GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL

COMPUTER ENGINEERING DEPARTMENT

A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: **System Programming & Compiler Construction**

Experiment No: 6

Title of Experiment: Implementation of Recursive Descent Parser.

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	10	
2	Oral	5	
	Total	15	

Signature of Subject Teacher

Theory:

Recursive Descent Parser

Recursive Descent Parser uses the technique of Top-Down Parsing without backtracking. It can be defined as a Parser that uses the various recursive procedure to process the input string with no backtracking. It can be simply performed using a Recursive language. The first symbol of the string of R.H.S of production will uniquely determine the correct alternative to choose.

The major approach of recursive-descent parsing is to relate each non-terminal with a procedure. The objective of each procedure is to read a sequence of input characters that can be produced by the corresponding non-terminal, and return a pointer to the root of the parse tree for the non-terminal. The structure of the procedure is prescribed by the productions for the equivalent non-terminal.

The recursive procedures can be simply to write and adequately effective if written in a language that executes the procedure call effectively. There is a procedure for each non-terminal in the grammar. It can consider a global variable lookahead, holding the current input token and a procedure match (Expected Token) is the action of recognizing the next token in the parsing process and advancing the input stream pointer, such that lookahead points to the next token to be parsed. Match () is effectively a call to the lexical analyzer to get the next token.

For example, input stream is a + b$.

lookahead == a

match()

lookahead == +

match ()

lookahead == b

The working of Recursive Descent Parser can be broken down into the following steps:

1. **Definition of grammar**: The grammar is defined using a formal notation such as BNF (Backus-Naur Form) or EBNF (Extended Backus-Naur Form).
2. **Creation of parsing functions**: Each non-terminal symbol in the grammar is associated with a separate function. These functions are called recursively to parse the input string.
3. **Input tokenization**: The input string is broken down into a sequence of tokens. Each token represents a terminal symbol in the grammar.
4. **Parsing**: The parsing process starts with the first function that represents the start symbol of the grammar. The function looks at the first token and decides which function to call next, based on the production rules of the grammar.
5. **Recursive parsing**: The called function looks at the next token and decides which function to call next, and so on. This recursive process continues until the input string is fully parsed.
6. **Error handling**: If the parser encounters an error during parsing, it stops and returns an error message to the user.
7. **Output**: If the input string is parsed successfully, the parser returns a parse tree or an Abstract Syntax Tree (AST). This can be used for further processing, such as code generation.

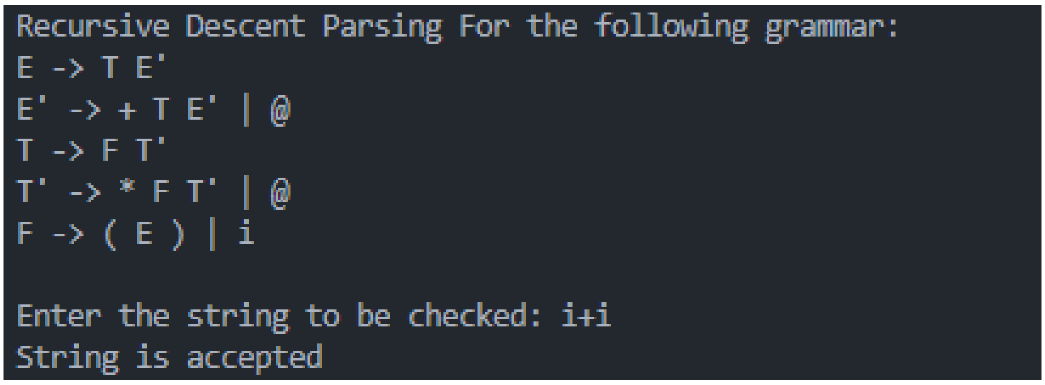
Program Code –

```
# Read the input grammar from a file
with open("Expt6.txt", "r") as file:
    grammar = file.read()

print("Recursive Descent Parsing For the following grammar: ")
print(grammar)

# Take the input string from the user
s = input("\nEnter the string to be checked: ")

i = 0

# Define the match function
def match(a):
    global s
    global i
    if i >= len(s):
        return False
    elif s[i] == a:
        i += 1
        return True
    else:
        return False
# Define the F function
def F():
    if match("("):
        if E():
            if match(")"):
                return True
            else:
                return False
        else:
            return False
    elif match("i"):
        return True
    else:
        return False

# Define the Tx function
def Tx():
    if match("*"):
        if F():
            if Tx():
                return True
            else:
                return False
        else:
            return False
    else:
        return True

# Define the T function
def T():
    if F():
        if Tx():
            return True
        else:
            return False
    else:
        return False

# Define the Ex function
def Ex():
    if match("+"):
        if T():
            if Ex():
                return True
            else:
                return False
        else:
            return False
    else:
        return True

# Define the E function
def E():
    if T():
        if Ex():
            return True
        else:
            return False
    else:
        return False

# Check whether the input string is generated by the grammar or not
if E():
    if i == len(s):
        print("String is accepted")
    else:
        print("String is not accepted")
else:
    print("String is not accepted")
```

Input File

```
E -> T E'
E' -> + T E' | @
T -> F T'
T' -> * F T' | @
F -> ( E ) | i
```

Output –



```
Recursive Descent Parsing For the following grammar:
E -> T E'
E' -> + T E' | @
T -> F T'
T' -> * F T' | @
F -> ( E ) | i

Enter the string to be checked: i+i
String is accepted
```

Conclusion –

Thus we studied the implementation of Recursive Descent Parser.