PRACTICAL FILE

BE (CSE) 6th Semester

COMPILER DESIGN (CS654)

March 2021 – May 2021

<u>Submitted By</u>

Jatin Ghai
(UE183042)

<u>Submitted To</u>

Dr. Akashdeep
Assistant Professor, Computer Science and Engineering

Computer Science and Engineering
University Institute of Engineering and Technology
Panjab University, Chandigarh – 160014, INDIA
2021

Experiment No. 5

AIM :- To implement a Predictive Parser for a given Context Free Grammar.

<u>Parsing</u>: Parsing or syntactic analysis is the process of analyzing a string of symbols, conforming to the rules of a given grammar grammar. Parser takes tokens as input and builds a parse tree in order to check whether the given input string is accepted by the given grammar or not.

Parser is mainly classified into 2 categories: Top-down Parser, and Bottom-up Parser.

Predictive Parser is a type of **Top-down Parser**, which generates the input string starting from the start symbol and using Leftmost derivations. It generates the parse tree using the parse table. The grammar used in predictive parsing must not be left recursive and must be left factored. This ensures it doesn't require any backtracking. The grammar is called the LL(1) grammar.

<u>CFG Used:</u>

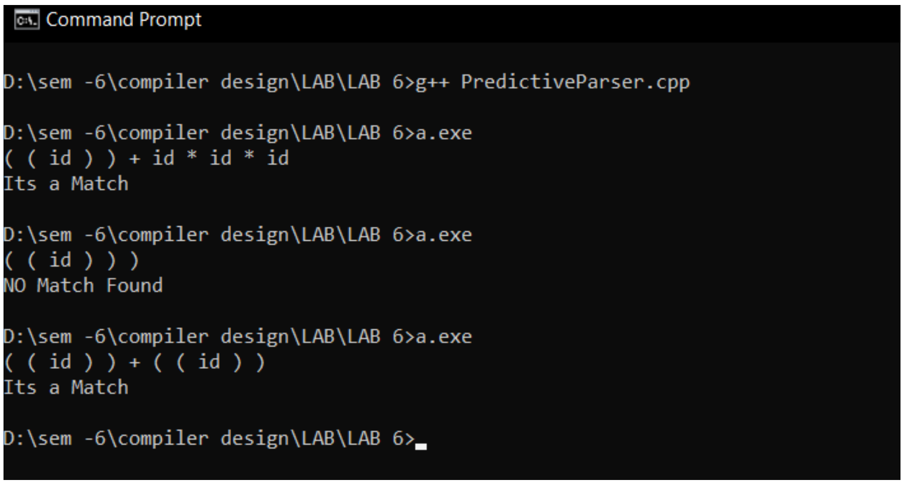
```
// E -> T E′
// E′-> + TE′  | ϵ
// T -> F T′
// T′-> * F T′  | ϵ
// F -> (E)  | id
```

<u>Source Code:</u>

```
#include<bits/stdc++.h>
using namespace std;

// epsilon 0 || E 1 || E' 2 || T 3 || T' 4 || F 5 || + 6 || * 7 || id 8
| ) 9 || ( 10
```

```
void pre(unordered_map<string,unordered_map<string,int>>& table,
vector<vector<int>>& rules,vector<string>& states){

    states = {"epsilon","E","E'","T","T'","F","+","*","id",")","("};

    rules.push_back({2,3});    //E -> TE`      0
    rules.push_back({2,3,6}); //E'-> +TE`     1
    rules.push_back({});       //E'-> epsilon 2
    rules.push_back({4,5});    //T-> FT`       3
    rules.push_back({});       //T`-> epsilon 4
    rules.push_back({4,5,7}); //T`-> *FT`     5
    rules.push_back({8});      //F -> id       6
    rules.push_back({9,1,10});//F -> (E)      7

    table["E"]["id"] = 0; table["E"]["("] = 0;
    table["E'"]["+"] = 1;table["E'"][")"] = 2;table["E'"]["$"] = 2;
    table["T"]["id"] = 3;table["T"]["("] = 3;
    table["T'"]["+"] = 4;table["T'"]["*"] = 5;table["T'"][")"] =
4;table["T'"]["$"] = 4;
    table["F"]["id"] = 6;table["F"]["("] = 7;

}
int main(){
    unordered_map<string,unordered_map<string,int>> table;
    vector<string> states;
    vector<vector<int>> rules;

    pre(table,rules,states);

    stack <string> st;
    st.push("$");
    st.push("E");
    int i =0,error = 0;

    // Tokenizing the input (space separated input expected)
    string input,s;
    vector<string> arr;
    getline(cin,input);

    for(int i = 0; i< input.size();i++){
        if(input[i] == ' '){
            arr.push_back(s);
            s= "";
        }else{
            s.push_back(input[i]);
        }
    }
    arr.push_back(s);
    arr.push_back("$");

    // parsing
    s = "";
    while(!st.empty() && i < arr.size()){
        input = st.top();
        st.pop();
        for(auto x: rules[table[input][arr[i]]]){
            cout << states[x];
            st.push(states[x]);
        }
        cout << endl;
        if(s == st.top()){
            cout << "yes\n";
            error = 1;
            break;
        }


        while(!st.empty() && st.top() == arr[i]){
            i++;
            st.pop();
        }
        if(!st.empty())
            s = st.top();
    }
    // cout << st.empty() << i <<endl;
    if(st.empty() && i == arr.size())
        cout <<"Its a Match"<<endl;
    else
        cout <<"NO Match Found"<<endl;
}
```

Input / Output :



Command Prompt

```
D:\sem -6\compiler design\LAB\LAB 6>g++ PredictiveParser.cpp

D:\sem -6\compiler design\LAB\LAB 6>a.exe
( ( id ) ) + id * id * id
Its a Match

D:\sem -6\compiler design\LAB\LAB 6>a.exe
( ( id ) ) )
NO Match Found

D:\sem -6\compiler design\LAB\LAB 6>a.exe
( ( id ) ) + ( ( id ) )
Its a Match

D:\sem -6\compiler design\LAB\LAB 6>_
```

Algorithm :

1) We construct an LL(1) parser table using the first and follow of each symbol from productions.(we have hard coded the table in this code)
2) Then input tokens are taken one by one and appropriate production rule is applied using the parser table.
3) If in the end we get $ both in input array and symbol stack. Then that string is accepted by the grammar else not.
4) Also if for some input and corresponding symbol on stack we don't have a production it is assumed that string cannot be accepted by the grammar

Learning :

1) Grammar should not be left recursive or left factoring.
2) Addition of $ symbol to input is necessary to mark the ending of input.
3) In predictive parsing, there’s only one choice of production, unlike the recursive descent parser, hence requires no backtracking.