Recursive Descent Parser

A recursive descent parser is a top-down parser that uses procedures for each non-terminal symbol. It then recursively parses the input to construct the parse tree. It may or may not use backtracking.

Recursive Descent Parser is a type of top-down parser, which means it builds the parse tree from the root to the leaf. It uses recursive functions to recognize the input and may or may not use backtracking. The form of recursive descent parser that does not require backtracking is also called predictive parsing.
Backtracking: Making repeated input scans until we find a correct path.
To implement a recursive descent parser, the grammar must hold the following properties:

1. It should not be left recursive.
2. It should be left-factored. (Alternates should not have common
3.).
4. Language should have a recursion facility.

If the grammar is not left-factored, the recursive descent parser will have to use backtracking.

Recursive descent parser is a top-down parser.

• It requires backtracking to find the correct production to be applied.

• The parsing program consists of a set of procedures, one for each non-terminal.

• Process begins with the procedure for start symbol.

• Start symbol is placed at the root node and on encountering each non-terminal, the procedure concerned is called to expand the non-terminal with its corresponding production.

• Procedure is called recursively until all non-terminals are expanded.

• Successful completion occurs when the scan over entire input string is done. ie., all terminals in the sentence are derived by parse tree.

```
void A()
{
      choose an A-production, A ---> X1 X2 X3... Xk;
        for (i = 1 to k)
```

```
            if (Xi is a non-terminal)
              call procedure Xi ();
              else if (Xi equals the current input symbol a)
                    advance the input to the next symbol;
              else
                    error;
}
```

Example
Consider the following grammar:
Here E and E' are non-terminals and '+', 'i' are characters.
E -> iE' (This means that E can be replaced with iE'.)
E' -> +iE / ε (This means that E' can be replaced either with +iE or with epsilon. Epsilon represents the end of the recursion.).
Now we will write procedures for each non-terminal.

```
# This is the procedure for E

E(){
        # l is the lookahead
        if (l == 'i'){
                match('i');
                E'();
        }
}


# This is the procedure for E'

E'(){
        if (l == '+'){
```

```
            match(‘+’)
            match(‘i’)
            E()
        }
}

# This is the procedure to match a character and input new character
match(char t){
        if (l == t) {
            l = getchar();
        }
        else{
            printf("Error");
        }
}

main(){
        E();
        if (l==’$’){
            printf(“Parsing Successful”);
        }
}
```

Pros and Cons

Pros: Recursive Descent parsers are very easy to implement. These are great for "quick and dirty" parsing.
Cons: These parsers are not as fast as some of the other parsers. We cannot use them for parses that require arbitrary long lookaheads.

Example :

The grammar on which we are going to do recursive descent parsing is:

E -> E+T | T
T -> T*F | F
F -> (E) | id

RD parser will verify whether the syntax of the input stream is correct by checking each character from left to right. A basic operation necessary is reading characters from the input stream and matching then with terminals from the grammar that describes the syntax of the input.

The given grammar can accept all arithmetic equations involving +, * and ().
eg:
a+(a*a) a+a*a , (a), a , a+a+a*a+a.... etc are accepted
a++a, a***a, +a, a*, ((a . . . etc are rejected.

Solution:
First we have to avoid left recursion

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | id

After eliminating Left recursion, we have to simply move from one character to next by checking whether it follow the grammar. In this program, ε is indicated as $.

Write down the algorithm using Recursive procedures to implement the following Grammar.

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$T \rightarrow FT'$

$T' \rightarrow * FT'|\varepsilon$

$F \rightarrow (E)|id$

Solution

```
Procedure E ( )                          ┐
{                                        │
        T ( );                           │  E → TE'
        E'( );                           │
}                                        ┘
Procedure E'( )                          ┐
{                                        │
        If input symbol ='+' then        │  E → + TE'
                                         │
                                         │
                                         │
                                         │
                                         │
                                         │
        advance( );                      │
        T ( );                           │
        E'( );                           ┘
}
```

```
Procedure T( )
{
        F ( );
        T′( );
}
```

$T \rightarrow F\,T'$

```
Procedure T′( )
{
        If input symbol ='*' then
        advance( );
        F ( );
        T′( );
}
```

$T' \rightarrow * \; FT'$

```
Procedure F ( )
{
        If input symbol ='id' then
        advance ( );
        else if input-symbol ='(' then
        advance ( );
        E ( );
        If input-symbol = ')'
        advance ( );
        else error ( );
        else error ( );
}
```

$F \rightarrow id$

$F \rightarrow (E)$

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
```

```
char input[10];
int i,error;
void E();
void T();
void Eprime();
void Tprime();
void F();
        main()
        {
i=0;
error=0;
            printf("Enter an arithmetic expression   :  "); // Eg: a+a*a
            gets(input);
            E();
            if(strlen(input)==i&&error==0)
                printf("\nAccepted..!!!\n");
            else printf("\nRejected..!!!\n");
                    }
void E()
{
    T();
    Eprime();
}
```

```
void Eprime()
{
    if(input[i]=='+')
    {
    i++;
    T();
    Eprime();
    }
    }
void T()
{
    F();
    Tprime();
}
void Tprime()
{
    if(input[i]=='*')
    {
                i++;
                F();
                Tprime();
                }
                }
```

```
void F()
{
    if(isalnum(input[i]))i++;
    else if(input[i]=='(')
    {
    i++;
    E();
    if(input[i]==')')
    i++;

    else error=1;
      }
    else error=1;
    }
```

Output:

a+(a*a) a+a*a , (a), a , a+a+a*a+a.... etc are accepted
++a, a***a, +a, a*, ((a . . . etc are rejected.