Practical 11:

Aim: Write an algorithm and program on Recursive Descent parse

Algorithm:

- Define a set of recursive procedures, one for each non-terminal symbol in the grammar. These procedures will be responsible for recognizing and parsing the corresponding non-terminal symbol.
- Define a function to get the next token from the input stream.
- Define a function to match a token against an expected token. If the token matches, consume the token from the input stream. Otherwise, throw a syntax error.
- Define a procedure for the start symbol of the grammar. This procedure will call the appropriate recursive procedures to parse the input string.
- Begin parsing by calling the start symbol procedure.
- Within each recursive procedure, first get the next token from the input stream.
- Then, based on the current non-terminal symbol, call the appropriate recursive procedure or match the token against an expected terminal symbol.
- If the current non-terminal symbol has multiple production rules, try each rule in order until one succeeds. If none of the rules succeed, throw a syntax error.
- Repeat steps 6-8 until the entire input string has been parsed.
- If the parser successfully parses the input string, return a parse tree. Otherwise, throw a syntax error.

Python Program:

```
grammar_rules = {
    'S': [('A', 'B', 'C')],
    'A': [('a',)],
    'B': [('b',)],
    'C': [('c',), ('d', 'C')]}

terminal_symbols = {'a', 'b', 'c', 'd'}

start_symbol = 'S'

# Define parse tree data structure
class ParseTreeNode:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children if children is not None else []
```

```
 # Define recursive functions
def parse_S(tokens):
    parse_tree_node = ParseTreeNode('S')
    children = []
    children.append(parse_A(tokens))
    children.append(parse_B(tokens))
    children.append(parse_C(tokens))
    parse_tree_node.children = children
    return parse_tree_node

def parse_A(tokens):
    parse_tree_node = ParseTreeNode('A')
    if tokens[0] == 'a':
        parse_tree_node.children.append(ParseTreeNode(tokens[0]))
        tokens.pop(0)
    else:
        raise ValueError('Invalid token for A')
    return parse_tree_node

def parse_B(tokens):
    parse_tree_node = ParseTreeNode('B')
    if tokens[0] == 'b':
        parse_tree_node.children.append(ParseTreeNode(tokens[0]))
        tokens.pop(0)
    else:
        raise ValueError('Invalid token for B')
    return parse_tree_node
 def parse_C(tokens):
    parse_tree_node = ParseTreeNode('C')
    if tokens[0] == 'c':
        parse_tree_node.children.append(ParseTreeNode(tokens[0]))
        tokens.pop(0)
    elif tokens[0] == 'd':
        parse_tree_node.children.append(ParseTreeNode(tokens[0]))
        tokens.pop(0)
        parse_tree_node.children.append(parse_C(tokens))
    else:
        raise ValueError('Invalid token for C')
    return parse_tree_node

def parse(input_string):
    tokens = input_string.split()
    parse_tree_node = parse_S(tokens)
    if len(tokens) != 0:
        raise ValueError('Invalid input string')
    return parse_tree_node

input_string = 'a b c'
parse_tree_node = parse(input_string)
print(parse_tree_node)
```

Program Output:

```
Enter the string

Input              Action
--------------------------------
i+(i+i)*i          E -> T E'
i+(i+i)*i          T -> F T'
+(i+i)*i           F -> i
+(i+i)*i           T' -> $
+(i+i)*i           E' -> + T E'
(i+i)*i            T -> F T'
(i+i)*i            F -> ( E )
i+i)*i             E -> T E'
i+i)*i             T -> F T'
+i)*i              F -> i
+i)*i              T' -> $
+i)*i              E' -> + T E'
i)*i               T -> F T'
)*i                F -> i
)*i                T' -> $
)*i                E' -> $
*i                 T' -> * F T'
                   F -> i
                   T' -> $
                   E' -> $
--------------------------------
String is successfully parsed
```