

## **PRACTICAL-11**

**AIM : Write a C program for implementation of Recursive Descent Parser.**

### **INTRODUCTION:**

- Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right.
- It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking.
- But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.
- This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

### **Back-tracking :**

- Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:  
 $S \rightarrow rXd \mid rZd$   
 $X \rightarrow oa \mid ea$   
 $Z \rightarrow ai$
- For an input string: read, a top-down parser, will behave like this:
  - It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S ( $S \rightarrow rXd$ ) matches with it.
  - So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left ( $X \rightarrow oa$ ). It does not match with the next input symbol.
  - So the top-down parser backtracks to obtain the next production rule of X, ( $X \rightarrow ea$ ).

### **Predictive Parser:**

- Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.
- To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.
- Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol \$ to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.
- In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each

step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

**CODE:**

```
#include<stdio.h>

#include<ctype.h>

#include<string.h>

void Tprime();

void Eprime();

void E();

void check();

void T();

char expression[10];

int count, flag;

int main(){

    count = 0;

    flag = 0;

    printf("\nEnter an Algebraic Expression:\t");

    scanf("%s", expression);

    E();

    if((strlen(expression) == count) && (flag == 0)){

        printf("\nThe Expression %s is Valid\n", expression); }

    else{

        printf("\nThe Expression %s is Invalid\n", expression); }

}
```

```
void E(){
    T();
    Eprime(); }

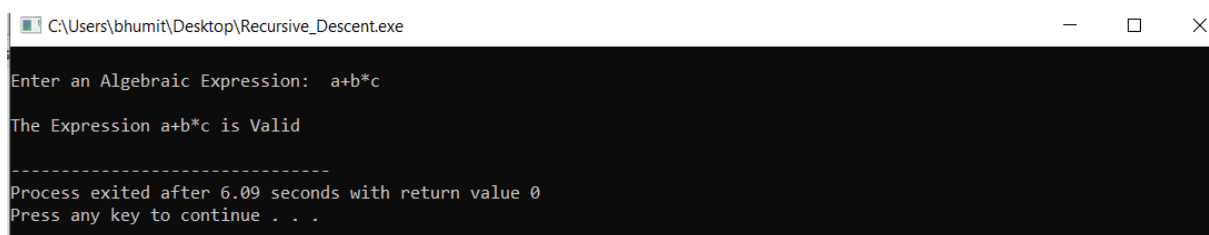
void T(){
    check();
    Tprime(); }

void Tprime(){
    if(expression[count] == '*'){
        count++;
        check();
        Tprime(); }}

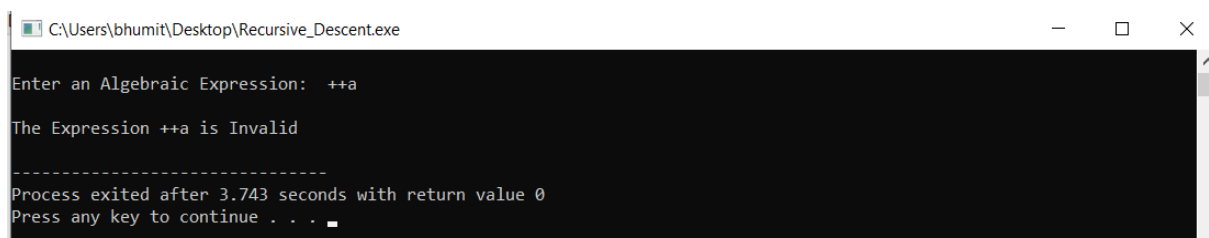
void check(){
    if(isalnum(expression[count])){
        count++; }
    else if(expression[count] == '(){
        count++;
        E();
        if(expression[count] == ')')
        {
            count++;
        }
        else
        {
            flag = 1;
        }
    }
```

```
    }  
  
    else  
  
    {  
  
        flag = 1;  
  
    }  
  
}  
  
void Eprime()  
  
{  
  
    if(expression[count] == '+')  
  
    {  
  
        count++;  
  
        T();  
  
        Eprime();  
  
    }  
  
}
```

## OUTPUT:



```
C:\Users\bhumit\Desktop\Recursive_Descent.exe  
Enter an Algebraic Expression: a+b*c  
The Expression a+b*c is Valid  
-----  
Process exited after 6.09 seconds with return value 0  
Press any key to continue . . .
```



```
C:\Users\bhumit\Desktop\Recursive_Descent.exe  
Enter an Algebraic Expression: ++a  
The Expression ++a is Invalid  
-----  
Process exited after 3.743 seconds with return value 0  
Press any key to continue . . .
```