Department of CSE
SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VI

26 February 2021

UCS 1602 - Compiler Design

Exercise 4: Recursive Descent Parser Using C

Aim:

Write a program in C to construct **Recursive Descent Parser** for the following grammar which is for arithmetic expression involving + and *. Check the Grammar for left recursion and convert into suitable for this parser. Write recursive functions for every non-terminal. Call the function for start symbol of the Grammar in main().

G1:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow i$

Extend this parser to include division, subtraction and parenthesis operators.

G2:

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid i$

Code - Grammar 1:

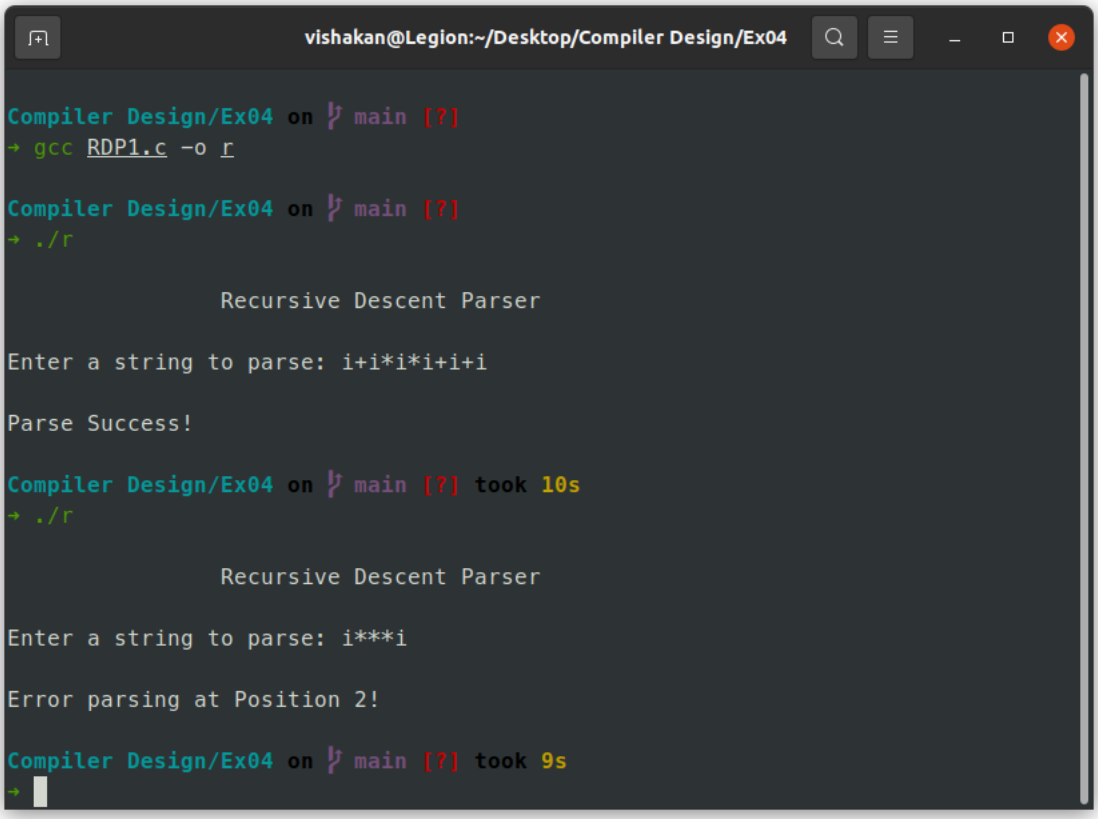
```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

/*Recursive Descent Parser*/

/*
Grammar: G: E->E+T|T
            T->T*F|F
            F->i
*/

/*
Removed Left Recursion
Grammar G': E->TE'
            E'->+TE'|e
            T->FT'
            T'->*FT'|e
            F->i
*/

struct parse_struct{
    char str[100];
    int pos;
    int len;
};

typedef struct parse_struct parser;

parser E(parser p);
parser T(parser p);
parser EPrime(parser p);
parser F(parser p);
parser TPrime(parser p);
parser parse(parser p, char s);

int main(void){
    parser p;

    printf("\n\t\tRecursive Descent Parser\n");
    printf("\nEnter a string to parse: ");
    scanf("%s", p.str);

    p.len = strlen(p.str);
    p.pos = 0;

    p = E(p);
```

```c

    if(p.pos == p.len){
        //All characters have been parsed
        printf("\nParse Success!\n");
    }

    else{
        //Some characters haven't been parsed, but returned to main
        printf("\nError parsing at Position %d!\n", p.pos);
    }


    return 0;
}

parser E(parser p){
    //printf("\nAt E");
    p = T(p);
    p = EPrime(p);

    return p;
}

parser T(parser p){
    //printf("\nAt T");
    p = F(p);
    p = TPrime(p);

    return p;
}

parser EPrime(parser p){
    //printf("\nAt EPrime");
    if(p.str[p.pos] == '+'){
        p = parse(p, '+');
        p = T(p);
        p = EPrime(p);
    }

    return p;
}

parser TPrime(parser p){
    //printf("\nAt TPrime");
    if(p.str[p.pos] == '*'){
        p = parse(p, '*');
        p = F(p);
        p = TPrime(p);
    }

    return p;
```

```c
}

parser F(parser p){
    //printf("\nAt F");
    if(p.str[p.pos] == 'i'){
        p = parse(p, 'i');
    }
    else{
        printf("\nError parsing at Position %d!\n", p.pos);
        exit(0);
    }

    return p;
}

parser parse(parser p, char s){
    if(p.str[p.pos] != s){
        printf("\nError parsing at Position %d!\n", p.pos);
        exit(0);
    }
    else{
        p.pos++;
    }

    return p;
}
```

Output - Grammar 1:

Figure 1: Console Output for parsed strings of Grammar 1.



Code - Grammar 2:

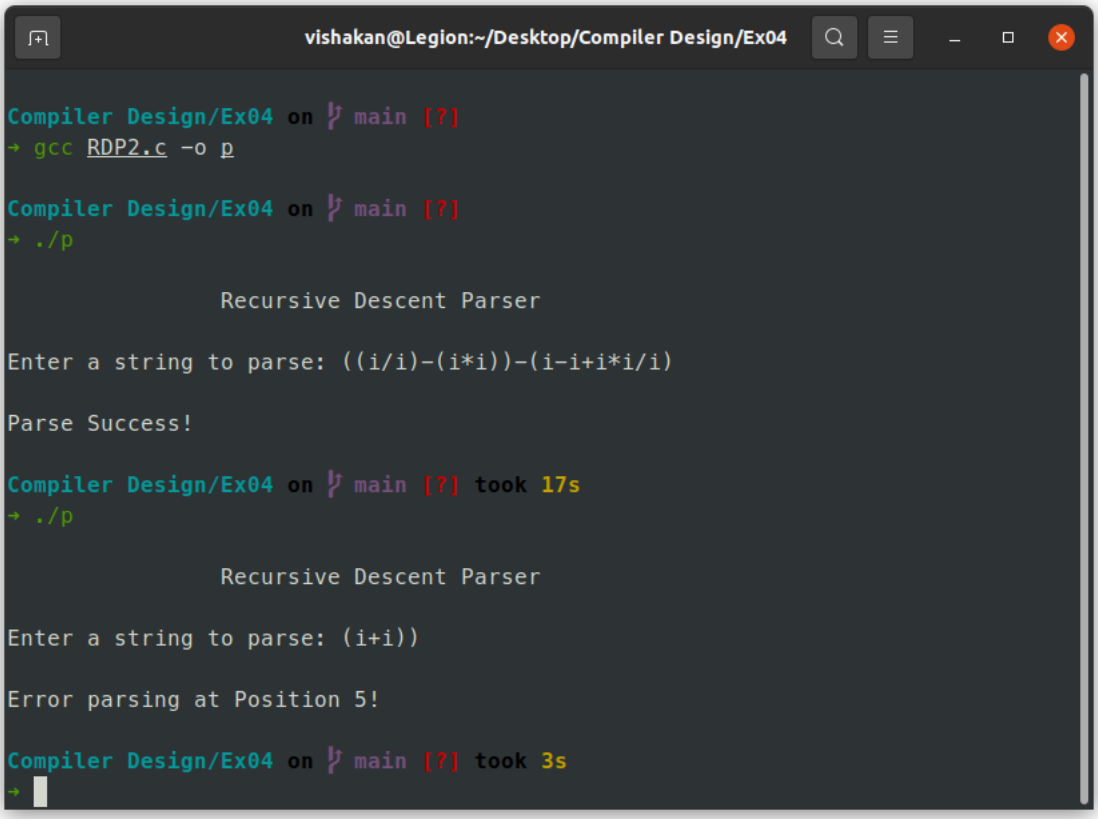
```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

/*Recursive Descent Parser*/

/*
Grammar: G: E->E+T|E-T|T
            T->T*F|T/F|F
            F->(E)|i
*/

/*
Removed Left Recursion
Grammar G': E->TE'
            E'->+TE'|-TE'|e
            T->FT'
            T'->*FT'|/FT'|e
            F->(E)|i
*/

struct parse_struct{
    char str[100];
    int pos;
    int len;
};

typedef struct parse_struct parser;

parser E(parser p);
parser T(parser p);
parser EPrime(parser p);
parser F(parser p);
parser TPrime(parser p);
parser parse(parser p, char s);

int main(void){
    parser p;

    printf("\n\t\tRecursive Descent Parser\n");
    printf("\nEnter a string to parse: ");
    scanf("%s", p.str);

    p.len = strlen(p.str);
    p.pos = 0;

    p = E(p);
```

```c

    if(p.pos == p.len){
        //All characters have been parsed
        printf("\nParse Success!\n");
    }

    else{
        //Some characters haven't been parsed, but returned to main
        printf("\nError parsing at Position %d!\n", p.pos);
    }

    return 0;
}

parser E(parser p){
    //printf("\nAt E");
    p = T(p);
    p = EPrime(p);

    return p;
}

parser T(parser p){
    //printf("\nAt T");
    p = F(p);
    p = TPrime(p);

    return p;
}

parser EPrime(parser p){
    //printf("\nAt EPrime");
    if(p.str[p.pos] == '+'){
        p = parse(p, '+');
        p = T(p);
        p = EPrime(p);
    }
    else if(p.str[p.pos] == '-'){
        p = parse(p, '-');
        p = T(p);
        p = EPrime(p);
    }

    return p;
}

parser TPrime(parser p){
    //printf("\nAt TPrime");
    if(p.str[p.pos] == '*'){
        p = parse(p, '*');
        p = F(p);
```

```
        p = TPrime(p);
    }
    else if(p.str[p.pos] == '/'){
        p = parse(p, '/');
        p = F(p);
        p = TPrime(p);
    }

    return p;
}

parser F(parser p){
    //printf("\nAt F");
    if(p.str[p.pos] == '('){
        p = parse(p, '(');
        p = E(p);
        p = parse(p, ')');
    }
    else if(p.str[p.pos] == 'i'){
        p = parse(p, 'i');
    }
    else{
        printf("\nError parsing at Position %d!\n", p.pos);
        exit(0);
    }

    return p;
}

parser parse(parser p, char s){
    if(p.str[p.pos] != s){
        printf("\nError parsing at Position %d!\n", p.pos);
        exit(0);
    }
    else{
        p.pos++;
    }

    return p;
}
```

Output - Grammar 2:

Figure 2: Console Output for parsed strings of Grammar 2.



Learning Outcome:

- I understood about the working of a **Recursive Descent Parser**.
- I understood that Recursive Descent Parser, being a Top-Down Parser, does not work with Left-Recursive Grammars.
- I was able to implement a working Recursive Descent Parser for a simple grammar.
- I was able to extend the concept to implement a Recursive Descent Parser for a complicated grammar with more productions.
- I refreshed my concepts with recursion & return handling in functions with C.
- I understood how to manually perform the Recursive Descent Parsing Process.