**COMPILER CONSTRUCTION**

**CSE304**

**Practical file**

****

**AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY**

**AMITY UNIVERSITY, UTTAR PRADESH**

**Submitted by:**

**MOHIT**

**A2305220680**

**6CSE2X**

**Submitted to:**

**Ms. Neetu Narayan**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.no.** | **Program** | **Date** | **Sign** |
| 1. | To convert a given infix expression to postfix and evaluate that postfix expression. | 05/01/2023 |  |
| 2. | To construct a Deterministic Finite Automata for Regular Expression. | 12/01/2023 |  |
| 3. | To count the number of tokens in a program. | 03/02/2023 |  |
| 4. | To count the number and type of tokens in a program. | 17/02/2023 |  |
| 5. | To remove the Left Recursion from a given grammar. | 24/02/2023 |  |
| 6. | To remove ambiguity from a grammar. | 03/03/2023 |  |
| 7. | To construct predictive parser. | 10/03/2023 |  |
| 8. | To compute First and Follow for a grammar. | 24/03/2023 |  |
| 9. | To compute Leading and Trailing for a grammar. | 31/03/2023 |  |
| 10. | To design a Recursive Descent Parser. | 14/04/2023 |  |

**LAB-1**

**AIM:** To convert a given infix expression to postfix and evaluate that postfix expression

**THEORY**

Infix expressions are the expressions which have the operators written between the operands while postfix expressions have operators written after the operands.

Infix expression is of the form: *50 + ( 2 \* 40 - 1 ) + 8 / 5*

Postfix expression corresponding to this is: *50 2 40 \* 1 - + 8 5 / +*

**Steps to convert an infix expression to postfix:**

1. Insert ‘)’ at the end of the infix expression and push ‘(‘ onto the stack.
2. Traverse the infix expression, if encountered an operand, write it to the postfix notation and if encountered an operator push it onto the stack if the top of the stack has an operator with lesser precedence than the encountered operator or else pop the stack till we get an operator of lesser precedence on the top of the stack.
3. If encountered ‘(‘, push onto the stack if encountered ‘)’ pop all the elements from the stack till the time we get ‘(‘ on the top of the stack.
4. At last, pop all the elements from the stack and insert them into the postfix expression.

**Steps to evaluate postfix expression:**

1. If encountered an operand, push it into the stack, when encountered an operator, pop 2 elements from the stack, perform the specified operation on these 2 elements and push the result into the stack.
2. The last step would have only 1 element on the stack and that element would be the evaluated result.

**SOURCE CODE**

#include<iostream>

#include<stack>

#include<string>

using namespace std;

string infix;

string postfix;

void conversion()

{

stack<char> s;

infix+=' ';

infix+=')';

s.push('(');

for(int i=0;i<=infix.length()-1;i++)

{

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

{

s.push(infix[i]);

}

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

{

while(s.top() != '(')

{

if(postfix.back()==' ')

{

postfix += s.top();

s.pop();

}

else

{

postfix+=' ';

postfix += s.top();

s.pop();

}

}

s.pop();

}

else if(infix[i]=='+'|| infix[i]=='-'|| infix[i]=='\*'|| infix[i]=='/')

{

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

{

while(s.top()=='\*'||s.top()=='/'||s.top()=='-'||s.top()=='+')

{

if(postfix.back()==' ')

{

postfix+=s.top();

s.pop();

}

else

{

postfix+=' ';

postfix+=s.top();

s.pop();

}

}

s.push(infix[i]);

}

if(infix[i]=='-')

{

while(s.top()=='\*'||s.top()=='/'||s.top()=='+'||s.top()=='-')

{

if(postfix.back()==' ')

{

postfix+=s.top();

s.pop();

}

else

{

postfix+=' ';

postfix+=s.top();

s.pop();

}

}

s.push(infix[i]);

}

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

{

while(s.top()=='/'||s.top()=='\*')

{

if(postfix.back()==' ')

{

postfix+=s.top();

s.pop();

}

else

{

postfix+=' ';

postfix+=s.top();

s.pop();

}

}

s.push(infix[i]);

}

if(infix[i]=='/')

{

while(s.top()=='\*'||s.top()=='/')

{

if(postfix.back()==' ')

{

postfix+=s.top();

s.pop();

}

else

{

postfix+=' ';

postfix+=s.top();

s.pop();

}

}

s.push(infix[i]);

}

}

else if(isdigit(infix[i]))

{

postfix += infix[i];

}

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

{

if(postfix.back()==' ')

{

continue;

}

else

{

postfix+=' ';

}

}

}

cout<<"Postfix expression: "<<postfix;

}

void evaluation()

{

stack<float> s;

float num1, num2, output;

for(int i=0;i<=postfix.length()-1;i++)

{

if(postfix[i]=='+'||postfix[i]=='-'||postfix[i]=='\*'||postfix[i]=='/')

{

num1=s.top();

s.pop();

num2=s.top();

s.pop();

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

{

output=num2+num1;

s.push(output);

}

else if(postfix[i]=='-')

{

output=num2-num1;

s.push(output);

}

else if(postfix[i]=='\*')

{

output=num2\*num1;

s.push(output);

}

else if(postfix[i]=='/')

{

output=num2/num1;

s.push(output);

}

}

else if (isdigit(postfix[i]))

{

string tempstr;

while(!isspace(postfix[i]))

{

tempstr+=postfix[i];

i++;

}

s.push(stof(tempstr));

}

else if(isspace(postfix[i]))

{

continue;

}

}

cout<<"The value is: "<<s.top();

}

int main()

{

cout<<"Enter the infix expression ";

getline(cin,infix);

conversion();

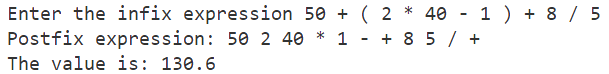
cout<<"\n";

evaluation();

return 0;

}

**OUTPUT**

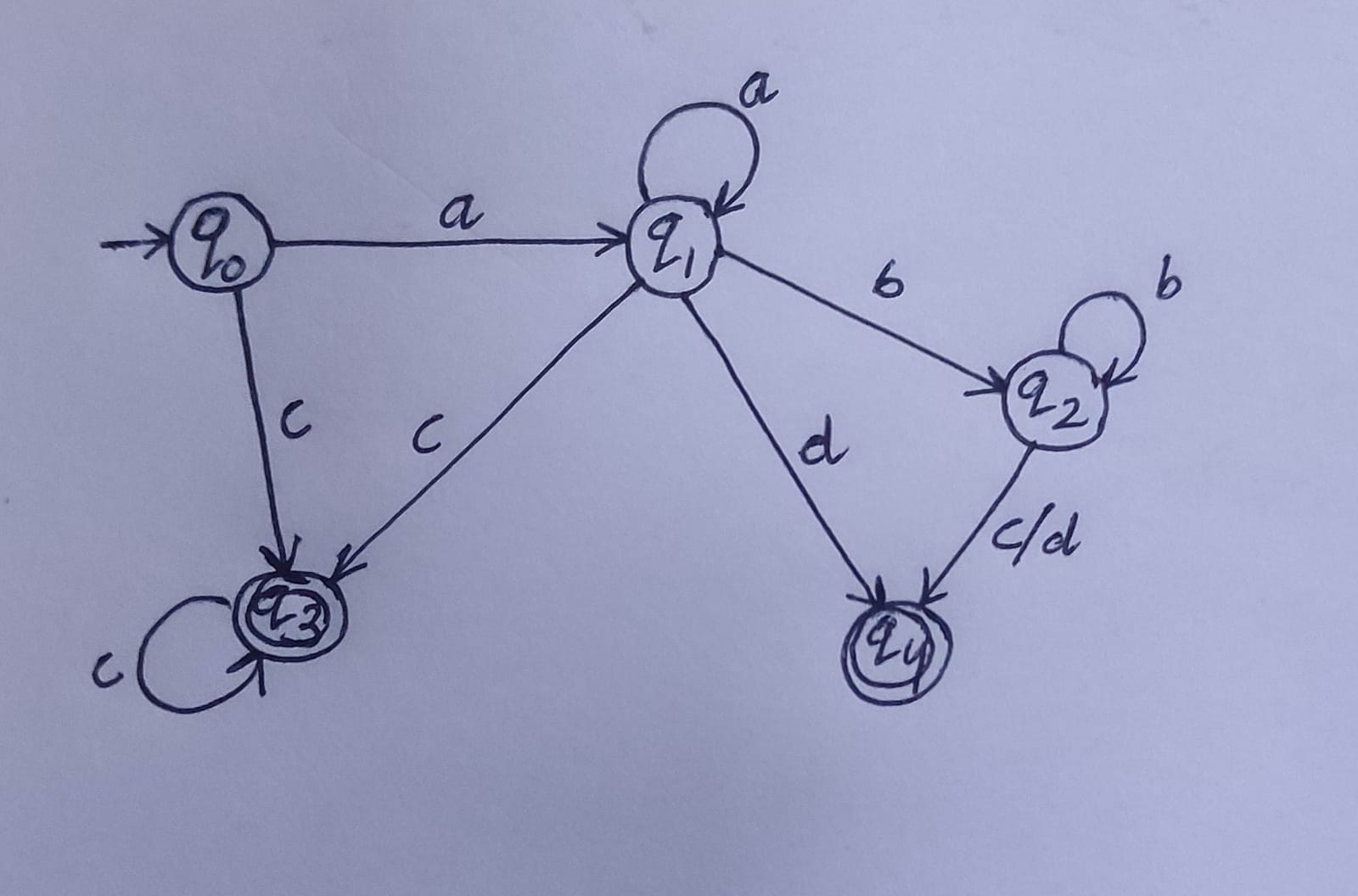


**LAB-2**

**AIM:** To construct a Deterministic Finite Automata for Regular Expression.

**THEORY**

* DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time.
* In DFA, there is only one path for specific input from the current state to the next state.
* DFA does not accept the null move, i.e., the DFA cannot change state without any input character.
* DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.



**Regular Expression 1**

**Source code**

#include <iostream>

#include <bits/stdc++.h>

#include <string>

using namespace std;

int main()

{

    string expr;

    cout << "Enter the expression: "<<endl;

    cin >> expr;

    string states = "q0";

    for (int i = 0; i < expr.length(); i++)

    {

        if (states == "q0")

        {

            if (expr[i] == 'a')

            {

                states = "q1";

            }

            else if (expr[i] == 'c')

            {

                states = "q3";

            }

            else

            {

                states = "-1";

            }

        }

        else if (states == "q1")

        {

            if (expr[i] == 'a')

                states = "q1";

            else if (expr[i] == 'b')

                states = "q2";

            else if (expr[i] == 'c')

                states = "q3";

            else if (expr[i] == 'd')

                states = "q4";

            else

                states = "-1";

        }

        else if (states == "q2")

        {

            if (expr[i] == 'b')

                states = "q2";

            else if (expr[i] == 'c' || expr[i] == 'd')

                states = "q4";

            else

                states = "-1";

        }

        else if (states == "q3")

        {

            if (expr[i] == 'c')

                states = "q3";

            else if (expr[i] == 'a' || expr[i] == 'b' || expr[i] == 'd')

                states = "-1";

        }

        else if (states == "q4")

        {

            if (expr[i] == 'a' || expr[i] == 'b' || expr[i] == 'c' || expr[i] == 'd')

                states = "-1";

            else

                states = "q4";

        }

        else

            break;

    }

    if ((states == "q4" || states == "q3"))

        cout << "\nThe input expression is Accepted"<<endl;

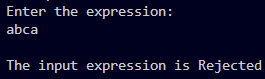
    else

        cout << "\nThe input expression is Rejected"<<endl;

    return 0;

}

**Output**

****

**Regular Expression 2:** 0\*1+(0+1)

**Source Code**

#include <bits/stdc++.h>

#include <string>

using namespace std;

bool match(string str) {

    int state = 0;

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

        char c = str[i];

        switch (state) {

            case 0:

                if (c == '0')

                    state = 0;

                else if (c == '1')

                    state = 1;

                else

                    return false;

                break;

            case 1:

                if (c == '0')

                    state = 2;

                else if (c == '1')

                    state = 3;

                else

                    return false;

                break;

            case 2:

                if(c=='\n')

                    return true;

                else

                    return false;

                break;

            case 3:

                if (c == '0')

                    state = 2;

                else if (c == '1')

                    state = 3;

                else

                    return false;

                break;

       }

    }

        return true;

}

int main() {

    string str;

    cout << "Enter a regular expression: ";

    cin >> str;

    if (match(str)) {

        cout << "String matches the pattern 0\*1+(0+1)" << endl;

    } else {

        cout << "String does not match the pattern 0\*1+(0+1)" << endl;

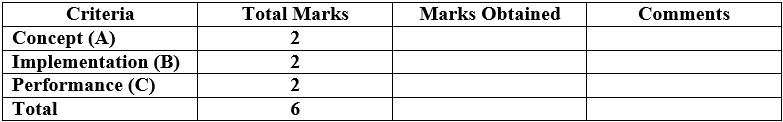
    }

    return 0;

}

**Output**



****

**LAB-3**

**AIM:** To count the number of tokens in a program.

**Source Code**

#include <iostream>

#include <fstream>

#include <regex>

using namespace std;

int main()

{

string filename;

cout << "Enter file name: ";

getline(cin, filename);

ifstream file(filename);

string line;

int token\_count = 0;

if (file.is\_open())

{

while (getline(file, line))

{

if (line.find("#include") != string::npos)

{

continue;

}

if (line[0] == '#')

{

continue;

}

if (line.find("//") != string::npos)

{

continue;

}

if (line.find("/\*") != string::npos)

{

continue;

}

if (line.find("\*/") != string::npos)

{

continue;

}

if (line.find("::") != string::npos)

{

token\_count--;

}

if (line.find(">>") != string::npos)

{

token\_count--;

}

if (line.find("<<") != string::npos)

{

token\_count--;

}

if (line.find("->") != string::npos)

{

token\_count--;

}

if (line.find("++") != string::npos)

{

token\_count--;

}

if (line.find("--") != string::npos)

{

token\_count--;

}

if (line.find("==") != string::npos)

{

token\_count--;

}

if (line.find("!=") != string::npos)

{

token\_count--;

}

if (line.find("<=") != string::npos)

{

token\_count--;

}

if (line.find(">=") != string::npos)

{

token\_count--;

}

if (line.find("&&") != string::npos)

{

token\_count--;

}

if (line.find("||") != string::npos)

{

token\_count--;

}

if (line.find("+=") != string::npos)

{

token\_count--;

}

if (line.find("-=") != string::npos)

{

token\_count--;

}

if (line.find("\*=") != string::npos)

{

token\_count--;

}

if (line.find("/=") != string::npos)

{

token\_count--;

}

if (line.find("%=") != string::npos)

{

token\_count--;

}

if (line.find("&=") != string::npos)

{

token\_count--;

}

if (line.find("|=") != string::npos)

{

token\_count--;

}

if (line.find("^=") != string::npos)

{

token\_count--;

}

if (line.find("<<=") != string::npos)

{

token\_count--;

}

if (line.find(">>=") != string::npos)

{

token\_count--;

}

regex token\_regex("[a-zA-Z\_][a-zA-Z\_0-9]\*|[0-9]+|[(){};,::]|[+\\-\*/=<>]|[^ \t\r\n]]");

sregex\_iterator iter(line.begin(), line.end(), token\_regex);

sregex\_iterator end;

for (; iter != end; ++iter)

{

string token = iter->str();

token\_count++;

}

}

file.close();

cout << "Number of tokens: " << token\_count << endl;

}

else

{

cout << "Unable to open file" << endl;

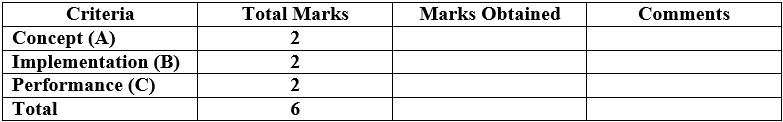
}

return 0;

}

**Output**

****

****

**LAB-4**

**AIM:** To count the number and type of tokens in a program.

**Source Code**

#include <iostream>

#include <string.h>

#include <fstream>

#include<sstream>

using namespace std;

int flag0=0,flag1=0,flag2=0,flag3=0;

bool isValidDelimiter(char ch) {

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

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

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

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

return (true);

return (false);

}

bool isValidOperator(char ch){

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

ch == '/' ||

ch == '='|| ch == '!' || ch == '&' || ch == '|'|| ch == '%'

|| ch == '^' || ch == '~' || ch == '?'|| ch == ':'|| ch == '.'

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

|| ch == '<' || ch == '>' || ch == ',' || ch == ';')

return (true);

return (false);

}

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

bool isvalidIdentifier(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' || isValidDelimiter(str[0]) == true)

return (false);

return (true);

}

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

bool isKeyword(char\* str){

if (!strcmp(str, "asm") || !strcmp(str, "case") || !strcmp(str, "const") || !strcmp(str, "continue")

|| !strcmp(str, "default") || !strcmp(str, "do") || !strcmp(str, "else") || !strcmp(str, "enum")

|| !strcmp(str, "extern") || !strcmp(str, "for") || !strcmp(str, "goto") || !strcmp(str, "if")

|| !strcmp(str, "register") || !strcmp(str, "return") || !strcmp(str, "sizeof") || !strcmp(str, "static")

|| !strcmp(str, "struct") || !strcmp(str, "switch") || !strcmp(str, "typedef") || !strcmp(str, "union")

|| !strcmp(str, "volatile") || !strcmp(str, "while") || !strcmp(str, "auto") || !strcmp(str, "break")

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

|| !strcmp(str, "long") || !strcmp(str, "short") || !strcmp(str, "signed") || !strcmp(str, "unsigned")

|| !strcmp(str, "void") || !strcmp(str, "bool") || !strcmp(str, "catch") || !strcmp(str, "class")

|| !strcmp(str, "const\_cast") || !strcmp(str, "delete") || !strcmp(str, "dynamic\_cast")

|| !strcmp(str, "explicit") || !strcmp(str, "false") || !strcmp(str, "friend") || !strcmp(str, "inline")

|| !strcmp(str, "mutable") || !strcmp(str, "namespace") || !strcmp(str, "new") || !strcmp(str, "operator")

|| !strcmp(str, "private") || !strcmp(str, "protected") || !strcmp(str, "public") || !strcmp(str, "reinterpret\_cast")

|| !strcmp(str, "static\_cast") || !strcmp(str, "template") || !strcmp(str, "this") || !strcmp(str, "throw")

|| !strcmp(str, "true") || !strcmp(str, "try") || !strcmp(str, "typeid") || !strcmp(str, "typename")

|| !strcmp(str, "using") || !strcmp(str, "virtual") || !strcmp(str, "wchar\_t") || !strcmp(str, "and")

|| !strcmp(str, "and\_eq") || !strcmp(str, "bitand") || !strcmp(str, "bitor") || !strcmp(str, "compl")

|| !strcmp(str, "not") || !strcmp(str, "not\_eq") || !strcmp(str, "or") || !strcmp(str, "or\_eq")

|| !strcmp(str, "xor") || !strcmp(str, "xor\_eq") || !strcmp(str, "alignas") || !strcmp(str, "alignof")

|| !strcmp(str, "char16\_t") || !strcmp(str, "char32\_t") || !strcmp(str, "constexpr") || !strcmp(str, "decltype")

|| !strcmp(str, "noexcept") || !strcmp(str, "nullptr") || !strcmp(str, "static\_assert") || !strcmp(str, "thread\_local")

|| !strcmp(str, "export") || !strcmp(str,"main"))

return (true);

return (false);

}

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

bool isValidInteger(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);

}

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);

}

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);

}

void detectTokens(char \*str) {

int left = 0, right = 0;

int length = strlen(str);

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

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

right++;

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

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

flag0++;

right++;

left = right;

} else if (isValidDelimiter(str[right]) == true && left != right || (right == length && left != right)) {

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

if (isKeyword(subStr) == true)

flag1++;

else if (isValidInteger(subStr) == true)

flag2++;

else if (isRealNumber(subStr) == true)

flag2++;

else if (isvalidIdentifier(subStr) == true

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

flag3++;

else if (isvalidIdentifier(subStr) == false

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

flag3++;

left = right;

}

}

return;

}

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

char str[100] ="int main() { int a = 10; int b = 20; int c = a + b; cout << c; return 0; }";

printf("The Program is : '%s' ", str);

printf("\n\nAll Tokens are : ");

detectTokens(str);

printf("\nKeywords : %d", flag1);

printf("\nIdentifiers : %d", flag3);

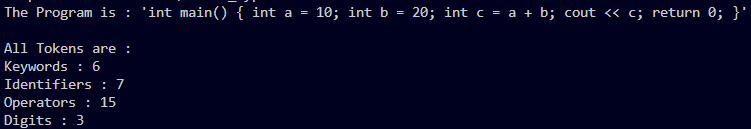
printf("\nOperators : %d", flag0);

printf("\nDigits : %d", flag2);

return (0);

}

**Output**



**Table

Description automatically generated**

**LAB-5**

**AIM:** To remove the Left Recursion from a given grammar.

**Source Code**

#include <iostream>

#include <string>

using namespace std;

int main()

{

string ip, op1, op2, temp;

int sizes[10] = {};

char c;

int n, j, l;

cout << "Enter the Parent Non-Terminal : ";

cin >> c;

ip.push\_back(c);

op1 += ip + "\'->";

ip += "->";

op2 += ip;

cout << "Enter the number of productions : ";

cin >> n;

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

{

cout << "Enter Production " << i + 1 << " : ";

cin >> temp;

sizes[i] = temp.size();

ip += temp;

if (i != n - 1)

ip += "|";

}

cout << "Production Rule : " << ip << endl;

for (int i = 0, k = 3; i < n; i++)

{

if (ip[0] == ip[k])

{

cout << "Production " << i + 1 << " has left recursion." << endl;

if (ip[k] != '#')

{

for (l = k + 1; l < k + sizes[i]; l++)

op1.push\_back(ip[l]);

k = l + 1;

op1.push\_back(ip[0]);

op1 += "\'|";

}

}

else

{

cout << "Production " << i + 1 << " does not have left recursion." << endl;

if (ip[k] != '#')

{

for (j = k; j < k + sizes[i]; j++)

op2.push\_back(ip[j]);

k = j + 1;

op2.push\_back(ip[0]);

op2 += "\'|";

}

else

{

op2.push\_back(ip[0]);

op2 += "\'";

}

}

}

op1 += "#";

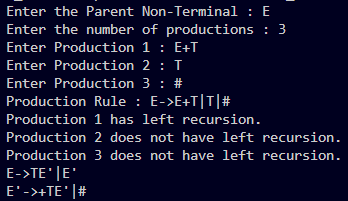
cout << op2 << endl;

cout << op1 << endl;

return 0;

}

**Output**



**Table

Description automatically generated**

**LAB-6**

**AIM:** To remove ambiguity from a grammar.

**Source Code**

#include <stdio.h>

#include <string.h>

int main()

{

char a[20];

int i, t = 0;

int x;

printf("Enter no of ambiguous productions: ");

scanf("%d", &x);

while(x!=0)

{

printf("Enter Ambiguous Production: E->");

scanf("%s",a);

char b[5] = {'E', 'P', 'Q', 'R', 'S'};

for (i = 0; a[i] != '\0';) {

if (a[i] == '|')

{

i++;

}

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

{

printf("%c->id", b[t]);

i = i + 2;

}

else

{

if (a[i + 1] == '^')

{

printf("%c->%c^%c|%c\n", b[t], b[t + 1], b[t], b[t + 1]);

t++;

i = i + 3;

}

else

{

printf("%c->%c%c%c|%c\n", b[t], b[t], a[i + 1], b[t + 1], b[t + 1]);

t++;

i = i + 3;

}

}

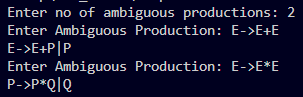
}

x--;

}

}

**Output**



**Table

Description automatically generated**

**LAB-7**

**AIM:** To construct predictive parser.

**Source Code**

#include <stdio.h>

#include <conio.h>

#include <string.h>

int main()

{

char fin[10][20], st[10][20], ft[20][20], fol[20][20];

int a = 0, e, i, t, b, c, n, k, l = 0, j, s, m, p;

printf("enter the no. of productions\n");

scanf("%d", &n);

printf("enter the productions in a grammar\n");

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

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

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

fol[i][0] = '\0';

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

{

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

{

j = 3;

l = 0;

a = 0;

l1:

if (!((st[i][j] > 64) && (st[i][j] < 91)))

{

for (m = 0; m < l; m++)

{

if (ft[i][m] == st[i][j])

goto s1;

}

ft[i][l] = st[i][j];

l = l + 1;

s1:

j = j + 1;

}

else

{

if (s > 0)

{

while (st[i][j] != st[a][0])

{

a++;

}

b = 0;

while (ft[a][b] != '\0')

{

for (m = 0; m < l; m++)

{

if (ft[i][m] == ft[a][b])

goto s2;

}

ft[i][l] = ft[a][b];

l = l + 1;

s2:

b = b + 1;

}

}

}

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

{

if (st[i][j] == '|')

{

j = j + 1;

goto l1;

}

j = j + 1;

}

ft[i][l] = '\0';

}

}

printf("first pos\n");

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

printf("FIRS[%c]=%s\n", st[i][0], ft[i]);

fol[0][0] = '$';

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

{

k = 0;

j = 3;

if (i == 0)

l = 1;

else

l = 0;

k1:

while ((st[i][0] != st[k][j]) && (k < n))

{

if (st[k][j] == '\0')

{

k++;

j = 2;

}

j++;

}

j = j + 1;

if (st[i][0] == st[k][j - 1])

{

if ((st[k][j] != '|') && (st[k][j] != '\0'))

{

a = 0;

if (!((st[k][j] > 64) && (st[k][j] < 91)))

{

for (m = 0; m < l; m++)

{

if (fol[i][m] == st[k][j])

goto q3;

}

fol[i][l] = st[k][j];

l++;

q3:

p++;

}

else

{

while (st[k][j] != st[a][0])

{

a++;

}

p = 0;

while (ft[a][p] != '\0')

{

if (ft[a][p] != 'e')

{

for (m = 0; m < l; m++)

{

if (fol[i][m] == ft[a][p])

goto q2;

}

fol[i][l] = ft[a][p];

l = l + 1;

}

else

e = 1;

q2:

p++;

}

if (e == 1)

{

e = 0;

goto a1;

}

}

}

else

{

a1:

c = 0;

a = 0;

while (st[k][0] != st[a][0])

{

a++;

}

while ((fol[a][c] != '\0') && (st[a][0] != st[i][0]))

{

for (m = 0; m < l; m++)

{

if (fol[i][m] == fol[a][c])

goto q1;

}

fol[i][l] = fol[a][c];

l++;

q1:

c++;

}

}

goto k1;

}

fol[i][l] = '\0';

}

printf("follow pos\n");

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

printf("FOLLOW[%c]=%s\n", st[i][0], fol[i]);

printf("\n");

s = 0;

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

{

j = 3;

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

{

if ((st[i][j - 1] == '|') || (j == 3))

{

for (p = 0; p <= 2; p++)

{

fin[s][p] = st[i][p];

}

t = j;

for (p = 3; ((st[i][j] != '|') && (st[i][j] != '\0')); p++)

{

fin[s][p] = st[i][j];

j++;

}

fin[s][p] = '\0';

if (st[i][t] == 'e')

{

b = 0;

a = 0;

while (st[a][0] != st[i][0])

{

a++;

}

while (fol[a][b] != '\0')

{

printf("M[%c,%c]=%s\n", st[i][0], fol[a][b], fin[s]);

b++;

}

}

else if (!((st[i][t] > 64) && (st[i][t] < 91)))

printf("M[%c,%c]=%s\n", st[i][0], st[i][t], fin[s]);

else

{

b = 0;

a = 0;

while (st[a][0] != st[i][3])

{

a++;

}

while (ft[a][b] != '\0')

{

printf("M[%c,%c]=%s\n", st[i][0], ft[a][b], fin[s]);

b++;

}

}

s++;

}

if (st[i][j] == '|')

j++;

}

}

return 0;

}

**Output**

****

****

**Text

Description automatically generated with low confidence**

**Text

Description automatically generated**

**A picture containing table

Description automatically generated**

**Table

Description automatically generated**

**LAB-**

**AIM:** To convert a given infix expression to postfix and evaluate that postfix expression

**Source Code**

#include <stdio.h>

#include <conio.h>

#include <string.h>

int main()

{

char fin[10][20], st[10][20], ft[20][20], fol[20][20];

int a = 0, e, i, t, b, c, n, k, l = 0, j, s, m, p;

printf("enter the no. of productions\n");

scanf("%d", &n);

printf("enter the productions in a grammar\n");

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

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

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

fol[i][0] = '\0';

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

{

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

{

j = 3;

l = 0;

a = 0;

l1:

if (!((st[i][j] > 64) && (st[i][j] < 91)))

{

for (m = 0; m < l; m++)

{

if (ft[i][m] == st[i][j])

goto s1;

}

ft[i][l] = st[i][j];

l = l + 1;

s1:

j = j + 1;

}

else

{

if (s > 0)

{

while (st[i][j] != st[a][0])

{a++; }

b = 0;

while (ft[a][b] != '\0')

{

for (m = 0; m < l; m++)

{

if (ft[i][m] == ft[a][b])

goto s2;

}

ft[i][l] = ft[a][b];

l = l + 1;

s2:

b = b + 1;

}

}

}

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

{

if (st[i][j] == '|')

{

j = j + 1;

goto l1;

}

j = j + 1;

}

ft[i][l] = '\0';

}

}

printf("first pos\n");

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

printf("FIRS[%c]=%s\n", st[i][0], ft[i]);

fol[0][0] = '$';

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

{

k = 0;

j = 3;

if (i == 0)

l = 1;

else

l = 0;

k1:

while ((st[i][0] != st[k][j]) && (k < n))

{

if (st[k][j] == '\0')

{

k++;

j = 2;

}

j++;

}

j = j + 1;

if (st[i][0] == st[k][j - 1])

{

if ((st[k][j] != '|') && (st[k][j] != '\0'))

{

a = 0;

if (!((st[k][j] > 64) && (st[k][j] < 91)))

{

for (m = 0; m < l; m++)

{

if (fol[i][m] == st[k][j])

goto q3;

}

fol[i][l] = st[k][j];

l++;

q3:

p++; }

else

{

while (st[k][j] != st[a][0])

{

a++;

}

p = 0;

while (ft[a][p] != '\0')

{

if (ft[a][p] != 'e')

{

for (m = 0; m < l; m++)

{

if (fol[i][m] == ft[a][p])

goto q2;

}

fol[i][l] = ft[a][p];

l = l + 1;

}

else

e = 1;

q2:

p++;

}

if (e == 1)

{

e = 0;

goto a1;

}

}

}

else

{

a1:

c = 0;

a = 0;

while (st[k][0] != st[a][0])

{

a++;

}

while ((fol[a][c] != '\0') && (st[a][0] != st[i][0]))

{

for (m = 0; m < l; m++)

{

if (fol[i][m] == fol[a][c])

goto q1;

}

fol[i][l] = fol[a][c];

l++;

q1:

c++;

}

}

goto k1;

}

fol[i][l] = '\0';

}

printf("follow pos\n");

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

printf("FOLLOW[%c]=%s\n", st[i][0], fol[i]);

printf("\n");

return 0;

}

**Output**

Text

Description automatically generated**Table

Description automatically generated**

**LAB-**

**AIM:** To compute Leading and Trailing for a grammar.

**Source Code**

#include <iostream>

#include <string.h>

#include <conio.h>

using namespace std;

int nt, t, top = 0;

char s[50], NT[10], T[10], st[50], l[10][10], tr[50][50];

int searchnt(char a)

{

int count = -1, i;

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

{

if (NT[i] == a)

return i;

}

return count;

}

int searchter(char a)

{

int count = -1, i;

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

{

if (T[i] == a)

return i;

}

return count;

}

void push(char a)

{

s[top] = a;

top++;

}

char pop()

{

top--;

return s[top];

}

void installl(int a, int b)

{

if (l[a][b] == 'f')

{

l[a][b] = 't';

push(T[b]);

push(NT[a]);

}

}

void installt(int a, int b)

{

if (tr[a][b] == 'f')

{

tr[a][b] = 't';

push(T[b]);

push(NT[a]);

}

}

int main()

{

int i, s, k, j, n;

char pr[30][30], b, c;

// clrscr();

cout << "Enter the no of productions:";

cin >> n;

cout << "Enter the productions one by one\n";

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

cin >> pr[i];

nt = 0;

t = 0;

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

{

if ((searchnt(pr[i][0])) == -1)

NT[nt++] = pr[i][0];

}

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

{

for (j = 3; j < strlen(pr[i]); j++)

{

if (searchnt(pr[i][j]) == -1)

{

if (searchter(pr[i][j]) == -1)

T[t++] = pr[i][j];

}

}

}

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

{

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

l[i][j] = 'f';

}

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

{

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

tr[i][j] = 'f';

}

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

{

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

{

if (NT[(searchnt(pr[j][0]))] == NT[i])

{

if (searchter(pr[j][3]) != -1)

installl(searchnt(pr[j][0]), searchter(pr[j][3]));

else

{

for (k = 3; k < strlen(pr[j]); k++)

{

if (searchnt(pr[j][k]) == -1)

{

installl(searchnt(pr[j][0]), searchter(pr[j][k]));

break;

}}}}}}

while (top != 0)

{

b = pop();

c = pop();

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

{

if (pr[s][3] == b)

installl(searchnt(pr[s][0]), searchter(c));

} }

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

{

cout << "Leading[" << NT[i] << "]"

<< "\t{";

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

{

if (l[i][j] == 't')

cout << T[j] << ",";

}

cout << "}\n";

}

top = 0;

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

{

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

{

if (NT[searchnt(pr[j][0])] == NT[i])

{

if (searchter(pr[j][strlen(pr[j]) - 1]) != -1)

installt(searchnt(pr[j][0]), searchter(pr[j][strlen(pr[j]) - 1]));

else

{

for (k = (strlen(pr[j]) - 1); k >= 3; k--)

{

if (searchnt(pr[j][k]) == -1)

{

installt(searchnt(pr[j][0]), searchter(pr[j][k]));

break;

}}}}}}

while (top != 0)

{

b = pop();

c = pop();

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

{

if (pr[s][3] == b)

installt(searchnt(pr[s][0]), searchter(c));

} }

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

{

cout << "Trailing[" << NT[i] << "]"

<< "\t{";

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

{

if (tr[i][j] == 't')

cout << T[j] << ",";

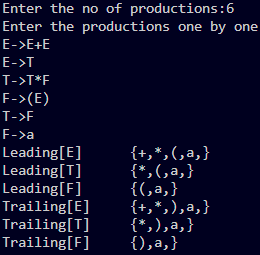
}

cout << "}\n";

}

getch();}

**Output**

**Table

Description automatically generated**