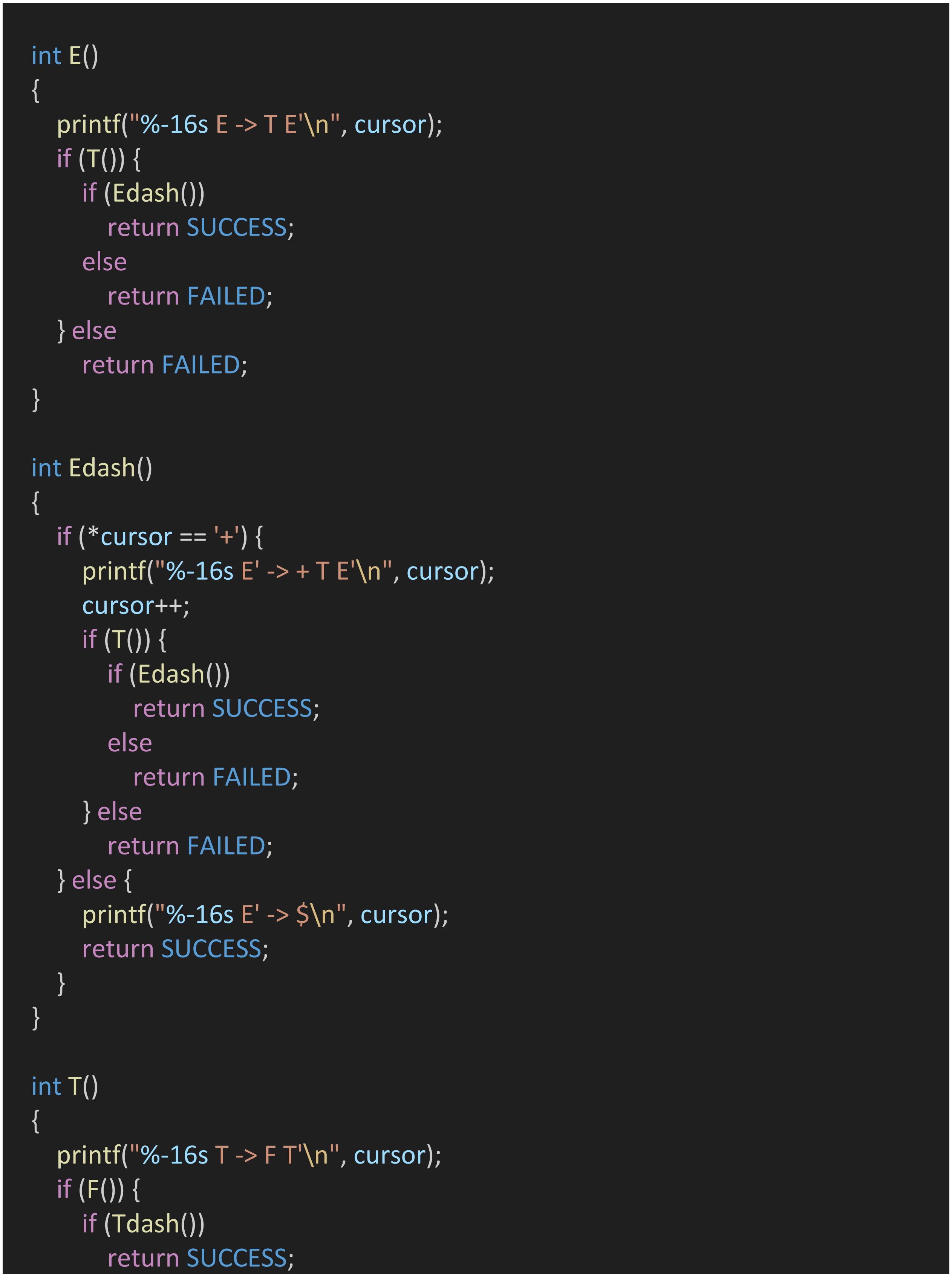
Assignment no: 5

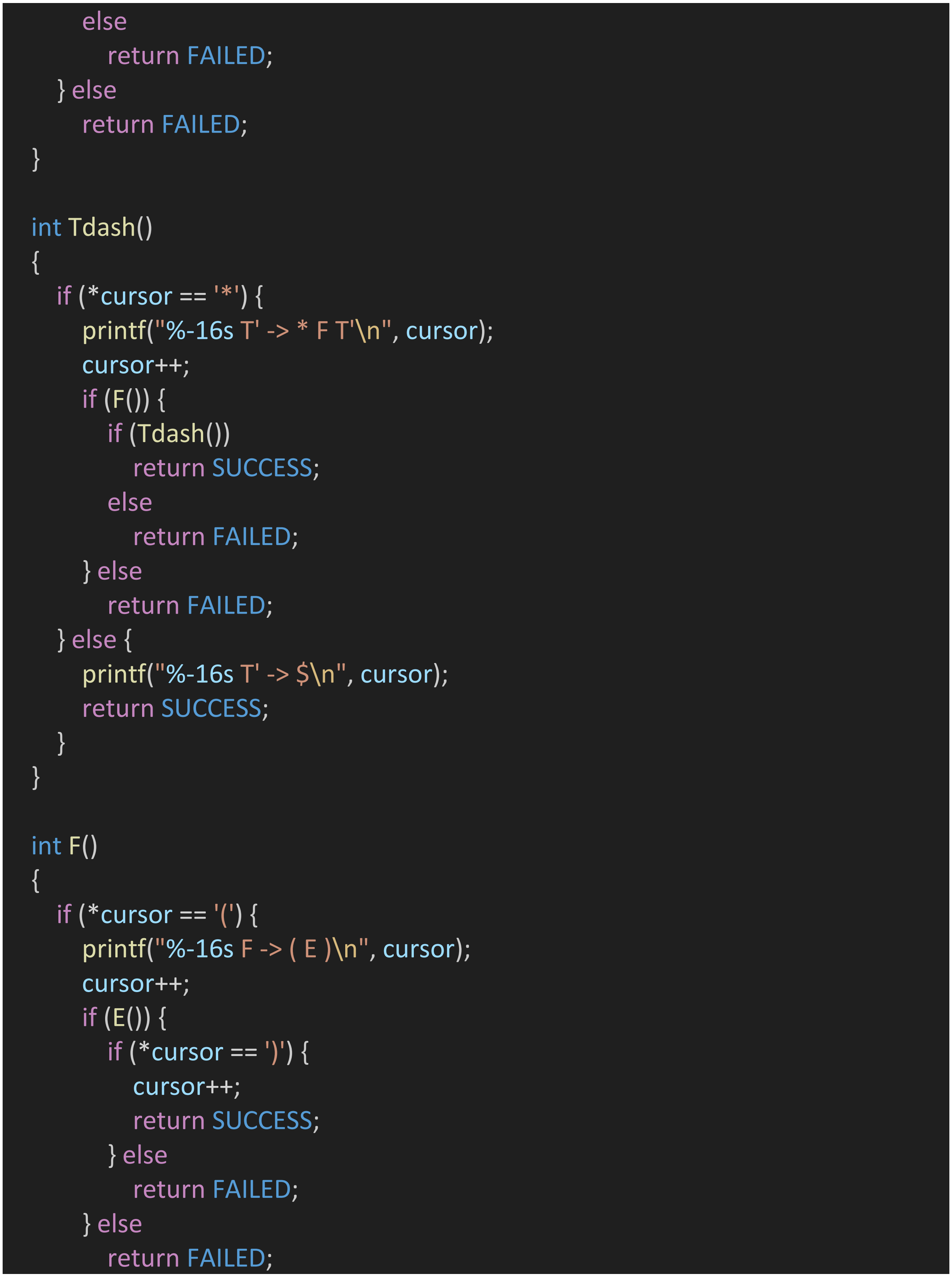
Write a program to implement recursive descent parser(RDP) for sample language

Rdp.c

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

#define SUCCESS 1
#define FAILED 0

int E(), Edash(), T(), Tdash(), F();

const char *cursor;
char string[64];

int main()
{
   puts("Enter the string");
   // scanf("%s", string);
   sscanf("i+(i+i)*i", "%s", string);
   cursor = string;
   puts("");
   puts("Input      Action");
   puts("--------------------------------");

   if (E() && *cursor == '\0') {
      puts("--------------------------------");
      puts("String is successfully parsed");
      return 0;
   } else {
      puts("--------------------------------");
      puts("Error in parsing String");
      return 1;
   }
}
```



```
int E()
{
    printf("%-16s E -> T E'\n", cursor);
    if (T()) {
        if (Edash())
            return SUCCESS;
        else
            return FAILED;
    } else
        return FAILED;
}

int Edash()
{
    if (*cursor == '+') {
        printf("%-16s E' -> + T E'\n", cursor);
        cursor++;
        if (T()) {
            if (Edash())
                return SUCCESS;
            else
                return FAILED;
        } else
            return FAILED;
    } else {
        printf("%-16s E' -> $\n", cursor);
        return SUCCESS;
    }
}

int T()
{
    printf("%-16s T -> F T'\n", cursor);
    if (F()) {
        if (Tdash())
            return SUCCESS;
```



```
        else
            return FAILED;
    } else
        return FAILED;
}

int Tdash()
{
    if (*cursor == '*') {
        printf("%-16s T' -> * F T'\n", cursor);
        cursor++;
        if (F()) {
            if (Tdash())
                return SUCCESS;
            else
                return FAILED;
        } else
            return FAILED;
    } else {
        printf("%-16s T' -> $\n", cursor);
        return SUCCESS;
    }
}

int F()
{
    if (*cursor == '(') {
        printf("%-16s F -> ( E )\n", cursor);
        cursor++;
        if (E()) {
            if (*cursor == ')') {
                cursor++;
                return SUCCESS;
            } else
                return FAILED;
        } else
            return FAILED;
    } else if (*cursor == 'i') {
        cursor++;
        printf("%-16s F ->i\n", cursor);
        return SUCCESS;
    } else
        return FAILED;
}
```

Output:

cc Rdp.c

./a.out

Explanation :

Certainly! Let's walk through the working of the given recursive descent parser program for the input `i+(i+i)*i`.

Step-by-Step Explanation

1. **Initialization:**

   - The program starts by initializing the input string to `i+(i+i)*i` and setting the cursor to point to the beginning of this string.

   - The main function calls `E()` to start parsing the input.

2. **Parsing Expression (E):**

   - The function `E()` is called first.

   - It prints the current state and attempts to match the rule `E -> T E'`.

   - It then calls `T()` to parse a term.

3. **Parsing Term (T):**

- The function `T()` is called from within `E()`.

- It prints the current state and attempts to match the rule `T -> F T'`.

- It then calls `F()` to parse a factor.

4. **Parsing Factor (F):**

- The function `F()` checks if the current character is an opening parenthesis `(` or an identifier `i`.

- In this case, the first character is `i`, so it matches the rule `F -> i`.

- It consumes `i` (moves the cursor forward) and returns success to `T()`.

5. **Continuing Term (Tdash):**

- After `F()` returns success, `T()` calls `Tdash()` to check for any multiplications.

- `Tdash()` checks if the current character is `*`. It is not (the cursor is at `+`), so it matches the rule `T' -> ε` (empty string) and returns success to `T()`.

- `T()` returns success to `E()`.

6. **Continuing Expression (Edash):**

- After `T()` returns success, `E()` calls `Edash()` to check for any additions.

- `Edash()` finds `+` at the current cursor position.

- It matches the rule `E' -> + T E'`, consumes `+`, and calls `T()` again to parse the next term.

7. **Parsing Nested Term (T):**

   - The function `T()` is called again from within `Edash()`.

   - It prints the current state and calls `F()` to parse the next factor.

8. **Parsing Nested Factor (F):**

   - This time `F()` finds `(` and matches the rule `F -> ( E )`.

   - It consumes `(`, calls `E()` to parse the nested expression inside the parentheses.

9. **Parsing Nested Expression (E):**

   - Inside the parentheses, `E()` is called and follows the same process:

     - Calls `T()` which calls `F()` to parse `i`.

     - `F()` matches `i`, returns success to `T()`.

     - `T()` calls `Tdash()`, which matches `ε` and returns success to `T()`.

     - `T()` returns success to `E()`.

   - `E()` then calls `Edash()`, which finds `+` and calls `T()` to parse the next term.

10. **Parsing Next Nested Term (T):**

    - `T()` calls `F()` to parse `i`.

    - `F()` matches `i`, returns success to `T()`.

    - `T()` calls `Tdash()`, which matches `ε` and returns success to `T()`.

    - `T()` returns success to `Edash()`, which also matches `ε` and returns success to `E()`.

    - The closing parenthesis `)` is then consumed by `F()`, returning success to the original `T()`.

11. **Continuing with Multiplication (Tdash):**

   - After parsing the nested expression, `T()` (from step 7) calls `Tdash()` which finds `*`.

   - It matches the rule `T' -> * F T'`, consumes `*`, and calls `F()` to parse `i`.

12. **Parsing Final Factor (F):**

   - `F()` matches `i`, returns success to `Tdash()`.

   - `Tdash()` calls itself again and matches `ε`, returning success to the original `T()`.

13. **Completing Parsing:**

   - `T()` returns success to `Edash()`, which matches `ε` and returns success to `E()`.

   - `E()` returns success to `main()`.

   - Finally, the main function checks if the cursor has reached the end of the string (`\0`), indicating successful parsing.

Summary:

- The parser follows the recursive descent method, where each function corresponds to a non-terminal in the grammar.

- It recursively applies grammar rules to match the input string.

- If the entire string is matched successfully without any remaining characters, the input is valid according to the grammar, and it prints "String is successfully parsed".

- If any mismatch occurs, it prints "Error in parsing String".