**✅ Title: Implementing Recursive Descent Parser for Arithmetic Construct**

**🎯 Aim:**

To implement a **Recursive Descent Parser** for a given arithmetic grammar.

**📌 Objectives:**

1. To study the **parsing phase** in the compiler.
2. To understand **top-down** and **bottom-up** parsing.
3. To identify **problems in top-down parsers** (e.g., left recursion).
4. To learn **writing a top-down parser** manually using recursive functions.

**📖 Theory**

**📘 1. Context-Free Grammar (CFG) and Its Elements**

| **Element** | **Description** |
| --- | --- |
| **CFG** | A grammar that defines syntactic rules where productions have one non-terminal on the LHS. |
| **Non-terminals** | Variables like E, T, F representing grammar constructs. |
| **Terminals** | Actual symbols like +, \*, (, ), id. |
| **Productions** | Rules defining how non-terminals can be replaced. |
| **Derivation** | A sequence of production applications generating a string. |

**🔁 2. Introduction to Recursive Descent Parser**

* A **top-down parser** built from mutually recursive functions.
* Each non-terminal has a corresponding function.
* Matches the input string against grammar rules.
* Cannot handle **left-recursive grammars** directly.

**⚠️ Left Recursion Problem**

**❌ Original Grammar:**

E → E + T | T

T → T \* F | F

F → ( E ) | id

This grammar is **left-recursive** and must be modified.

**✅ After Eliminating Left Recursion**

E → T E'

E' → + T E' | ε

T → F T'

T' → \* F T' | ε

F → ( E ) | id

Now suitable for recursive descent parsing.

**🧪 Input:**

* A string like id + id \* id (valid)
* A string like id + \* id (invalid)

**📤 Output:**

* Success if input conforms to grammar.
* Failure if input violates grammar rules.

**✅ Conclusion:**

The **recursive descent parser** was successfully implemented for arithmetic expressions. It accepts valid strings and handles syntax errors for invalid inputs.

**💻 Platform:**

Linux (C / C++ / Java)

**🎤 Viva Questions with One-Line Answers**

**🧠 Parsing & Grammar**

1. **What is a recursive descent parser?**  
   → A top-down parser built using recursive procedures for each grammar rule.
2. **Why can't recursive descent parsers handle left recursion?**  
   → Because it causes **infinite recursion** during parsing.
3. **What is CFG?**  
   → A grammar where productions have one non-terminal on the left-hand side.

**📘 Grammar Rules**

1. **What are terminals in CFG?**  
   → Symbols that appear in the final string (like id, +, \*, (, )).
2. **What are non-terminals?**  
   → Grammar variables (like E, T, F) used to derive terminal strings.
3. **What is left recursion?**  
   → A production where a non-terminal appears first on the RHS (e.g., E → E + T).
4. **How do you eliminate immediate left recursion?**  
   → Use a transformation to right recursion with auxiliary non-terminals (e.g., E → T E', E' → + T E' | ε).

**🧪 Parser Behavior**

1. **What is the output of a recursive descent parser?**  
   → Success for valid input; error for invalid syntax.
2. **What does ε mean in grammar?**  
   → It denotes an empty string (no symbol).
3. **What happens when input doesn't match any rule?**  
   → The parser returns failure and signals a syntax error.
4. **Which function handles input matching?**  
   → Functions for each non-terminal recursively match and validate the input.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Global variables

int i = 0;

char str[100] = "(id+id)\*id";  // Hardcoded expression

char tp;

// Function prototypes

void advance();

void E();

void EPrime();

void T();

void TPrime();

void F();

// Advance to the next character in the input string

void advance() {

    tp = str[i++];

}

// F → id | (E)

void F() {

    if (tp == 'i') {

        advance();

        if (tp == 'd') {

            advance();

        } else {

            printf("String not Accepted.\n");

            exit(1);

        }

    } else if (tp == '(') {

        advance();

        E();

        if (tp == ')') {

            advance();

        } else {

            printf("String not Accepted.\n");

            exit(1);

        }

    } else {

        printf("String not Accepted.\n");

        exit(1);

    }

}

// T → F T'

void T() {

    F();

    TPrime();

}

// T' → \* F T' | ε

void TPrime() {

    if (tp == '\*') {

        advance();

        F();

        TPrime();

    }

}

// E → T E'

void E() {

    T();

    EPrime();

}

// E' → + T E' | ε

void EPrime() {

    if (tp == '+') {

        advance();

        T();

        EPrime();

    }

}

// Main function

int main() {

    printf("Input string: %s\n", str);

    advance();  // Start parsing

    E();

    if (tp == '\0') {

        printf("String Accepted.\n");

    } else {

        printf("String not Accepted.\n");

    }

    return 0;

}