*''/'  
%%  
start: exp {printf("%d\n",$$);}  
exp:exp'+'exp {$$=$1+$3;}  
|exp'-'exp {$$=$1-$3;}  
|exp'\*'exp {$$=$1\*$3;}  
|exp'/'exp  
{  
if($3==0)  
yyerror("error");  
else  
{  
$$=$1/$3;  
}  
}  
|'('exp')' {$$=$2;}  
|NUM {$$=$1;}  
;  
%%  
main()  
{  
printf("Enter the Expr. in terms of integers\n");  
if(yyparse()==0)  
printf("Success\n");  
}  
yywrap(){}  
yyerror()  
{  
printf("Error\n");  
}

EXPERIMENT NO. 7

AIM :- To Write a YACC program using lex to evaluate an arithmetic expression involving operators +,-,\* and /.

LEX PROGRAM:-

%{  
#include"y.tab.h"  
#include<math.h>  
extern yylval;  
%}  
%%  
[0-9]+ {yylval=atoi(yytext);return NUM;}  
[+] {return '+';}  
[-] {return '-';}  
[\*] {return '\*';}  
[/] {return '/';}  
[\t]+;  
[\n] {return 0;}  
.{return yytext[0];  
}  
%%

YACC PROGRAM:-  
  
%{

#include<stdio.h>  
%}  
%token NUM  
%left '-''+'  
%right '\*''/'  
%%  
start: exp {printf("%d\n",$$);}  
exp:exp'+'exp {$$=$1+$3;}  
|exp'-'exp {$$=$1-$3;}  
|exp'\*'exp {$$=$1\*$3;}  
|exp'/'exp  
{  
if($3==0)  
yyerror("error");  
else  
{  
$$=$1/$3;  
}  
}  
|'('exp')' {$$=$2;}  
|NUM {$$=$1;}  
;  
%%  
main()  
{  
printf("Enter the Expr. in terms of integers\n");  
if(yyparse()==0)  
printf("Success\n");  
}  
yywrap(){}  
yyerror()  
{  
printf("Error\n");  
}

EXPERIMENT NO. 8

AIM :- To write a lex program for separating integers, floating point numbers and reserve words from a file.

%{

#include<stdio.h>

%}

DIGIT [0-9]

%%

{DIGIT}\* {ECHO;printf(" Integer");}

{DIGIT}\*?\.{DIGIT}\* {ECHO;printf(" Float ");}

%%

int main()

{

yylex();

return 0;

}

EXPERIMENT NO. 9

AIM :- To Write YACC program using lex to recognize a valid variable, which starts with letter, followed by any number of letters or digits.

%{

#include "y.tab.h"

%}

%%

[a-zA-Z\_][a-zA-Z\_0-9]\* return letter;

[0-9] return digit;

. return yytext[0];

\n return 0;

%%

int yywrap()

{

return 1;

} %%

YACC PROGRAM:-

%{

#include<stdio.h>

int valid=1;

%}

%token digit letter

%%

start : letter s

s : letter s

| digit s

|

; %%

int yyerror()

{

printf("\nIts not a identifier!\n");

valid=0;

return 0;

}

int main()

{

printf("\nEnter a name to tested for identifier ");

yyparse();

if(valid)

{

printf("\nIt is a identifier!\n");

}

}

EXPERIMENT NO. 10

AIM :- **To write a YACC program for desk calculator.**

%{

   /\* Definition section \*/

  #include<stdio.h>

  int flag=0;

%}

%token NUMBER

%left '+' '-'

%left '\*' '/' '%'

%left '(' ')'

/\* Rule Section \*/

%%

ArithmeticExpression: E{

         printf("\nResult=%d\n", $$);

         return 0;

        };

 E:E'+'E {$$=$1+$3;}

 |E'-'E {$$=$1-$3;}

 |E'\*'E {$$=$1\*$3;}

 |E'/'E {$$=$1/$3;}

 |E'%'E {$$=$1%$3;}

 |'('E')' {$$=$2;}

 | NUMBER {$$=$1;}

 ;

%%

//driver code

void main()

{

   printf("\nEnter Any Arithmetic Expression which

                   can have operations Addition,

                   Subtraction, Multiplication, Divison,

                          Modulus and Round brackets:\n");

   yyparse();

   if(flag==0)

   printf("\nEntered arithmetic expression is Valid\n\n");

}

void yyerror()

{

   printf("\nEntered arithmetic expression is Invalid\n\n");

   flag=1;

}