Saket Kumar Baranwal (RA1911003010414)

**COMPILER DESIGN LAB**

**EXP 4 -ELIMINATION OF LEFT RECURSION AND LEFT FACTORING**

**AIM :**To create a program which identifies the given grammar as left recursive or not and then perform elimination of the left recursion and left factoring

**THEORY :**

**Left Recursive Grammar**:

● A production of grammar is said to have left recursion if the leftmost variable of its RHS is the same as the variable of its LHS.

● A grammar containing a production having left recursion is called as Left Recursive Grammar

● Left recursion is considered to be a problematic situation for Top down parsers.

● Therefore, left recursion has to be eliminated from the grammar.

**Elimination of Left Recursion**

Left recursion is eliminated by converting the grammar into a right recursive grammar.

If we have the left-recursive pair of productions-

**A** → **Aα / β**

(Left Recursive Grammar)

where β does not begin with an A.

Then, we can eliminate left recursion by replacing the pair of productions with-

**A** → **βA’**

**A’** → **αA’ / ∈**

(Right Recursive Grammar)

This right recursive grammar functions the same as left recursive grammar.

**Left Factoring :**

Left factoring is used so that it makes the grammar useful for top-down parsers. Here we take out the left factor that appears in 2 productions of the same non-terminal. So, it is done to avoid backtracking by the parser.

**ALGORITHM :**

1. We declare d,a,b strings and a flag bit

2. Store the parent non-terminal in c var

3. Push c into d string such that E-> left production

4. Take the no of productions in. For example take 2

5. And for loop till iterator = 2

6. And take each production and add it to d string var such that after each production it will be E->E+T|

To detect if the production is left recursive or not:

7. If(d[0] != d[k]) it doesn’t have left recursion because E->E here k is the last var E and for left recursion the left variable should be the parent non-terminal.

8. If they have left recursion then push it to A vector array according to the laws of left recursion ie TE’ where E’->+TE’|# AND E->TE’

9. Print the two

**SOURCE CODE :**

**4a - DETECTION AND ELIMINATION OF LEFT RECURSIVE GRAMMAR :**

#include <iostream>

#include <string>

using namespace std;

int main()

{

int n, j, l, i, k;

int length[10] = {};

string d, a, b, flag;

char c;

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

cin >> c;

d.push\_back(c);

a += d + "\'->";

d += "->";

b += d;

cout<<"Enter productions: ";

cin >> n;

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

{

cout<<"Enter Production ";

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

cin >> flag;

length[i] = flag.size();

d += flag;

if (i != n - 1)

{

d += "|";

}

}

cout<<"The Production Rule is: ";

cout<<d<<endl;

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

{

if (d[0] != d[k])

{

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

cout<<" does not have left recursion.";

cout<<endl;

if (d[k] == '#')

{

b.push\_back(d[0]);

b += "\'";

}

else

{

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

{

b.push\_back(d[j]);

}

k = j + 1;

b.push\_back(d[0]);

b += "\'|";

}

}

else

{

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

cout<< " has left recursion";

cout<< endl;

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

{

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

{

a.push\_back(d[l]);

}

k = l + 1;

a.push\_back(d[0]);

a += "\'|";

}

}

}

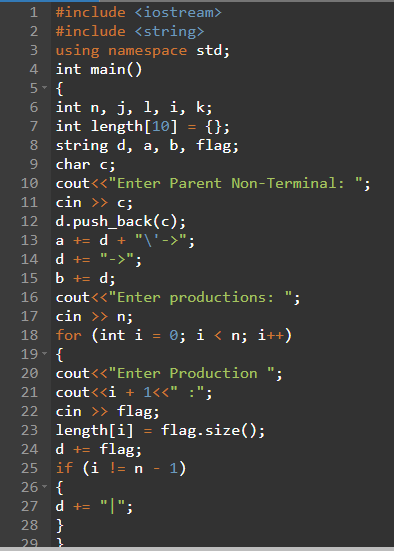
a += "#";

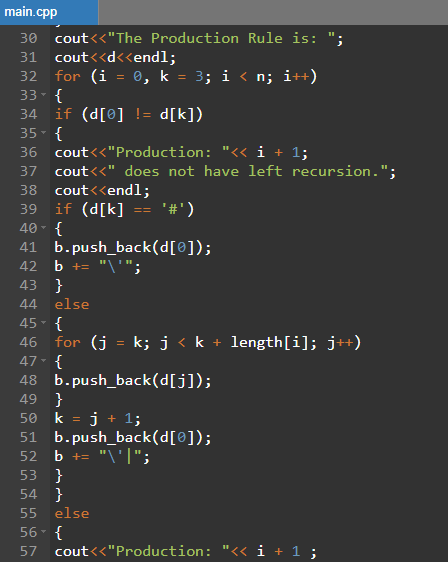
cout << b << endl;

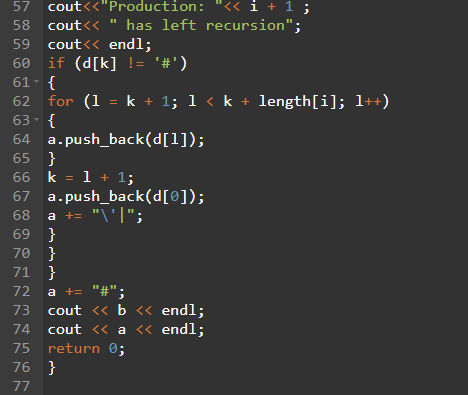
cout << a << endl;

return 0;

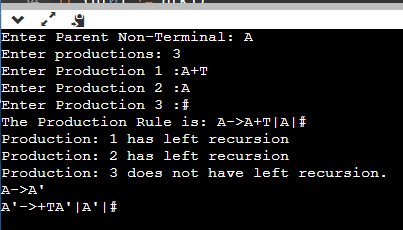
}







**Output:-**



**4b - LEFT FACTORING ELIMINATION**

#include <iostream>

#include <string>

using namespace std;

int main()

{

int n,j,l,i,m;

int len[10] = {};

string a, b1, b2, flag;

char c;

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

cin >> c;

a.push\_back(c);

b1 += a + "\'->";

b2 += a + "\'\'->";;

a += "->";

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

cin >> n;

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

{

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

cin >> flag;

len[i] = flag.size();

a += flag;

if (i != n - 1)

{

a += "|";

}

}

cout << "The Production Rule is : " << a << endl;

char x = a[3];

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

{

if (x != a[m])

{

while (a[m++] != '|');

}

else

{

if (a[m + 1] != '|')

{

b1 += "|" + a.substr(m + 1, len[i] - 1);

a.erase(m - 1, len[i] + 1);

}

else

{

b1 += "#";

a.insert(m + 1, 1, a[0]);

a.insert(m + 2, 1, '\'');

m += 4;

}

}

}

char y = b1[6];

for (i = 0, m = 6; i < n - 1; i++)

{

if (y == b1[m])

{

if (b1[m + 1] != '|')

{

flag.clear();

for (int s = m + 1; s < b1.length(); s++)

{

flag.push\_back(b1[s]);

}

b2 += "|" + flag;

b1.erase(m - 1, flag.length() + 2);

}

else

{

b1.insert(m + 1, 1, b1[0]);

b1.insert(m + 2, 2, '\'');

b2 += "#";

m += 5;

}

}

}

b2.erase(b2.size() - 1);

cout << "After Left Factoring : " << endl;

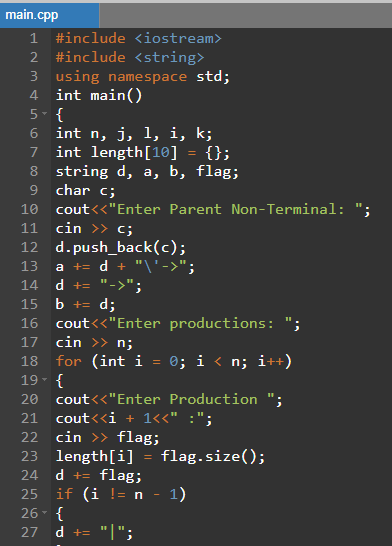
cout << a << endl;

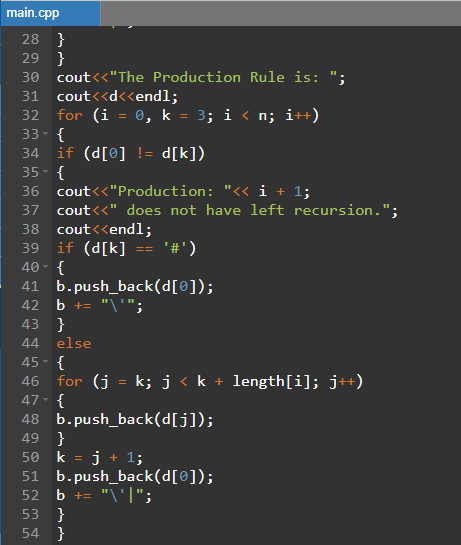
cout << b1 << endl;

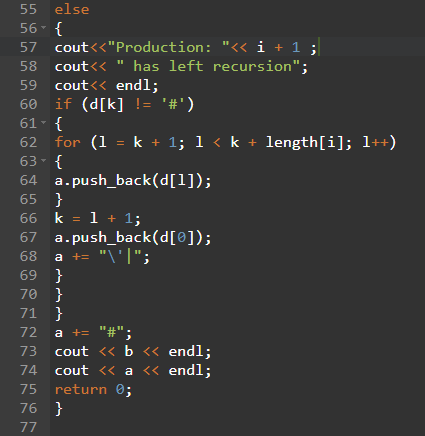
cout << b2 << endl;

return 0;

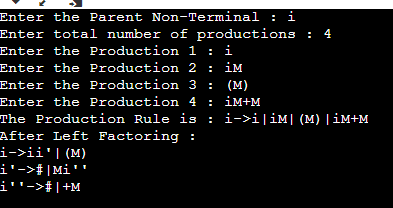
}







**Output:-**



**OBSERVATION :**

The grammars given as input were accurately detected as left recursive or not . Those identified as left recursive underwent left recursion elimination and left factoring elimination .The output was verified.

**RESULT :**

Thus we have successfully implemented a program to detect and eliminate left recursive grammar.