COMPILER DESIGN

**LAB MANUAL**

# COMPILER DESIGN LAB SYLLABUS

|  |  |
| --- | --- |
| **Sl. No.** | **List of Experiments** |
| 8 | Write a C program to identify different types of Tokens in a given Program. |
| 9 | Write a Lex Program to implement a Lexical Analyzer using Lex tool. |
| 10 | Write a C program to Simulate Lexical Analyzer to validating a given input String. |
| 11 | Write a C program to implement the Brute force technique of Top down Parsing. |
| 12 | Write a C program to implement a Recursive Descent Parser. |
| 13 | Write C program to compute the First and Follow Sets for the given Grammar |
| 14 | Write a C program for eliminating the left recursion and left factoring of a given grammar |
| 15 | Write a C program to check the validity of input string using Predictive Parser. |
| 16 | Write a C program for implementation of LR parsing algorithm to accept a given input string.. |
| 17 | Write a C program for implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar |
| 18 | Simulate the calculator using LEX and YACC tool. |

**\***Content beyond the University prescribed

1. write programs that execute faster.

# EXPERIMENT- 8

#### OBJECTIVE:

Write a C program to identify different types of Tokens in a given Program

#### PROGRAM:

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

// Returns 'true' if the character is a DELIMITER.

bool isDelimiter(char ch)

{

if (ch == ' ' || ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == ',' || ch == ';' || ch == '>' ||

ch == '<' || ch == '=' || ch == '(' || ch == ')' ||

ch == '[' || ch == ']' || ch == '{' || ch == '}')

return (true);

return (false);

}

// Returns 'true' if the character is an OPERATOR.

bool isOperator(char ch)

{

if (ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == '>' || ch == '<' ||

ch == '=')

return (true);

return (false);

}

// Returns 'true' if the string is a VALID IDENTIFIER.

bool validIdentifier(char\* str)

{

if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||

str[0] == '3' || str[0] == '4' || str[0] == '5' ||

str[0] == '6' || str[0] == '7' || str[0] == '8' ||

str[0] == '9' || isDelimiter(str[0]) == true)

return (false);

return (true);

}

// Returns 'true' if the string is a KEYWORD.

bool isKeyword(char\* str)

{

if (!strcmp(str, "if") || !strcmp(str, "else") ||

!strcmp(str, "while") || !strcmp(str, "do") ||

!strcmp(str, "break") ||

!strcmp(str, "continue") || !strcmp(str, "int")

|| !strcmp(str, "double") || !strcmp(str, "float")

|| !strcmp(str, "return") || !strcmp(str, "char")

|| !strcmp(str, "case") || !strcmp(str, "char")

|| !strcmp(str, "sizeof") || !strcmp(str, "long")

|| !strcmp(str, "short") || !strcmp(str, "typedef")

|| !strcmp(str, "switch") || !strcmp(str, "unsigned")

|| !strcmp(str, "void") || !strcmp(str, "static")

|| !strcmp(str, "struct") || !strcmp(str, "goto"))

return (true);

return (false);

}

// Returns 'true' if the string is an INTEGER.

bool isInteger(char\* str)

{

int i, len = strlen(str);

if (len == 0)

return (false);

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

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' || (str[i] == '-' && i > 0))

return (false);

}

return (true);

}

// Returns 'true' if the string is a REAL NUMBER.

bool isRealNumber(char\* str)

{

int i, len = strlen(str);

bool hasDecimal = false;

if (len == 0)

return (false);

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

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' && str[i] != '.' ||

(str[i] == '-' && i > 0))

return (false);

if (str[i] == '.')

hasDecimal = true;

}

return (hasDecimal);

}

// Extracts the SUBSTRING.

char\* subString(char\* str, int left, int right)

{

int i;

char\* subStr = (char\*)malloc(

sizeof(char) \* (right - left + 2));

for (i = left; i <= right; i++)

subStr[i - left] = str[i];

subStr[right - left + 1] = '\0';

return (subStr);

}

// Parsing the input STRING.

void parse(char\* str)

{

int left = 0, right = 0;

int len = strlen(str);

while (right <= len && left <= right) {

if (isDelimiter(str[right]) == false)

right++;

if (isDelimiter(str[right]) == true && left == right) {

if (isOperator(str[right]) == true)

printf("'%c' IS AN OPERATOR\n", str[right]);

right++;

left = right;

} else if (isDelimiter(str[right]) == true && left != right

|| (right == len && left != right)) {

char\* subStr = subString(str, left, right - 1);

if (isKeyword(subStr) == true)

printf("'%s' IS A KEYWORD\n", subStr);

else if (isInteger(subStr) == true)

printf("'%s' IS AN INTEGER\n", subStr);

else if (isRealNumber(subStr) == true)

printf("'%s' IS A REAL NUMBER\n", subStr);

else if (validIdentifier(subStr) == true

&& isDelimiter(str[right - 1]) == false)

printf("'%s' IS A VALID IDENTIFIER\n", subStr);

else if (validIdentifier(subStr) == false

&& isDelimiter(str[right - 1]) == false)

printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);

left = right;

}

}

return;

}

// DRIVER FUNCTION

int main()

{

// maximum length of string is 100 here

char str[100] = "int a = b + 1c; ";

parse(str); // calling the parse function

return (0);

}

#### Output:

'int' IS A KEYWORD

'a' IS A VALID IDENTIFIER

'=' IS AN OPERATOR

'b' IS A VALID IDENTIFIER

'+' IS AN OPERATOR

'1c' IS NOT A VALID IDENTIFIER

# EXPERIMENT- 9

#### OBJECTIVE: Write a Lex Program to implement a Lexical Analyzer using Lex tool.

**STEPS:**

1.INSTALL FLEX.

2.OPEN NEW FILE LEX AND WRITE LEX PROGRAM and save it as lexp.l

3. Then Go to Tools and click Compile Lex

4.yylex.c is created. go to Tools and click Lex Build.

5.Then lexp.exe is created.

6.Then open cmd and go to editor plus in FLEX directory and write a.exe

#### PROGRAM:

/\* program name is lexp.l \*/

%{

/\* program to recognize a c program \*/

int COMMENT=0;

int cnt=0;

%}

identifier [a-zA-Z][a-zA-Z0-9]\*

%%

#.\* { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}

int |

float |

char |

double |

while |

for |

do |

if |

break |

continue |

void |

switch |

case |

long |

struct |

const |

typedef |

return |

else |

goto {printf("\n\t%s is a KEYWORD",yytext);}

"/\*" {COMMENT = 1;}

"\*/" {COMMENT = 0; cnt++;}

{identifier}\( {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}

\{ {if(!COMMENT) printf("\n BLOCK BEGINS");}

\} {if(!COMMENT) printf("\n BLOCK ENDS");}

{identifier}(\[[0-9]\*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}

\".\*\" {if(!COMMENT) printf("\n\t%s is a STRING",yytext);}

[0-9]+ {if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);}

\)(\;)? {if(!COMMENT) printf("\n\t");ECHO;printf("\n");}

\( ECHO;

= {if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}

\<= |

\>= |

\< |

== |

\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}

%%

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

{

if (argc > 1)

{

FILE \*file;

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

if(!file)

{

printf("could not open %s \n",argv[1]);

exit(0);

}

yyin = file;

}

yylex();

printf("\n\n Total No.Of comments are %d",cnt);

return 0;

}

int yywrap()

{

return 1;

}

**Input:**

 #include<stdio.h>

main()

{

int a,b;

}

**Output:**

#include<stdio.h> is a PREPROCESSOR DIRECTIVE

FUNCTION

main (

)

BLOCK BEGINS

int is a KEYWORD

a IDENTIFIER

b IDENTIFIER

BLOCK ENDS

# EXPERIMENT- 10

#### OBJECTIVE:

Write a C program to Simulate Lexical Analyzer to validating a given input String

#### PROGRAM:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main()

{

char string[50];

int count=0,i;

printf("Enter the String: ");

gets(string);

//int len=strlen(string);

//printf("length:%d",len);

if((string[0]>='a'&&string[0]<='z') || (string[0]>='A'&&string[0]<='Z') || (string[0]=='\_') || (string[0]=='$'))

{

for(i=1;i<=strlen(string);i++)

{

if((string[i]>='a'&& string[i]<='z') || (string[i]>='A' && string[i]<='Z') || (string[i]>='0'&& string[i]<='9') || (string[i]=='\_'))

{

count++;

}

}

}

if(count==(strlen(string)-1))

{

printf("Input string is a valid identifier");

}

else

{

printf("Input string is not a valid identifier");

}

return 0;

}

**OUTPUT:**

Enter the String: Welcome123

Input String is valid identifier

Enter the String:123ygfjy

Input String is not a valid identifier

# EXPERIMENT- 11

#### OBJECTIVE:

Write a C program to implement the Brute force technique of Top down Parsing

#### PROGRAM:

#include<stdio.h>

#include<conio.h>

void check(void);

void set\_value\_backtracking(void);

void get\_value\_backtracking(void);

void display\_output\_string(void);

int iptr=0,optr=0,current\_optr=0;

char output\_string[20],current\_output\_string[20],input\_string[20],temp\_string[20];

int main(){

printf("\nEnter the string to check: ");

scanf("%s",input\_string);

check();

return 0;}

void check(void){

int flag=1,rule2\_index=1;

strcpy(output\_string,"S");

printf("\nThe output string in different stages are:\n");

while(iptr<=strlen(input\_string)){

if(strcmp(output\_string,temp\_string)!=0){

display\_output\_string();}

if((iptr!=strlen(input\_string)) || (optr!=strlen(output\_string))){

if(input\_string[iptr]==output\_string[optr]){

iptr=iptr+1;

optr=optr+1;}

else{

if(output\_string[optr]=='S'){

memset(output\_string,0,strlen(output\_string));

strcpy(output\_string,"cAd");}

else if(output\_string[optr]=='A'){

set\_value\_backtracking();

if(rule2\_index==1){

memset(output\_string,0,strlen(output\_string));

strcpy(output\_string,"cabd");}

else{

memset(output\_string,0,strlen(output\_string));

strcpy(output\_string,"cad");}}

else if(output\_string[optr]=='b' && input\_string[iptr]=='d'){

rule2\_index=2;

get\_value\_backtracking();

iptr=iptr-1;}

else{

printf("\nThe given string, '%s' is invalid.\n\n",input\_string);

break;}}}

else{

printf("\nThe given string, '%s' is valid.\n\n",input\_string);

break;}}}

void set\_value\_backtracking(void){ //setting values for backtracking

current\_optr=optr;

strcpy(current\_output\_string,output\_string);

return;}

void get\_value\_backtracking(void){ //backtracking and obtaining previous values

optr=current\_optr;

memset(output\_string,0,strlen(output\_string));

strcpy(output\_string,current\_output\_string);

return;}

void display\_output\_string(void){

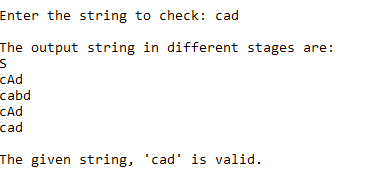
printf("%s\n",output\_string);

memset(temp\_string,0,strlen(temp\_string));

strcpy(temp\_string,output\_string);

return;}

**OUTPUT:**



# EXPERIMENT- 12

#### OBJECTIVE:

Write a C program to implement a Recursive Descent Parser.

#include<stdio.h>

#include<string.h>

#include<ctype.h>

char input[10];

int i,error;

void E();

void T();

void Eprime();

void Tprime();

void F();

main()

{

i=0;

error=0;

printf("Enter an arithmetic expression : "); // Eg: a+a\*a

gets(input);

E();

if(strlen(input)==i&&error==0)

printf("\nAccepted..!!!\n");

else printf("\nRejected..!!!\n");

}

void E()

{

T();

Eprime();

}

void Eprime()

{

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

{

i++;

T();

Eprime();

}

}

void T()

{

F();

Tprime();

}

void Tprime()

{

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

{

i++;

F();

Tprime();

}

}

void F()

{

if(isalnum(input[i]))i++;

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

{

i++;

E();

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

i++;

else error=1;

}

else error=1;

}

**OUTPUT:**

Enter an algebraic expression a+b\*c

Accepted !!!

Enter an algebraic expression a-b

Rejected !!!

# EXPERIMENT- 12

#### OBJECTIVE:

Write C program to compute the First and Follow Sets for the given Grammar

#### PROGRAM:

#include<stdio.h>

#include<ctype.h>

#include<string.h>

// Functions to calculate Follow

void followfirst(char, int, int);

void follow(char c);

// Function to calculate First

void findfirst(char, int, int);

int count, n = 0;

// Stores the final result

// of the First Sets

char calc\_first[10][100];

// Stores the final result

// of the Follow Sets

char calc\_follow[10][100];

int m = 0;

// Stores the production rules

char production[10][10];

char f[10], first[10];

int k;

char ck;

int e;

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

{

int jm = 0;

int km = 0;

int i, choice;

char c, ch;

count = 8;

// The Input grammar

strcpy(production[0], "E=TR");

strcpy(production[1], "R=+TR");

strcpy(production[2], "R=#");

strcpy(production[3], "T=FY");

strcpy(production[4], "Y=\*FY");

strcpy(production[5], "Y=#");

strcpy(production[6], "F=(E)");

strcpy(production[7], "F=i");

int kay;

char done[count];

int ptr = -1;

// Initializing the calc\_first array

for(k = 0; k < count; k++) {

for(kay = 0; kay < 100; kay++) {

calc\_first[k][kay] = '!';

}

}

int point1 = 0, point2, xxx;

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

{

c = production[k][0];

point2 = 0;

xxx = 0;

// Checking if First of c has

// already been calculated

for(kay = 0; kay <= ptr; kay++)

if(c == done[kay])

xxx = 1;

if (xxx == 1)

continue;

// Function call

findfirst(c, 0, 0);

ptr += 1;

// Adding c to the calculated list

done[ptr] = c;

printf("\n First(%c) = { ", c);

calc\_first[point1][point2++] = c;

// Printing the First Sets of the grammar

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

int lark = 0, chk = 0;

for(lark = 0; lark < point2; lark++) {

if (first[i] == calc\_first[point1][lark])

{

chk = 1;

break;

}

}

if(chk == 0)

{

printf("%c, ", first[i]);

calc\_first[point1][point2++] = first[i];

}

}

printf("}\n");

jm = n;

point1++;

}

printf("\n");

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

char donee[count];

ptr = -1;

// Initializing the calc\_follow array

for(k = 0; k < count; k++) {

for(kay = 0; kay < 100; kay++) {

calc\_follow[k][kay] = '!';

}

}

point1 = 0;

int land = 0;

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

{

ck = production[e][0];

point2 = 0;

xxx = 0;

// Checking if Follow of ck

// has already been calculated

for(kay = 0; kay <= ptr; kay++)

if(ck == donee[kay])

xxx = 1;

if (xxx == 1)

continue;

land += 1;

// Function call

follow(ck);

ptr += 1;

// Adding ck to the calculated list

donee[ptr] = ck;

printf(" Follow(%c) = { ", ck);

calc\_follow[point1][point2++] = ck;

// Printing the Follow Sets of the grammar

for(i = 0 + km; i < m; i++) {

int lark = 0, chk = 0;

for(lark = 0; lark < point2; lark++)

{

if (f[i] == calc\_follow[point1][lark])

{

chk = 1;

break;

}

}

if(chk == 0)

{

printf("%c, ", f[i]);

calc\_follow[point1][point2++] = f[i];

}

}

printf(" }\n\n");

km = m;

point1++;

}

}

void follow(char c)

{

int i, j;

// Adding "$" to the follow

// set of the start symbol

if(production[0][0] == c) {

f[m++] = '$';

}

for(i = 0; i < 10; i++)

{

for(j = 2;j < 10; j++)

{

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

{

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

{

// Calculate the first of the next

// Non-Terminal in the production

followfirst(production[i][j+1], i, (j+2));

}

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

{

// Calculate the follow of the Non-Terminal

// in the L.H.S. of the production

follow(production[i][0]);

}

}

}

}

}

void findfirst(char c, int q1, int q2)

{

int j;

// The case where we

// encounter a Terminal

if(!(isupper(c))) {

first[n++] = c;

}

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

{

if(production[j][0] == c)

{

if(production[j][2] == '#')

{

if(production[q1][q2] == '\0')

first[n++] = '#';

else if(production[q1][q2] != '\0'

&& (q1 != 0 || q2 != 0))

{

// Recursion to calculate First of New

// Non-Terminal we encounter after epsilon

findfirst(production[q1][q2], q1, (q2+1));

}

else

first[n++] = '#';

}

else if(!isupper(production[j][2]))

{

first[n++] = production[j][2];

}

else

{

// Recursion to calculate First of

// New Non-Terminal we encounter

// at the beginning

findfirst(production[j][2], j, 3);

}

}

}

}

void followfirst(char c, int c1, int c2)

{

int k;

// The case where we encounter

// a Terminal

if(!(isupper(c)))

f[m++] = c;

else

{

int i = 0, j = 1;

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

{

if(calc\_first[i][0] == c)

break;

}

//Including the First set of the

// Non-Terminal in the Follow of

// the original query

while(calc\_first[i][j] != '!')

{

if(calc\_first[i][j] != '#')

{

f[m++] = calc\_first[i][j];

}

else

{

if(production[c1][c2] == '\0')

{

// Case where we reach the

// end of a production

follow(production[c1][0]);

}

else

{

// Recursion to the next symbol

// in case we encounter a "#"

followfirst(production[c1][c2], c1, c2+1);

}

}

j++;

}

}

}

**OUTPUT:**

First(E) = { (, i, }

First(R) = { +, #, }

First(T) = { (, i, }

First(Y) = { \*, #, }

First(F) = { (, i, }

-----------------------------------------------

Follow(E) = { $, ), }

Follow(R) = { $, ), }

Follow(T) = { +, $, ), }

Follow(Y) = { +, $, ), }

Follow(F) = { \*, +, $, ), }

# EXPERIMENT- 14

#### OBJECTIVE:

#### Write a C program for eliminating the left recursion and left factoring of a given grammar

#### PROGRAM FOR LEFT RECURSION:

|  |
| --- |
|  |
|  | #include<stdio.h>  #inclide<string.h> |
|  | void main() { |
|  | char input[100],l[50],r[50],temp[10],tempprod[20],productions[25][50]; |
|  | int i=0,j=0,flag=0,consumed=0; |
|  | printf("Enter the productions: "); |
|  | scanf("%1s->%s",l,r); |
|  | printf("%s",r); |
|  | while(sscanf(r+consumed,"%[^|]s",temp) == 1 && consumed <= strlen(r)) { |
|  | if(temp[0] == l[0]) { |
|  | flag = 1; |
|  | sprintf(productions[i++],"%s->%s%s'\0",l,temp+1,l); |
|  | } |
|  | else |
|  | sprintf(productions[i++],"%s'->%s%s'\0",l,temp,l); |
|  | consumed += strlen(temp)+1; |
|  | } |
|  | if(flag == 1) { |
|  | sprintf(productions[i++],"%s->ε\0",l); |
|  | printf("The productions after eliminating Left Recursion are:\n"); |
|  | for(j=0;j<i;j++) |
|  | printf("%s\n",productions[j]); |
|  | } |
|  | else |
|  | printf("The Given Grammar has no Left Recursion"); |
| **OUTPUT:** | } |

Enter the productions: E->E+E|T

The productions after eliminating Left Recursion are:

E->+EE'

E'->TE'

E->ε

**PROGRAM FOR LEFT FACTORING:**

#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("\nGrammar Without Left Factoring : : \n");

printf(" A->%s",modifiedGram);

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

}

|  |  |
| --- | --- |
| **OUTPUT:** |  |

Enter the productions: A->Ba | Bb

Grammar Without Left Factoring:

A->BX

X->a | b

# EXPERIMENT- 15

#### OBJECTIVE:

#### Write a C program to check the validity of input string using Predictive Parser.

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

#include<ctype.h>

char ch;

#define id 0

#define CONST 1

#define mulop 2

#define addop 3

#define op 4

#define cp 5

#define err 6

32 | P a g e

#define col 7

#define size 50

int token;

char lexbuff[size];

int lookahead=0;

int main() {

clrscr();

printf(" Enter the string :");

gets(lexbuff);

parser();

return 0; }

parser() {

if(E())

printf("valid string");

else

printf("invalid string");

getch();

return 0; }

E() {

if(T()) {

if(EPRIME())

return 1;

else

return 0;

}

else

return 0; }

T() {

if(F())

{

if(TPRIME())

return 1;

else

return 0;

}

else

return 0; }

EPRIME() {

token=lexer();

33 | P a g e

if(token==addop)

{

lookahead++;

if(T())

{

if(EPRIME())

return 1;

else

return 0;

}

else

return 0;

}

els

e

return 1; }

TPRIME() {

token=lexer();

if(token==mulop)

{

lookahead++;

if(F())

{

if(TPRIME())

return 1;

else

return 0;

}

else

return 0;

}

else

return 1;

}

F() {

token=lexer();

if(token==id)

return 1;

else

{

if(token==4)

{

if(E())

{

if(token==5)

return 1;

else

34 | P a g e

return 0;

}

else

return 0;

}

else

return 0; }}

lexer() {

if(lexbuff[lookahead]!='

\n')

{

while(lexbuff[lookahead]=='

\t')

lookahead++;

if(isalpha(lexbuff[lookahead]))

{

while(isalnum(lexbuff[lookahead]))

lookahead++;

return(id);

}

else

{

if(isdigit(lexbuff[lookahead]))

{

while(isdigit(lexbuff[lookahead]))

lookahead++;

return CONST;

}

else

{

if(lexbuff[lookahead]=='+')

{

return(addop);

}

else

{

if(lexbuff[lookahead]=='\*') {

return(mulop); }

else {

if(lexbuff[lookahead]=='(')

{

lookahead++;

return(op);

}

else

35 | P a g e

{

if(lexbuff[lookahead]==')')

{

return(op);

}

else

{

return(err);

}

}

}

}

}

}

}

else

return (col);

}

**OUTPUT:**

Enter the string: id\*id+id

valid string

Enter the string: id-id

invalid string

# EXPERIMENT- 16

#### OBJECTIVE:

#### Write a C program for implementation of LR parsing algorithm to accept a given input string

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

void push(char \*,int \*,char);

char stacktop(char \*);

void isproduct(char,char);

int ister(char);

int isnter(char);

int isstate(char);

void error();

void isreduce(char,char);

char pop(char \*,int \*);

void printt(char \*,int \*,char [],int);

void rep(char [],int);

struct action

{

char row[6][5];

};

const struct action A[12]={

{"sf","emp","emp","se","emp","emp"},

{"emp","sg","emp","emp","emp","acc"},

{"emp","rc","sh","emp","rc","rc"},

{"emp","re","re","emp","re","re"},

{"sf","emp","emp","se","emp","emp"},

{"emp","rg","rg","emp","rg","rg"},

{"sf","emp","emp","se","emp","emp"},

{"sf","emp","emp","se","emp","emp"},

{"emp","sg","emp","emp","sl","emp"},

{"emp","rb","sh","emp","rb","rb"},

{"emp","rb","rd","emp","rd","rd"},

{"emp","rf","rf","emp","rf","rf"}

};

struct gotol

{

char r[3][4];

};

const struct gotol G[12]={

{"b","c","d"},

{"emp","emp","emp"},

{"emp","emp","emp"},

{"emp","emp","emp"},

{"i","c","d"},

{"emp","emp","emp"},

{"emp","j","d"},

{"emp","emp","k"},

{"emp","emp","emp"},

{"emp","emp","emp"},

};

char ter[6]={'i','+','\*',')','(','$'};

char nter[3]={'E','T','F'};

char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};

char stack[100];

int top=-1;

char temp[10];

struct grammar

{

char left;

char right[5];

};

const struct grammar rl[6]={

{'E',"e+T"},

{'E',"T"},

{'T',"T\*F"},

{'T',"F"},

{'F',"(E)"},

{'F',"i"},

};

void main()

{

char inp[80],x,p,dl[80],y,bl='a';

int i=0,j,k,l,n,m,c,len;

clrscr();

printf(" Enter the input :");

scanf("%s",inp);

len=strlen(inp);

inp[len]='$';

inp[len+1]='\0';

push(stack,&top,bl);

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

printt(stack,&top,inp,i);

do

{

x=inp[i];

p=stacktop(stack);

isproduct(x,p);

if(strcmp(temp,"emp")==0)

error();

if(strcmp(temp,"acc")==0)

break;

else

{

if(temp[0]=='s')

{

push(stack,&top,inp[i]);

push(stack,&top,temp[1]);

i++;

}

else

{

if(temp[0]=='r')

{

j=isstate(temp[1]);

strcpy(temp,rl[j-2].right);

dl[0]=rl[j-2].left;

dl[1]='\0';

n=strlen(temp);

for(k=0;k<2\*n;k++)

pop(stack,&top);

for(m=0;dl[m]!='\0';m++)

push(stack,&top,dl[m]);

l=top;

y=stack[l-1];

isreduce(y,dl[0]);

for(m=0;temp[m]!='\0';m++)

push(stack,&top,temp[m]);

}

}

}

printt(stack,&top,inp,i);

}while(inp[i]!='\0');

if(strcmp(temp,"acc")==0)

printf(" \n accept the input ");

else

printf(" \n do not accept the input ");

getch();

}

void push(char \*s,int \*sp,char item)

{

if(\*sp==100)

printf(" stack is full ");

else

{

\*sp=\*sp+1;

s[\*sp]=item;

}

}

char stacktop(char \*s)

{

char i;

i=s[top];

return i;

}

void isproduct(char x,char p)

{

int k,l;

k=ister(x);

l=isstate(p);

strcpy(temp,A[l-1].row[k-1]);

}

int ister(char x)

{

int i;

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

if(x==ter[i])

return i+1;

return 0;

}

int isnter(char x)

{

int i;

for(i=0;i<3;i++)

if(x==nter[i])

return i+1;

return 0;

}

int isstate(char p)

{

int i;

for(i=0;i<12;i++)

if(p==states[i])

return i+1;

return 0;

}

void error()

{

printf(" error in the input ");

exit(0);

}

void isreduce(char x,char p)

{

int k,l;

k=isstate(x);

l=isnter(p);

strcpy(temp,G[k-1].r[l-1]);

}

char pop(char \*s,int \*sp)

{

char item;

if(\*sp==-1)

printf(" stack is empty ");

else

{

item=s[\*sp];

\*sp=\*sp-1;

}

return item;

}

void printt(char \*t,int \*p,char inp[],int i)

{

int r;

printf("\n");

for(r=0;r<=\*p;r++)

rep(t,r);

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

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

30

printf("%c",inp[r]);

}

void rep(char t[],int r)

{

char c;

c=t[r];

switch(c)

{

case 'a': printf("0");

break;

case 'b': printf("1");

break;

case 'c': printf("2");

break;

case 'd': printf("3");

break;

case 'e': printf("4");

break;

case 'f': printf("5");

break;

case 'g': printf("6");

break;

case 'h': printf("7");

break;

case 'm': printf("8");

break;

case 'j': printf("9");

break;

case 'k': printf("10");

break;

case 'l': printf("11");

break;

default :printf("%c",t[r]);

break;

}

}

**OUTPUT:**

Enter the input: i\*i+1

Output

Stack input

0 I \*i+i$

0i5 \*i+i$

0F3 \*i+i$

0T2 \*i+i$

0T2\*7 i+i$

0T2\*7i5 +i$

0T2\*7i5F10 +i$

0T2 +i$

0E1 +i$

0E1+6 i$

0E1+6i5 $

0E1+6F3 $

0E1+6T9 $

0E1 $

accept the input\*

# EXPERIMENT- 17

#### OBJECTIVE:

#### Write a C program for implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar

**PROGRAM:**

#include<stdio.h>

#include<string.h>

int k=0,z=0,i=0,j=0,c=0;

char a[16],ac[20],stk[15],act[10];

void check();

int main()

{

puts("GRAMMAR is E->E+E \n E->E\*E \n E->(E) \n E->id");

puts("enter input string ");

gets(a);

c=strlen(a);

strcpy(act,"SHIFT->");

puts("stack \t input \t action");

for(k=0,i=0; j<c; k++,i++,j++)

{

if(a[j]=='i' && a[j+1]=='d')

{

stk[i]=a[j];

stk[i+1]=a[j+1];

stk[i+2]='\0';

a[j]=' ';

a[j+1]=' ';

printf("\n$%s\t%s$\t%sid",stk,a,act);

check();

}

else

{

stk[i]=a[j];

stk[i+1]='\0';

a[j]=' ';

printf("\n$%s\t%s$\t%ssymbols",stk,a,act);

check();

}

}

}

void check()

{

strcpy(ac,"REDUCE TO E");

for(z=0; z<c; z++)

if(stk[z]=='i' && stk[z+1]=='d')

{

stk[z]='E';

stk[z+1]='\0';

printf("\n$%s\t%s$\t%s",stk,a,ac);

j++;

}

for(z=0; z<c; z++)

if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')

{

stk[z]='E';

stk[z+1]='\0';

stk[z+2]='\0';

printf("\n$%s\t%s$\t%s",stk,a,ac);

i=i-2;

}

for(z=0; z<c; z++)

if(stk[z]=='E' && stk[z+1]=='\*' && stk[z+2]=='E')

{

stk[z]='E';

stk[z+1]='\0';

stk[z+1]='\0';

printf("\n$%s\t%s$\t%s",stk,a,ac);

i=i-2;

}

for(z=0; z<c; z++)

if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')

{

stk[z]='E';

stk[z+1]='\0';

stk[z+1]='\0';

printf("\n$%s\t%s$\t%s",stk,a,ac);

i=i-2;

}

}

**OUTPUT:**

GRAMMAR is E->E+E

E->E\*E

E->(E)

E->id enter input string

id+id\*id+id

stack input action

$id +id\*id+id$ SHIFT->id

$E +id\*id+id$ REDUCE TO E

$E+ id\*id+id$ SHIFT->symbols

$E+id \*id+id$ SHIFT->id

$E+E \*id+id$ REDUCE TO E

$E \*id+id$ REDUCE TO E

$E\* id+id$ SHIFT->symbols

$E\*id +id$ SHIFT->id

$E\*E +id$ REDUCE TO E

$E +id$ REDUCE TO E

$E+ id$ SHIFT->symbols

$E+id $ SHIFT->id

$E+E $ REDUCE TO E

$E $ REDUCE TO E

# EXPERIMENT- 18

#### OBJECTIVE: Simulate the calculator using LEX and YACC tool.

**PROGRAM:**

%{

   /\* 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, Division,

                          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;

}

**OUTPUT:**

**Input:** 4+5

**Output:** Result=9

Entered arithmetic expression is Valid

**Input:** 10-5

**Output:** Result=5

Entered arithmetic expression is Valid

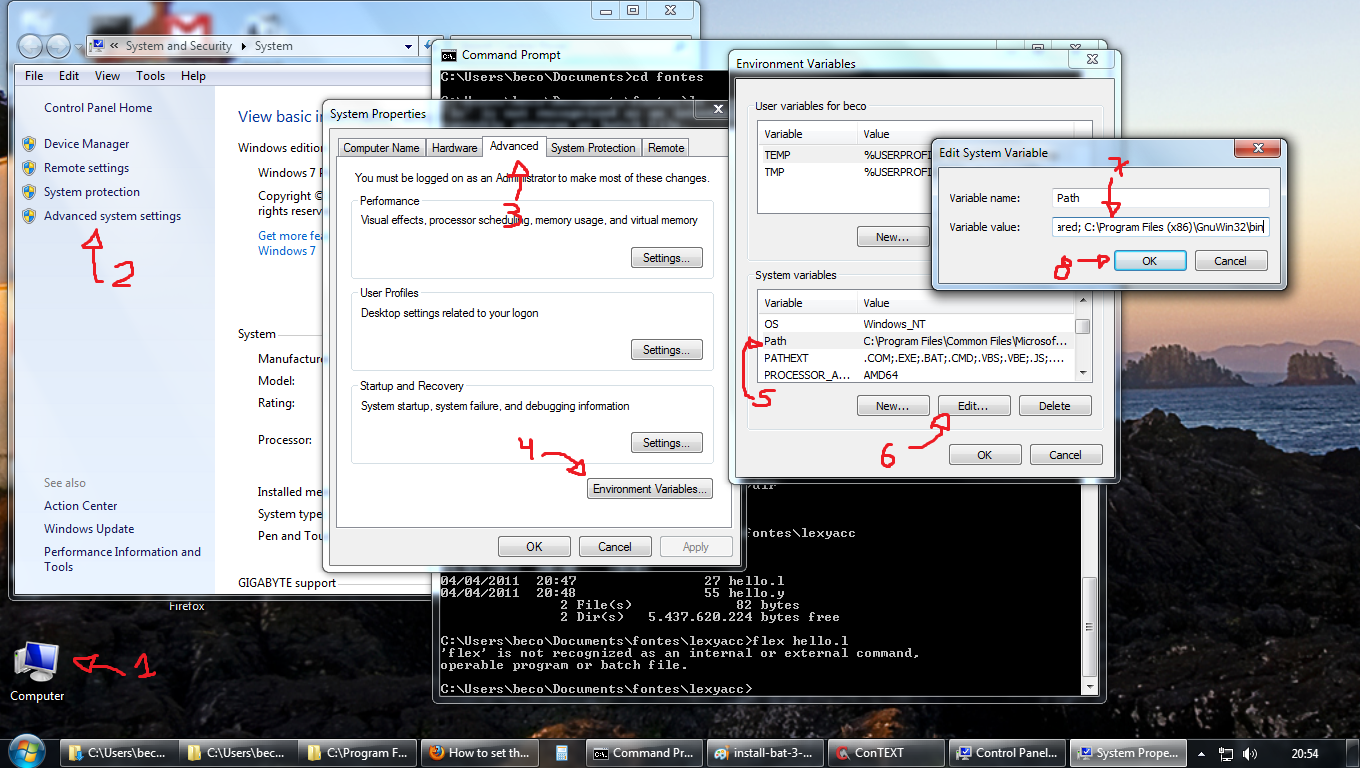
**Input:** 10+5-

**Output:**

Entered arithmetic expression is Invalid

---------------------------------------------------------------------------------------------------------------------------------------------------

(**This is just for reference for exercise 18 don’t write in Record**)

1. [flex-2.5.4a-1.exe](http://gnuwin32.sourceforge.net/packages/flex.htm)
2. [bison-2.4.1-setup.exe](http://gnuwin32.sourceforge.net/packages/bison.htm)
3. After that, do a full install in a directory of your preference **without spaces in the name**. I suggest C:\GnuWin32. Do **not** install it in the *default* (C:\Program Files (x86)\GnuWin32) because bison has problems with spaces in directory names, not to say parenthesis.
4. Also, consider installing [Dev-CPP](http://www.bloodshed.net/dev/devcpp.html) in the default directory (C:\Dev-Cpp)
5. After that, set the PATH variable to include the bin directories of gcc (in C:\Dev-Cpp\bin) and flex\bison (in C:\GnuWin32\bin). To do that, copy this: ;C:\Dev-Cpp\bin;C:\GnuWin32\bin and append it to the end of the PATH variable, defined in the place show by this figure:  
   

If the figure is not in good resolution, you can see a [step-by-step here](http://geekswithblogs.net/renso/archive/2009/10/21/how-to-set-the-windows-path-in-windows-7.aspx).

1. Open a prompt, cd to the directory where your ".l" and ".y" are, and compile them with:
   1. flex hello.l
   2. bison -dy hello.y
   3. gcc lex.yy.c y.tab.c -o hello.exe

