Compiler Design Lab

Date: 9-1-2026 **Name:** S Vishwajith
Assignment No.: 5 **Register No.:** 23BCE1145

Aim:

To write a C++ program that parses the given LL (1) grammar: E->E+T/T; T->T*F/F; F->(E)/id; and prints the first(), follow(), parse table and the stack trace, and check if the string “(id+id)*id” is accepted by the grammar or not.

Algorithm:

1. Eliminate left recursion present in the given grammar.
2. Find first() for all non-terminals.
3. Using first(), find follow() for all non-terminals.
4. Using first() and follow(), construct the parse table.
5. Using this parse table, check whether the string can be accepted or not, and print the trace for each step.

Code:

```
#include <iostream>

#include <vector>

#include <string>

#include <map>

#include <set>

#include <iomanip>

using namespace std;


vector<pair<char, string>> get_grammar()

{

   return {

      {'E', "TA"},
```

```
        {'A', "+TA"},
        {'A', "e"},
        {'T', "FB"},
        {'B', "*FB"},
        {'B', "e"},
        {'F', "(E)"},
        {'F', "id"}};
}

map<char, set<string>> FIRST, FOLLOW;
map<char, map<string, string>> TABLE;

bool isNonTerminal(char c)
{
    return c >= 'A' && c <= 'Z';
}

void computeFIRST(vector<pair<char, string>> &G)
{
    bool changed = true;
    while (changed)
    {
        changed = false;
        for (auto &prod : G)
        {
            char A = prod.first;
            string rhs = prod.second;
```

```
if (rhs == "e")
{
   if (FIRST[A].insert("e").second)
      changed = true;
   continue;
}

for (int i = 0; i < rhs.length(); i++)
{
   char X = rhs[i];

   if (isNonTerminal(X))
   {
      for (auto t : FIRST[X])
         if (t != "e")
            if (FIRST[A].insert(t).second)
               changed = true;

      if (FIRST[X].count("e") == 0)
         break;
   }
   else
   {
      if (X == 'i')
         FIRST[A].insert("id");
      else
         FIRST[A].insert(string(1, X));
      break;
```

```
            }
        }
    }
}

void computeFOLLOW(vector<pair<char, string>> &G)
{
    FOLLOW['E'].insert("$");

    bool changed = true;
    while (changed)
    {
        changed = false;
        for (auto &prod : G)
        {
            char A = prod.first;
            string rhs = prod.second;

            for (int i = 0; i < rhs.length(); i++)
            {
                char B = rhs[i];
                if (!isNonTerminal(B))
                    continue;

                bool epsilonTrail = true;

                for (int j = i + 1; j < rhs.length(); j++)
```

```
{
   epsilonTrail = false;
   char C = rhs[j];

   if (isNonTerminal(C))
   {
      for (auto t : FIRST[C])
         if (t != "e")
            if (FOLLOW[B].insert(t).second)
               changed = true;

      if (FIRST[C].count("e"))
         epsilonTrail = true;
      else
         break;
   }
   else
   {
      if (C == 'i')
         FOLLOW[B].insert("id");
      else
         FOLLOW[B].insert(string(1, C));
      break;
   }
}

if (epsilonTrail || i == rhs.length() - 1)
   for (auto t : FOLLOW[A])
```

```
                if (FOLLOW[B].insert(t).second)
                    changed = true;
            }
        }
    }
}

set<string> FIRST_of_string(string s)
{
    set<string> result;
    for (int i = 0; i < s.length(); i++)
    {
        char X = s[i];

        if (isNonTerminal(X))
        {
            for (auto t : FIRST[X])
                if (t != "e")
                    result.insert(t);
            if (FIRST[X].count("e") == 0)
                break;
        }
        else
        {
            if (X == 'i')
                result.insert("id");
            else
                result.insert(string(1, X));
```

```
            break;
        }
    }
    return result;
}

void buildParsingTable(vector<pair<char, string>> &G)
{
    for (auto &prod : G)
    {
        char A = prod.first;
        string alpha = prod.second;

        auto f = FIRST_of_string(alpha);

        for (auto t : f)
            TABLE[A][t] = alpha;

        if (f.count("e"))
            for (auto b : FOLLOW[A])
                TABLE[A][b] = alpha;
    }
}

vector<string> tokenize(string s)
{
    vector<string> tokens;
```

```
    for (int i = 0; i < s.length();)
    {
        if (i + 1 < s.length() && s.substr(i, 2) == "id")
        {
            tokens.push_back("id");
            i += 2;
        }
        else
        {
            tokens.push_back(string(1, s[i]));
            i++;
        }
    }
    return tokens;
}

int parse(string input)
{
    input += "$";

    vector<string> stk = {"$", "E"};
    int i = 0;

    cout << left
         << setw(20) << "Stack"
         << setw(20) << "Input"
         << "Action" << endl;
    cout << "------------------------------------------------------------" << endl;
```

```
while (true)
{
   string stackStr;
   for (auto &s : stk)
      stackStr += s;

   cout << left
        << setw(20) << stackStr
        << setw(20) << input.substr(i);

   string X = stk.back();

   if (X == "$" && input[i] == '$')
   {
      cout << "Accept\n";
      return 0;
   }

   string a;
   if (i + 1 < input.length() && input.substr(i, 2) == "id")
      a = "id";
   else
      a = string(1, input[i]);

   if (!isNonTerminal(X[0]))
   {
      if (X == a)
```

```
    {
        stk.pop_back();
        i += (a == "id" ? 2 : 1);
        cout << "Match " << a << "\n";
    }
    else
    {
        cout << "Error\n";
        return 1;
    }
}
else
{
    if (TABLE[X[0]].count(a) == 0)
    {
        cout << "Error\n";
        return 1;
    }

    stk.pop_back();
    string prod = TABLE[X[0]][a];

    if (prod != "e")
    {
        auto tokens = tokenize(prod);
        for (int k = tokens.size() - 1; k >= 0; k--)
            stk.push_back(tokens[k]);
    }
```

```
            cout << X << " --> " << prod << "\n";
        }
    }
}

int main()
{
    auto grammar = get_grammar();

    computeFIRST(grammar);
    computeFOLLOW(grammar);
    buildParsingTable(grammar);

    cout << endl
         << "First():" << endl;
    for (auto &p : FIRST)
    {
        cout << "first(" << p.first << ") = { ";
        for (auto &x : p.second)
            cout << x << " ";
        cout << "}\n";
    }

    cout << endl
         << "Follow():" << endl;
    for (auto &p : FOLLOW)
    {
```

```
        cout << "follow(" << p.first << ") = { ";
        for (auto &x : p.second)
            cout << x << " ";
        cout << "}\n";
    }

    cout << endl
         << "Parsing Table:" << endl;
    for (auto &r : TABLE)
        for (auto &c : r.second)
            cout << "map[" << r.first << "," << c.first << "] = " << r.first << " --> " << c.second <<
endl;

    string input;
    cout << endl
         << "Enter input string: ";
    cin >> input;

    int res = parse(input);
    cout << endl
         << ((res == 0) ? "String is accepted." : "String is rejected.") << endl;
}
```

Input/Output:

First():

first(A) = { + e }

first(B) = { * e }

first(E) = { (id }

first(F) = { (id }

first(T) = { (id }

Follow():

follow(A) = { $) }

follow(B) = { $) + }

follow(E) = { $) }

follow(F) = { $) * + }

follow(T) = { $) + }

Parsing Table:

map[A,$] = A --> e

map[A,)] = A --> e

map[A,+] = A --> +TA

map[A,e] = A --> e

map[B,$] = B --> e

map[B,)] = B --> e

map[B,*] = B --> *FB

map[B,+] = B --> e

map[B,e] = B --> e

map[E,(] = E --> TA

map[E,id] = E --> TA

map[F,(] = F --> (E)

map[F,id] = F --> id

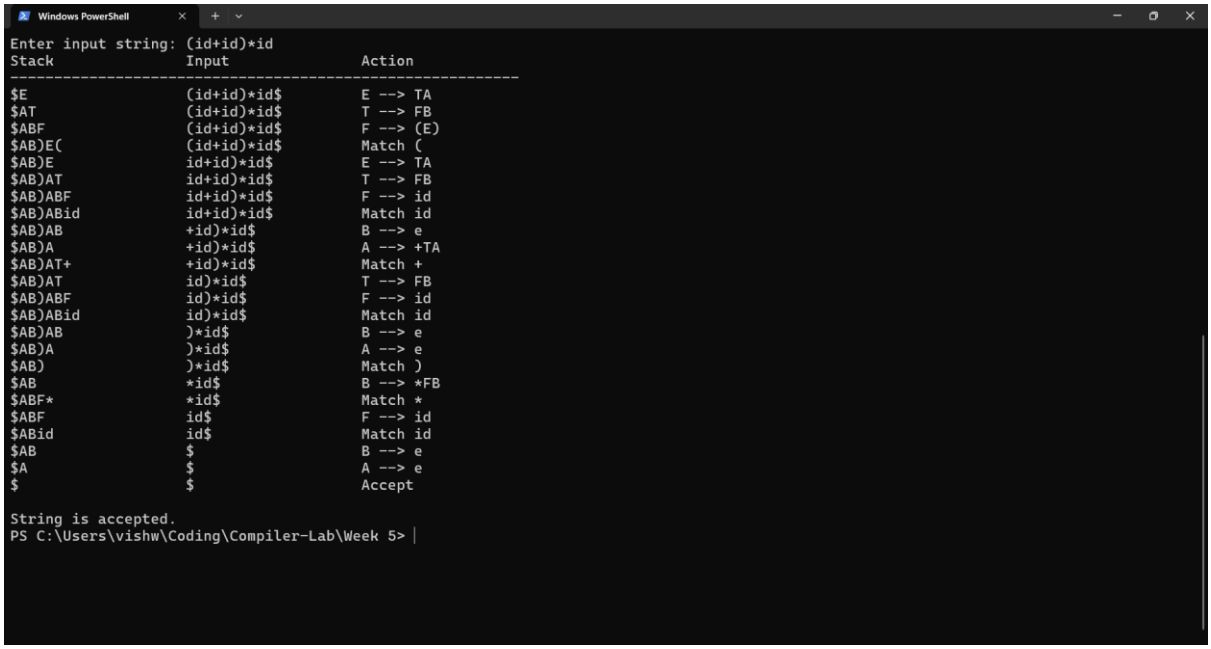
map[T,(] = T --> FB

map[T,id] = T --> FB

Enter input string: (id+id)*id

Stack	Input	Action
$E	(id+id)*id$	E --> TA
$AT	(id+id)*id$	T --> FB
$ABF	(id+id)*id$	F --> (E)
$AB)E(	(id+id)*id$	Match (
$AB)E	id+id)*id$	E --> TA
$AB)AT	id+id)*id$	T --> FB
$AB)ABF	id+id)*id$	F --> id
$AB)ABid	id+id)*id$	Match id
$AB)AB	+id)*id$	B --> e
$AB)A	+id)*id$	A --> +TA
$AB)AT+	+id)*id$	Match +
$AB)AT	id)*id$	T --> FB
$AB)ABF	id)*id$	F --> id
$AB)ABid	id)*id$	Match id
$AB)AB	)*id$	B --> e
$AB)A	)*id$	A --> e
$AB)	)*id$	Match)
$AB	*id$	B --> *FB
$ABF*	*id$	Match *
$ABF	id$	F --> id
$ABid	id$	Match id
$AB	$	B --> e
$A	$	A --> e
$	$	Accept

String is accepted.



Result:

The C++ program that parses the given LL (1) grammar: E->E+T/T; T->T*F/F; F->(E)/id; and prints the first(), follow(), parse table and the stack trace, and check if the string “(id+id)*id” is accepted by the grammar or not was successfully run and the results were verified.