PRACTICAL FILE

BE (CSE) 6th Semester

COMPILER DESIGN (CS654)

March 2021 – May 2021

Submitted By

Jatin Ghai
(UE183042)

Submitted To

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. 4

AIM :- To implement a Recursive Descent Parser for a given CFG

Parsing: 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.

Recursive Descent 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 backtracking.

CFG Used:

```
E->T+ E | T
T-> int | int * T|(E)
```

Source Code:

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

int nextt = 0;
vector<string> arr;

bool E(int level);
bool T(int level);

bool match(string s){
    cout <<"M ";
    // cout <<"match "<< nextt <<" "<<arr[nextt]<<" "<<s<<endl;
    if(nextt >= arr.size()) return false;
    return (arr[nextt++] == s);
```

```
}
bool T1(int level){
    cout <<"T1 ";
    int save = nextt;
    return match("int");
}
bool T2(int level){
    level++;
    cout <<"T2 ";
    int save = nextt;
    return (match("int") && match("*") && T(level));
}

bool T3(int level){
    level++;
    cout <<"T3 ";
    return (match("(") && E(level) && match(")"));
}

bool T(int level){
    level++;
    cout <<"T ";

    int save = nextt;
    bool a = T3(level);
    if(a) return a;

    nextt = save;
    bool b = T2(level);
    if(b) return b;

    nextt = save;
    bool c = T1(level);
    if(c) return c;

    return 0;
}

bool E(int level){
    cout << "E ";
```

```
    level++;

    int save = nextt;
    bool b = (T(level) && match("+") && E(level));

    if(level == 1){
        if(nextt < arr.size() && arr[nextt] == "$" && b) return b;
    }else {
        if(b) return b;
    }

    nextt = save;
    bool a = T(level);

    if(level == 1){
        if(arr[nextt] != "$") return 0;
        else return a;
    }else {
        return a;
    }

}

int main(){
    string input,s;
    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("$");
    if(E(0)) cout <<"\nString matched";
    else cout <<"\nNOT matched";
}
```

Input / Output :

```
C:\Users\jatin\Desktop>a.exe
( int )
E T T3 M E T T3 M T2 M M T1 M M T T3 M T2 M M T1 M M M T T3 M E T T3 M T2
String matched
C:\Users\jatin\Desktop>a.exe
( int + int )
E T T3 M E T T3 M T2 M M T1 M M E T T3 M T2 M M T1 M M T T3 M T2 M M T1 M
M M T1 M M T T3 M T2 M M T1 M M
String matched
C:\Users\jatin\Desktop>
```

Algorithm :

1) First we make a function for each non - terminal symbol.
2) Token matching is done for terminal symbols.
3) $ is appended into each input to mark its ending symbol.
4) $ is only matched at the level 1 of recursion.
5) For rules with multiple options , rules are explored in such a order that longest rule with maximum non- terminal are explored first.

Learning :

1) Grammar should not be left recursive or left factoring.
2) Backtracking will terminate if we explore rules with less non terminal symbols on the right side first. Since a match can be found in a small part of input leading to an unmatched string.
3) Addition of $ symbol to input is necessary to mark the ending of input.
4) we need to know the depth of recursion as we only check for $ in the depth 1.