1. Write a C program to identify different types of Tokens in a given 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);

}

Exp3: Write a C program to Simulate Lexical Analyzer to validating a given input String.

#include <stdio.h>

#include <string.h>

int main ()

{

char arithmetic[5]={'+','-','\*','/','%'};

char relational[4]={'<','>','!','='};

char bitwise[5]={'&','^','~','|'};

char str[2]={' ',' '};

printf ("Enter value to be identified: ");

scanf ("%s",&str);

int i;

if(((str[0]=='&'||str[0]=='|') && str[0]==str[1])||(str[0]=='!' && str[1]=='\0'))

{

printf("\nIt is Logical operator");

}

for(i=0;i<4;i++)

{

if(str[0]==relational[i]&&(str[1]=='='||str[1]=='\0'))

{

printf("\n It is releational Operator"); break;

}

}

for(i=0;i<4;i++)

{

if((str[0]==bitwise[i] && str[1]=='\0') || ((str[0]=='<' || str[0]=='>') && str[1]==str[0]))

{

printf("\n It is Bitwise Operator"); break;

}

}

if(str[0]=='?'&&str[1]==':')

printf("\nIt is ternary operator");

for(i=0;i<5;i++)

{

if((str[0]=='+' || str[0]=='-') && str[0]==str[1])

{

printf("\nIt is unary operator"); break;

}

else if((str[0]==arithmetic[i] && str[1]=='=') || (str[0]=='=' && str[1]==' '))

{

printf("\nIt is Assignment operator"); break;

}

else if(str[0]==arithmetic[i] && str[1]=='\0')

{

printf("\nIt is arithmetic operator"); break;

}

}

return 0;

}

**OUTPUT1** **OUTPUT2**

Enter value to be identified: >= Enter value to be identified: &

It is releational Operator It is Bitwise Operator

=== Code Execution Successful === === Code Execution Successful ===

**Exp4:** **Write a C program to implement the Brute force technique of Top-down Parsing.**

**Consider the CFG: S -> cAd, A -> ab | a .**

#include <stdio.h>

#include <stdbool.h>

#include <string.h>

bool parse(char \*);

int main() {

char input[100];

printf("Enter a string to parse: ");

scanf("%s", input);

if (parse(input)) {

printf("String is successfully parsed.\n");

} else {

printf("Error in parsing string.\n");

}

return 0;

}

bool parse(char \*input) {

int index = 0;

if (input[index] == 'c') {

index++;

if (input[index] == 'a') {

index++;

if (input[index] == 'b' || input[index] == 'd') {

index++;

if (input[index] == 'd' || input[index] == '\0') {

return true;

}

}

}

}

return false;

}

**OUTPUT1:**

**Enter a string to parse: ccabdd**

**Error in parsing string.**

**OUTPUT2:**

**Enter a string to parse: cad**

**String is successfully parsed.**

**5.**

#include <stdio.h>

#include <string.h>

#define SUCCESS 1

#define FAILED 0

int E(), Edash(), T(), Tdash(), F();

const char \*cursor;

char string[64];

int main()

{

puts("CFG for implementing Recursive Descent Parser:\nE -> T E'\nE' -> + T E' | e\nT -> F T'\nT' -> \* F T' | e\nF ->( E ) | i \nEnter the string");

scanf("%s", string);

cursor = string;

puts(string);

puts("Input Action");

puts("--------------------------------");

if (E() && \*cursor == '\0') {

puts("--------------------------------");

puts("String is successfully parsed");

return 0;

} else {

puts("--------------------------------");

puts("Error in parsing String");

return 1;

}

}

int E()

{

printf("%-16s E -> T E'\n", cursor);

if (T()) {

if (Edash())

return SUCCESS;

else

return FAILED;

} else

return FAILED;

}

int Edash()

{

if (\*cursor == '+') {

printf("%-16s E' -> + T E'\n", cursor);

cursor++;

if (T()) {

if (Edash())

return SUCCESS;

else

return FAILED;

} else

return FAILED;

} else {

printf("%-16s E' -> $\n", cursor);

return SUCCESS;

}

}

int T()

{

printf("%-16s T -> F T'\n", cursor);

if (F()) {

if (Tdash())

return SUCCESS;

else

return FAILED;

} else

return FAILED;

}

int Tdash()

{

if (\*cursor == '\*') {

printf("%-16s T' -> \* F T'\n", cursor);

cursor++;

if (F()) {

if (Tdash())

return SUCCESS;

else

return FAILED;

} else

return FAILED;

} else {

printf("%-16s T' -> $\n", cursor);

return SUCCESS;

}

}

int F()

{

if (\*cursor == '(') {

printf("%-16s F -> ( E )\n", cursor);

cursor++;

if (E()) {

if (\*cursor == ')') {

cursor++;

return SUCCESS;

} else

return FAILED;

} else

return FAILED;

} else if (\*cursor == 'i') {

cursor++;

printf("%-16s F ->i\n", cursor);

return SUCCESS;

} else

return FAILED;

}

**Exp6: FIRST and FOLLOW Computation**

#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;

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, x;

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

{

c = production[k][0];

point2 = 0;

x = 0;

// Checking if First of c has

// already been calculated

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

if(c == done[kay])

x = 1;

if (x == 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;

x = 0;

// Checking if Follow of ck

// has alredy been calculated

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

if(ck == donee[kay])

x = 1;

if (x == 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++;

}

}

}

7a: Left Recursion for expression grammar using C++.

/\*

E->E+T/T

T->T\*F/F

F->(E)/id

\*/

#include <bits/stdc++.h>

using namespace std;

void removeLeftRecursion(string str) {

int ind = 0;

string s1="",s2="";

for (int i = 0; i < str.size(); i++) {

if (str[i] == '/') {

ind = i;

break;

}

}

if(!isupper(str[0]))

{

cout<<"Invalid Production";

}

else if (str[0] == str[3] && ind != 0) {

s1 = string(1,str[0]) + " -> " + string(1, str[ind + 1]) + string(1, str[0]) + "'";

s2 = string(1, str[0]) + "'"+ " -> "+str.substr(4,str.size()-ind) + string(1, str[0]) + "'"+string(1, str[ind])+"ε";

cout << s1<<"\n";

cout<< s2 <<"\n";

}

else

{

cout<<str<<"\n";

}

}

int main() {

int n;

cout<<"Enter your number of production: ";

cin>>n;

string str[n];

cout << "Enter a grammar: \n";

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

{

cin >> str[i];

}

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

{

removeLeftRecursion(str[i]);

}

}

**7.b)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 Production : A->bE+acf|bE+f

Grammar Without Left Factoring: :

A->bE+X

X->acf | f

**8. Write a C program to implement predictive parser**

**Description**:

Predictive parsing is one of the top-down parsing technique. In this parsing the parse table is used with input. The parse table doesn’t contain multiple entries. So the grammar which is acceptable by this parser is LL(1) grammar. There is no backtracking in predictive parsing.

**Algorithm:**

**Algorithm for function FIRST:**

1.if x is terminal then FIRST(x) is {x}.

2.if x is a non terminal and x ->a a(alpha) is a production then add a to FIRST(x). If x->e(epsilon) is a production then add e to FIRST(x).

3. If x->Y1Y2Y3…..Yk is a production, then for all I such that all of Y1,Y2,….Yi-1 are non terminals and FIRST(Yj) contains e for j=1,2…i-1(i.e Y1,Y2,…Yi=1\*=>E).add every non e-symbol in FIRST(Yi) to FIRST(x). If e is in FIRST(Yj) for all j=1,2…k, then add e to FIRST(x).

**Algorithm for function FOLLOW:**

1.$ IS IN follow(s), where S is the start symbol.

2. If there is a production A->a(alpha)Bb(beta),b not equal e, then everything in FIRST(b) but e is in FOLLOW(B).

3. If there is a production A->aB, or a production A->aBb where FIRST(B) contains e(i.e.b\*=>e), then everything in FOLLOW(A) is in FOLLOW(B).

**Algorithm for CONSTABLE(predictive parsing table):**

1. repeat
2. begin

let X be the top stack symbol and if a the next input symbol

3. if X is a terminal or $ then

4.if X=a then

Pop **X** from the stack and remove a from the input

1. else

ERROR()

1. else/\* X is a nonterminal\*/
2. if M[X,a]=X->Y1,Y2,……,Yk then
3. begin

pop X from the stack

push Yk,Yk-1,………..Y1 on to the stack,Y1 on top

1. end
2. else

ERROR()

1. end/\*first begin end\*/
2. until

X=$/\* stack has emptied\*/

**Algorithm for function PARSE:**

1. If X=a=$, the parser halts and announces successful completion of parsing.
2. If X=a=$, the parser pops the X off the stack and advances the input pointer to the next input symbol.

3. If X is a non terminal, the program consults entry M[X, a] of the parsing table M. This entry will be either an X production of the grammar or an error entry. If M[X, a]={X->UVW}, the parser replaces X on top of the stack by WVU(with U on top). As output, the grammar does the symmetric action associated with this production, which, for the time being, we shall assume is just printing the production used. If M[X, a]=error, the parser calls an error recovery routine.

**Program:**

/\*program to implement PREDICTIVE PARSER \*/

#include<stdio.h>

#include<stdlib.h>

int stack[20],top=-1;

void push(int item)

{

if(top>=20)

{

printf("STACK OVERFLOW");

exit(1);

}

stack[++top]=item;

}

int pop()

{

int ch;

if(top<=-1)

{

printf("underflow");

exit(1);

}

ch=stack[top--];

return ch;

}

char convert(int item)

{

char ch;

switch(item)

{

case 0:return('E');

case 1:return('e');

case 2:return('T');

case 3:return('t');

case 4:return('F');

case 5:return('i');

case 6:return('+');

case 7:return('\*');

case 8:return('(');

case 9:return(')');

case 10:return('$');

}

}

void main()

{

int m[10][10],i,j,k;

char ips[20];

int ip[10],a,b,t;

m[0][0]=m[0][3]=21;

m[1][1]=621;

m[1][4]=m[1][5]=-2;

m[2][0]=m[2][3]=43;

m[3][1]=m[3][4]=m[3][5]=-2;

m[3][2]=743;

m[4][0]=5;

m[4][3]=809;

printf("\n Given CFG: \n E->E+T/T \n T->T\*F/F \n F->(E)/i\n Enter the input string Ending with $(Ex:i+i$,i+(i+i)$ etc..):");

scanf("%s",ips);

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

{

switch(ips[i])

{

case 'E':k=0;break;

case 'e':k=1;break;

case 'T':k=2;break;

case 't':k=3;break;

case 'F':k=4;break;

case 'i':k=5;break;

case '+':k=6;break;

case '\*':k=7;break;

case '(':k=8;break;

case ')':k=9;break;

case '$':k=10;break;

}

ip[i]=k;

}

ip[i]=-1;

push(10);

push(0);

i=0;

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

while(1)

{

printf("\t\t");

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

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

printf("\t\t");

for(k=i;ip[k]!=-1;k++)

printf("%c",convert(ip[k]));

printf("\n");

if(stack[top]==ip[i])

{

if(ip[i]==10)

{

printf("\t\t SUCCESS");

return;

}

else

{

top--;

i++;

}

}

else if(stack[top]<=4&&stack[top]>=0)

{

a=stack[top];

b=ip[i]-5;

t=m[a][b];

top--;

while(t>0)

{

push(t%10);

t=t/10;

}

}

else

{

printf("ERROR");

return;

}

}

}

**OUTPUT:**

Given CFG:

E->E+T/T

T->T\*F/F

F->(E)/i

Enter the input string Ending with $(Ex:i+i$,i+(i+i)$ etc..):i+(i\*i)$

stack input

$E i+(i\*i)$

$eT i+(i\*i)$

$etF i+(i\*i)$

$eti i+(i\*i)$

$et +(i\*i)$

$e +(i\*i)$

$eT+ +(i\*i)$

$eT (i\*i)$

$etF (i\*i)$

$et)E( (i\*i)$

$et)E i\*i)$

$et)eT i\*i)$

$et)etF i\*i)$

$et)eti i\*i)$

$et)et \*i)$

$et)etF\* \*i)$

$et)etF i)$

$et)eti i)$

$et)et )$

$et)e )$

$et) )$

$et $

$e $

$ $

SUCCESS

**Exp9:** **Write a C Program for implementation of LR Parsing algorithm to accept a given input string.**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

//Global Variables

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

// Modify array size to increase

// length of string to be parsed

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

// This Function will check whether

// the stack contain a production rule

// which is to be Reduce.

// Rules can be E->2E2 , E->3E3 , E->4

void check()

{

// Copying string to be printed as action

strcpy(ac,"REDUCE TO E -> ");

// c=length of input string

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

{

//checking for producing rule E->4

if(stk[z] == '4')

{

printf("%s4", ac);

stk[z] = 'E';

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

//printing action

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

}

}

for(z = 0; z < c - 2; z++)

{

//checking for another production

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

stk[z + 2] == '2')

{

printf("%s2E2", ac);

stk[z] = 'E';

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

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

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

i = i - 2;

}

}

for(z=0; z<c-2; z++)

{

//checking for E->3E3

if(stk[z] == '3' && stk[z + 1] == 'E' &&

stk[z + 2] == '3')

{

printf("%s3E3", ac);

stk[z]='E';

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

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

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

i = i - 2;

}

}

return ; //return to main

}

//Driver Function

int main()

{

char str[50];

printf("Consider the CFG -\nE->2E2 \nE->3E3 \nE->4\n");

printf("\nEnter input string from given CFG: ");

scanf("%s",str);

// a is input string

strcpy(a,str);

// strlen(a) will return the length of a to c

c=strlen(a);

// "SHIFT" is copied to act to be printed

strcpy(act,"SHIFT");

// This will print Labels (column name)

printf("\nStack \t\tInput \t\t Action");

// This will print the initial

// values of stack and input

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

// This will Run upto length of input string

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

{

// Printing action

printf("\t%s", act);

// Pushing into stack

stk[i] = a[j];

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

// Moving the pointer

a[j]=' ';

// Printing action

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

// Call check function ..which will

// check the stack whether its contain

// any production or not

check();

}

// Rechecking last time if contain

// any valid production then it will

// replace otherwise invalid

check();

// if top of the stack is E(starting symbol)

// then it will accept the input

if(stk[0] == 'E' && stk[1] == '\0')

printf("Accept\n");

else //else reject

printf("Reject\n");

}

**OUTPUT1**

GRAMMAR is -

E->2E2

E->3E3

E->4

Enter input string from given CFG: 2324232

stack input action

$ 2324232$ SHIFT

$2 324232$ SHIFT

$23 24232$ SHIFT

$232 4232$ SHIFT

$2324 232$ REDUCE TO E -> 4

$232E 232$ SHIFT

$232E2 32$ REDUCE TO E -> 2E2

$23E 32$ SHIFT

$23E3 2$ REDUCE TO E -> 3E3

$2E 2$ SHIFT

$2E2 $ REDUCE TO E -> 2E2

$E $ Accept

GRAMMAR is -

E->2E2

E->3E3

E->4

**OUTPUT2**

Enter input string from given CFG: 1234

stack input action

$ 1234$ SHIFT

$1 234$ SHIFT

$12 34$ SHIFT

$123 4$ SHIFT

$1234 $ REDUCE TO E -> 4

$123E $ Reject

=== Code Execution Successful ===

**Exp10: Write a C program for the implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar.**

#include<stdio.h>

#include<stdlib.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()

{

printf("\n\n\t Shift Reduce Parser\n");

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

printf("\nGrammar\n");

printf("E->E+E\nE->E/E\n");

printf("E->E\*E\nE->a/b");

printf("\nEnter the Input Symbol:");

scanf("%s",ip\_sym);

printf("\n\n\t Stack Implementation Table");

printf("\n Stack\t\t Input Symbol\t\t Action");

printf("\n $\t\t\t%s$\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();

return 0;

}

void check()

{

int flag=0;

temp2[0]=stack[st\_ptr];

temp[1]='\0';

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

{

stack[st\_ptr]='E';

if(!strcmp(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->a",stack,ip\_sym);

flag=1;

}

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

{

flag=1;

}

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

{

strcpy(stack,"E");

st\_ptr=0;

if(!strcmp(stack,"E+E"))

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

else

if(!strcmp(stack,"E/E"))

printf("\n$%s\t\t\t%s$\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(!strcmp(stack,"E")&&ip\_ptr==len)

{

printf("\n$%s\t\t%s$\t\t\tAccept",ip\_sym);

exit(0);

}

if(flag==0)

{

printf("\n %s \t\t\t %s \t\t Reject",stack,ip\_sym);

}

return;

}

**OUTPUT1**

Shift Reduce Parser

\*\*\*\*\* \*\*\*\*\*\* \*\*\*\*\*\*

Grammar

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$$ shifta

$E +b$$ E->a

$E+ b$$ shift+

$E+b $$ shiftb

$E+E $$ E->a

$E $$ E->E\*E

$E$ $ shift$

=== Code Execution Successful ===