A4-Recursive Descent Parser using C

Sneha Sriram Kannan 185001157

26-02-2021

1 Code for input 1

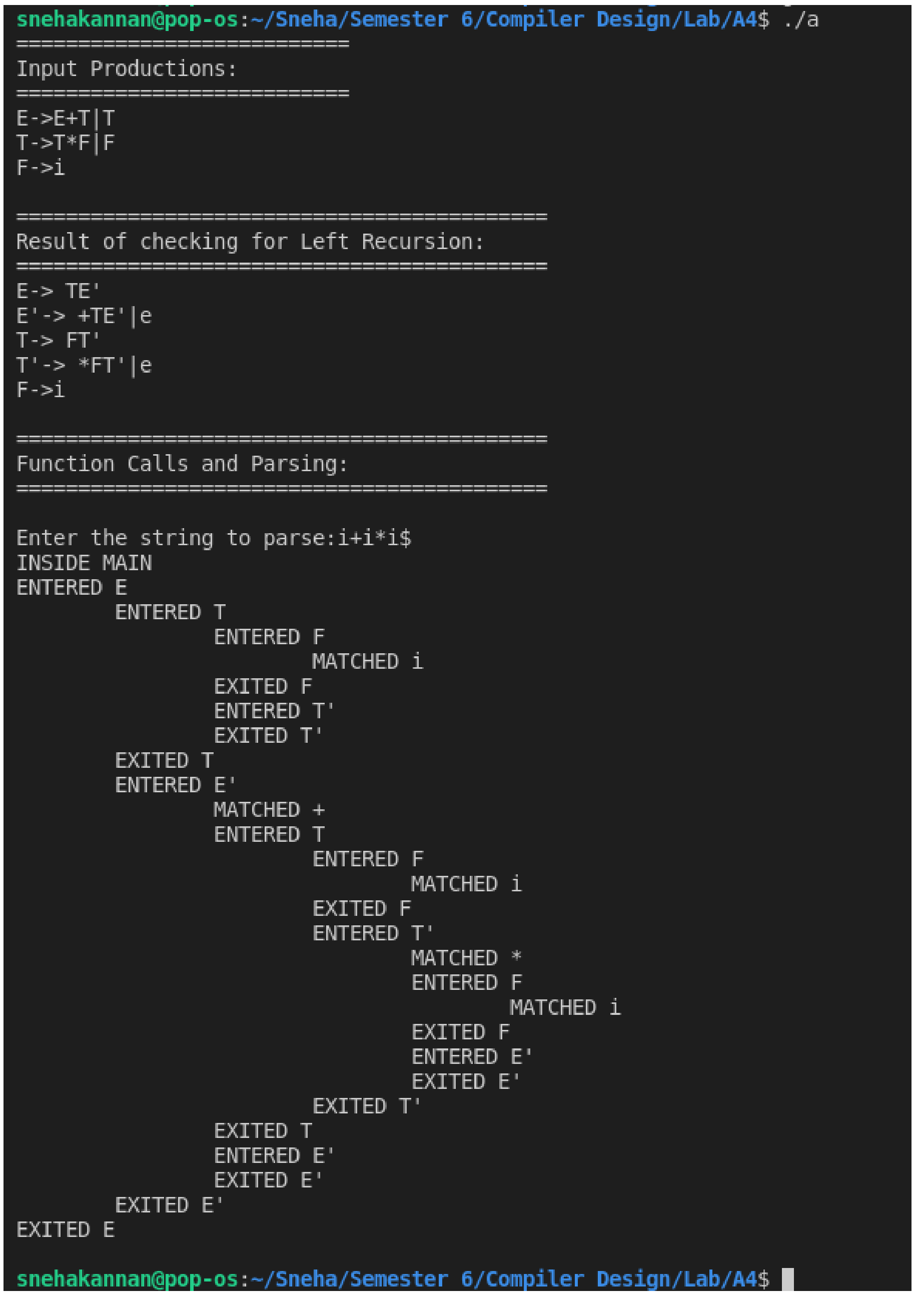
```c
#include "functions1.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct parsestring
{
    char instring[128];
    int ptr;
} parsestring;

int substr(char str1[], char str2[]);
int checkFunction(char str[]);
int EliminateLeftRecursion();
int Eliminate(char production[], char newFile[10][128], int count, int pos);
int recursiveDescent(char newProduction[10][128], int count);
int checkterminal(char ch);
void E(parsestring *p,int tab);
void E_Dash(parsestring *p,int tab);
void T(parsestring *p,int tab);
void T_Dash(parsestring *p,int tab);
void F(parsestring *p,int tab);

int main()
{
    EliminateLeftRecursion();
    return 0;
}

int EliminateLeftRecursion()
{
    char file[10][128];
    char newProduction[10][128];
    char functions[128][128];
    int newProductionCount = 0;
    int lrcount = 0;
    char inputfile[30];
    FILE *fd;
    fd = fopen("input1.txt", "r");
    int i = 0;
    //reading code from a file and storing in an array
    while (fgets(file[i], sizeof(file[i]), fd))
        i++;

    printf("========================\n");
    printf("Input Productions:\n");
    printf("===========================\n");
    for (int j = 0; j < i; j++)
    {
        printf("%s", file[j]);
    }
    printf("\n\n");
    printf("===========================================\n");
    printf("Result of checking for Left Recursion:\n");
    printf("===========================================\n");
    for (int j = 0; j < i; j++)
    {
        char lhs = file[j][0];
        int noLeft = 1;
        for (int k = 3; k < strlen(file[j]); k++)
        {
            if (lhs == file[j][k]) //checking if left recursion occurs in the productions
            {
                noLeft = 0;
                Eliminate(file[j], newProduction, newProductionCount, k);
                newProductionCount += 2;
                lrcount++;
                break;
            }
        }
        if (noLeft == 1) //no LR so no change in the production
        {
            strcpy(newProduction[newProductionCount++], file[j]);
        }
    }
    if (lrcount == 0)
        printf("NO LEFT RECURSION\n");
    else
    {
        for (int j = 0; j < newProductionCount; j++)
        {
            printf("%s", newProduction[j]);
        }
    }
    printf("\n\n");
    recursiveDescent(newProduction, newProductionCount);
}

int Eliminate(char production[], char newFile[10][128], int count, int pos)
{
    char new[3];
    new[0] = production[0];
    new[1] = '\'';
    new[2] = '\0';
    char alpha[20];
    int j = 0;
    int betacount = 0;
    int betapointer = 0;
    char betaproduction[30];
    char newProduction[50];
    sprintf(newProduction, "%c-> ", production[0]);
    int k = 0;
    int newFlag = 0;
    //FINDING A->betaA
    for (int i = 3; i < strlen(production); i++)
    {
        if (production[i] == '|' || production[i] == '\n') //end of a production
        {
            strcat(newProduction, new);
            betacount++;
            newFlag = 1;
        }
        else if (production[i] != new[0])
        {
            if (newFlag == 1) //must concatenate |
            {
                strcat(newProduction, "|");
                newFlag = 0;
            }
            char temp[2];
            temp[0] = production[i];
            temp[1] = '\0';
            strcat(newProduction, temp); //Adding character of beta
        }
        else
        { //left recursion position so not beta
            while (production[i] != '|')
                i++;
        }
    }
    if (strlen(newProduction) != 4)
    {
        strcat(newProduction, "\n");
        strcpy(newFile[count++], newProduction);
    }
    else
    {
        strcat(newProduction, new);
        strcat(newProduction, "\n");
        strcpy(newFile[count++], newProduction);
    }
    //FINDING A'->epsilon|alphaA
    //finding alpha if there is more than one
    int alphapos[5];
    int alphacount = 0;
    for (int i = pos; i < strlen(production); i++)
    {
        if (production[i] == new[0] && (production[i - 1] == '|' || production[i - 1] == '>'))
            alphapos[alphacount++] = i;
    }
    for (int i = 0; i < alphacount; i++)
        j = 0;
    char alphaproduction[100];
    memset(alphaproduction, 0, 100);
    alphaproduction[0] = new[0];
    alphaproduction[1] = new[1];
    alphaproduction[2] = '-';
    alphaproduction[3] = '>';
    alphaproduction[4] = ' ';
    int pointer = 5;
    for (int i = 0; i < alphacount; i++)
    {
        j = alphapos[i] + 1;
        while (production[j] != '|' && production[j] != '\n')
        {
            alphaproduction[pointer] = production[j];
            pointer++;
            j++;
        }
        strcat(alphaproduction, new);
        pointer += 2;
        alphaproduction[pointer++] = '|';
    }
    strcat(alphaproduction, "e\n\0");
    sprintf(newFile[count++], "%s", alphaproduction);
    return count;
}

int recursiveDescent(char newProduction[10][128], int count)
{
    char instring[128];
    printf("===========================================\n");
    printf("Function Calls and Parsing:\n");
    printf("===========================================\n\n");
    printf("Enter the string to parse:");
    scanf("%s", instring);
    parsestring *p = malloc(sizeof(parsestring));
    strcpy(p->instring, instring);
    p->ptr = 0;
    printf("INSIDE MAIN\n");
    printf("ENTERED E\n");
    E(p,1);
    printf("EXITED E\n");
    if (p->instring[p->ptr] == '$')
    {
        printf("\nPARSING SUCCESSFUL!!!");
    }
    else
    {
        printf("ERROR!!");
        exit(0);
    }
}

void E(parsestring *p,int tab)
{
    for(int i=0;i<tab;i++)
        printf("\t");
        printf("ENTERED T\n");
    T(p,tab+1);
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("EXITED T\n");
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("ENTERED E'\n");
    E_Dash(p,tab+1);
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("EXITED E'\n");
}
void E_Dash(parsestring *p,int tab)
{
    if (p->instring[p->ptr] == '+')
    {
        //match +
        p->ptr += 1;
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("MATCHED +\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED T\n");
        T(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED T\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED E'\n");
        E_Dash(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED E'\n");
    }
    else
        return;
}
void T(parsestring *p,int tab)
{
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("ENTERED F\n");
    F(p,tab+1);
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("EXITED F\n");
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("ENTERED T'\n");
    T_Dash(p,tab+1);
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("EXITED T'\n");
}
void T_Dash(parsestring *p,int tab)
{
    if (p->instring[p->ptr] == '*')
    {
        //match *
        p->ptr += 1;
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("MATCHED *\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED F\n");
        F(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED F\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED E'\n");
        E_Dash(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED E'\n");
    }
    else
        return;
}
void F(parsestring *p,int tab)
{
    if (p->instring[p->ptr] == 'i')
    {
        //match i
        for(int i=0;i<tab;i++){
            printf("\t");
        }
        printf("MATCHED i\n");
        p->ptr += 1;
    }
    else
    {
        printf("ERROR!!");
        exit(0);
    }
}
```

2 Code for input 2

```c
#include "functions1.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct parsestring
{
    char instring[128];
    int ptr;
} parsestring;

int substr(char str1[], char str2[]);
int checkFunction(char str[]);
int EliminateLeftRecursion();
int Eliminate(char production[], char newFile[10][128], int count, int pos);
int recursiveDescent(char newProduction[10][128], int count);
int checkterminal(char ch);
void E(parsestring *p, int tab);
void E_Dash(parsestring *p, int tab);
void T(parsestring *p, int tab);
void T_Dash(parsestring *p, int tab);
void F(parsestring *p, int tab);

int main()
{
    EliminateLeftRecursion();
    return 0;
}

int EliminateLeftRecursion()
{
    char file[10][128];
    char newProduction[10][128];
    char functions[128][128];
    int newProductionCount = 0;
    int lrcount = 0;
    char inputfile[30];
    FILE *fd;
    fd = fopen("input2.txt", "r");
    int i = 0;
    //reading code from a file and storing in an array
    while (fgets(file[i], sizeof(file[i]), fd))
        i++;

    printf("===========================\n");
    printf("Input Productions:\n");
    printf("===========================\n");
    for (int j = 0; j < i; j++)
    {
        printf("%s", file[j]);
    }
    printf("\n\n");
    printf("===========================================\n");
    printf("Result of checking for Left Recursion:\n");
    printf("===========================================\n");
    for (int j = 0; j < i; j++)
    {
        char lhs = file[j][0];
        int noLeft = 1;
        for (int k = 3; k < strlen(file[j]); k++)
        {
            if (lhs == file[j][k]) //checking if left recursion occurs in the productions
            {
                noLeft = 0;
                Eliminate(file[j], newProduction, newProductionCount, k);
                newProductionCount += 2;
                lrcount++;
                break;
            }
        }
        if (noLeft == 1) //no LR so no change in the production
        {
            strcpy(newProduction[newProductionCount++], file[j]);
        }
    }
    if (lrcount == 0)
        printf("NO LEFT RECURSION\n");
    else
    {
        for (int j = 0; j < newProductionCount; j++)
        {
            printf("%s", newProduction[j]);
        }
    }
    printf("\n\n");
    recursiveDescent(newProduction, newProductionCount);
}

int Eliminate(char production[], char newFile[10][128], int count, int pos)
{
    char new[3];
    new[0] = production[0];
    new[1] = '\'';
    new[2] = '\0';
    char alpha[20];
    int j = 0;
    int betacount = 0;
    int betapointer = 0;
    char betaproduction[30];
    char newProduction[50];
    sprintf(newProduction, "%c-> ", production[0]);
    int k = 0;
    int newFlag = 0;
    //FINDING A->betaA
    for (int i = 3; i < strlen(production); i++)
    {
        if (production[i] == '|' || production[i] == '\n') //end of a production
        {
            strcat(newProduction, new);
            betacount++;
            newFlag = 1;
        }
        else if (production[i] != new[0])
        {
            if (newFlag == 1) //must concatenate |
            {
                strcat(newProduction, "|");
                newFlag = 0;
            }
            char temp[2];
            temp[0] = production[i];
            temp[1] = '\0';
            strcat(newProduction, temp); //Adding character of beta
        }
        else
        { //left recursion position so not beta
            while (production[i] != '|')
                i++;
        }
    }
    if (strlen(newProduction) != 4)
    {
        strcat(newProduction, "\n");
        strcpy(newFile[count++], newProduction);
    }
    else
    {
        strcat(newProduction, new);
        strcat(newProduction, "\n");
        strcpy(newFile[count++], newProduction);
    }
    //FINDING A'->epsilon|alphaA
    //finding alpha if there is more than one
    int alphapos[5];
    int alphacount = 0;
    for (int i = pos; i < strlen(production); i++)
    {
        if (production[i] == new[0] && (production[i - 1] == '|' || production[i - 1] == '>'
    ))
            alphapos[alphacount++] = i;
    }
    for (int i = 0; i < alphacount; i++)
        j = 0;
    char alphaproduction[100];
    memset(alphaproduction, 0, 100);
    alphaproduction[0] = new[0];
    alphaproduction[1] = new[1];
    alphaproduction[2] = '-';
    alphaproduction[3] = '>';
    alphaproduction[4] = ' ';
    int pointer = 5;
    for (int i = 0; i < alphacount; i++)
    {
        j = alphapos[i] + 1;
        while (production[j] != '|' && production[j] != '\n')
        {
            alphaproduction[pointer] = production[j];
            pointer++;
            j++;
        }
        strcat(alphaproduction, new);
        pointer += 2;
        alphaproduction[pointer++] = '|';
    }
    strcat(alphaproduction, "e\n\0");
    sprintf(newFile[count++], "%s", alphaproduction);
    return count;
}

int recursiveDescent(char newProduction[10][128], int count)
{
    char instring[128];
    printf("===========================================\n");
    printf("Function Calls and Parsing:\n");
    printf("===========================================\n\n");
    printf("Enter the string to parse:");
    scanf("%s", instring);
    parsestring *p = malloc(sizeof(parsestring));
    strcpy(p->instring, instring);
    p->ptr = 0;
    printf("INSIDE MAIN\n");
    printf("ENTERED E\n");
    E(p, 1);
    printf("EXITED E\n");
    if (p->instring[p->ptr] == '$')
    {
        printf("\nPARSING SUCCESSFUL!!!");
    }
    else
    {
        printf("ERROR!!");
        exit(0);
    }
}

void E(parsestring *p,int tab)
{
    for(int i=0;i<tab;i++)
        printf("\t");
        printf("ENTERED T\n");
    T(p,tab+1);
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("EXITED T\n");
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("ENTERED E'\n");
    E_Dash(p,tab+1);
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("EXITED E'\n");
}
void E_Dash(parsestring *p,int tab)
{
    if (p->instring[p->ptr] == '+')
    {
        //match +
        p->ptr += 1;
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("MATCHED +\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED T\n");
        T(p,tab+1);
        for(int i=0;i<tab;i++)
```

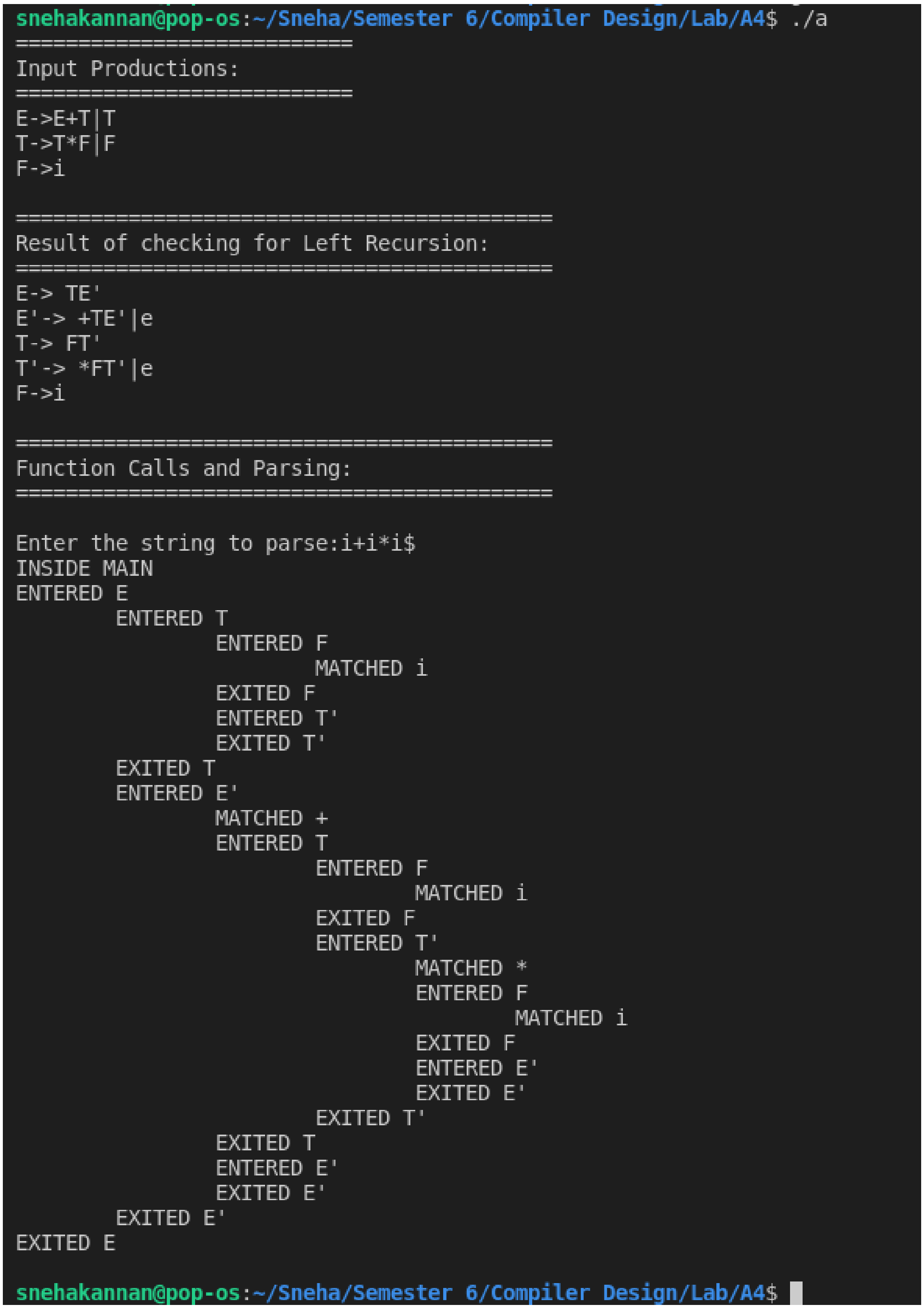
```c
            printf("\t");
        printf("EXITED T\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED E'\n");
        E_Dash(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED E'\n");
    }
    else if (p->instring[p->ptr] == '-')
    {
        //match +
        p->ptr += 1;
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("MATCHED -\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED T\n");
        T(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED T\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED E'\n");
        E_Dash(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED E'\n");
    }
    else
        return;
}
void T(parsestring *p,int tab)
{
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("ENTERED F\n");
    F(p,tab+1);
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("EXITED F\n");
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("ENTERED T'\n");
    T_Dash(p,tab+1);
    for(int i=0;i<tab;i++)
        printf("\t");
    printf("EXITED T'\n");
}
void T_Dash(parsestring *p,int tab)
{
    if (p->instring[p->ptr] == '*')
    {
        //match *
        p->ptr += 1;
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("MATCHED *\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED F\n");
        F(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED F\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED T'\n");
        T_Dash(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED T'\n");
    }
    else if (p->instring[p->ptr] == '/')
    {
        //match *
        p->ptr += 1;
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("MATCHED /\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED F\n");
        F(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED F\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED T'\n");
        T_Dash(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED T'\n");
    }
    else
        return;
}
void F(parsestring *p,int tab)
{
    if (p->instring[p->ptr] == '(')
    {
        //match *
        p->ptr += 1;
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("MATCHED (\n");
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("ENTERED E\n");
        E(p,tab+1);
        for(int i=0;i<tab;i++)
            printf("\t");
        printf("EXITED E\n");
        if (p->instring[p->ptr] == ')')
        {
            for(int i=0;i<tab;i++)
                printf("\t");
            printf("MATCHED )\n");
        }
        else{
            printf("ERROR!!");
            exit(0);
        }

    }
    else if (p->instring[p->ptr] == 'i')
    {
        //match i
        for(int i=0;i<tab;i++){
            printf("\t");
        }
        printf("MATCHED i\n");
        p->ptr += 1;
    }
    else
    {
        printf("ERROR!!");
        exit(0);
    }
}
```

3 Output Screenshots



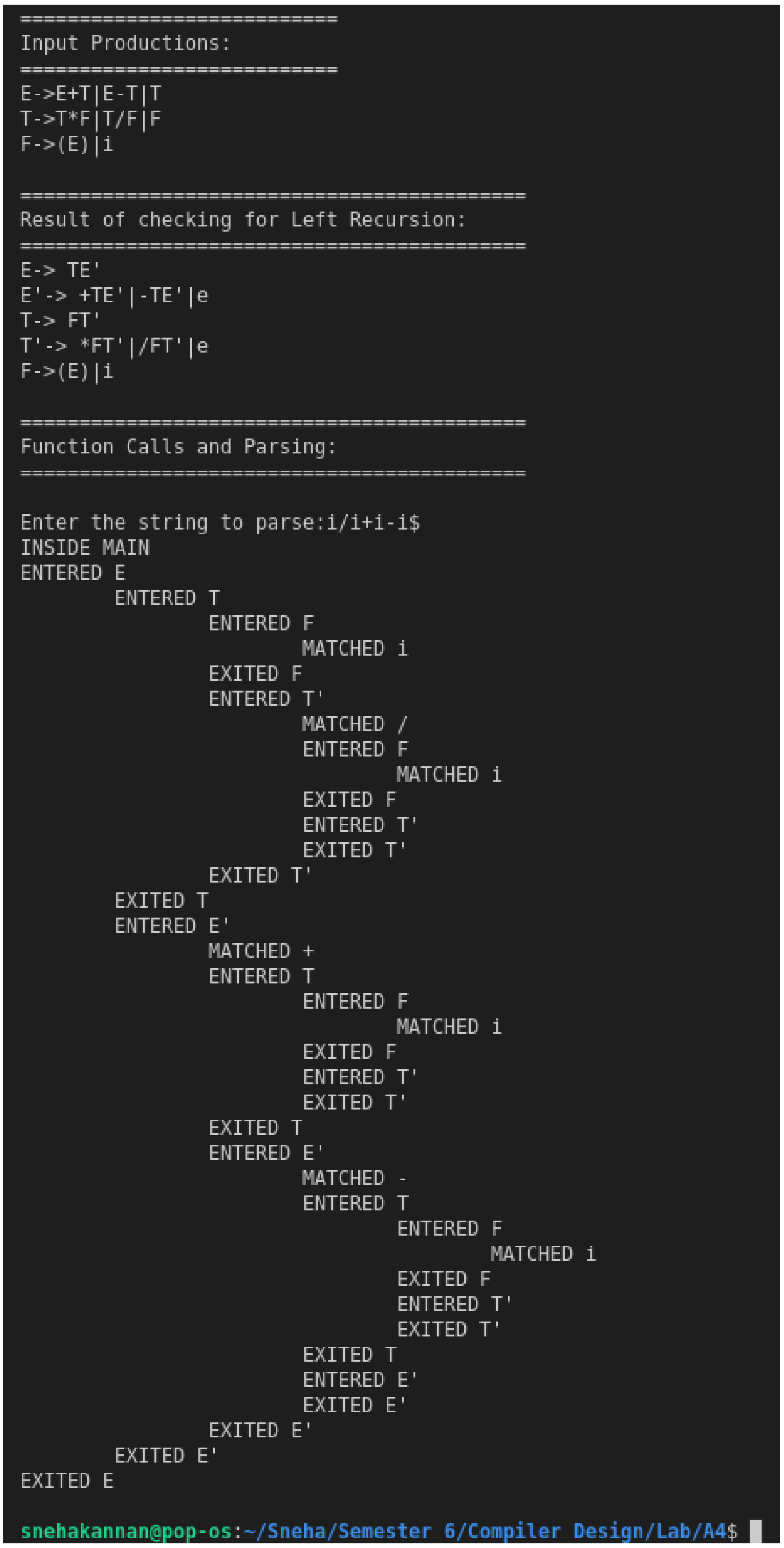
```
snehakannan@pop-os:~/Sneha/Semester 6/Compiler Design/Lab/A4$ ./a
===========================
Input Productions:
===========================
E->E+T|T
T->T*F|F
F->i

===========================================
Result of checking for Left Recursion:
===========================================
E-> TE'
E'-> +TE'|e
T-> FT'
T'-> *FT'|e
F->i

===========================================
Function Calls and Parsing:
===========================================

Enter the string to parse:i+i*i$
INSIDE MAIN
ENTERED E
        ENTERED T
                ENTERED F
                        MATCHED i
                EXITED F
                ENTERED T'
                EXITED T'
        EXITED T
        ENTERED E'
                MATCHED +
                ENTERED T
                        ENTERED F
                                MATCHED i
                        EXITED F
                        ENTERED T'
                                MATCHED *
                                ENTERED F
                                        MATCHED i
                                EXITED F
                                ENTERED E'
                                EXITED E'
                        EXITED T'
                EXITED T
                ENTERED E'
                EXITED E'
        EXITED E'
EXITED E

snehakannan@pop-os:~/Sneha/Semester 6/Compiler Design/Lab/A4$ 
```

Figure 1: Output for grammar 1 success case



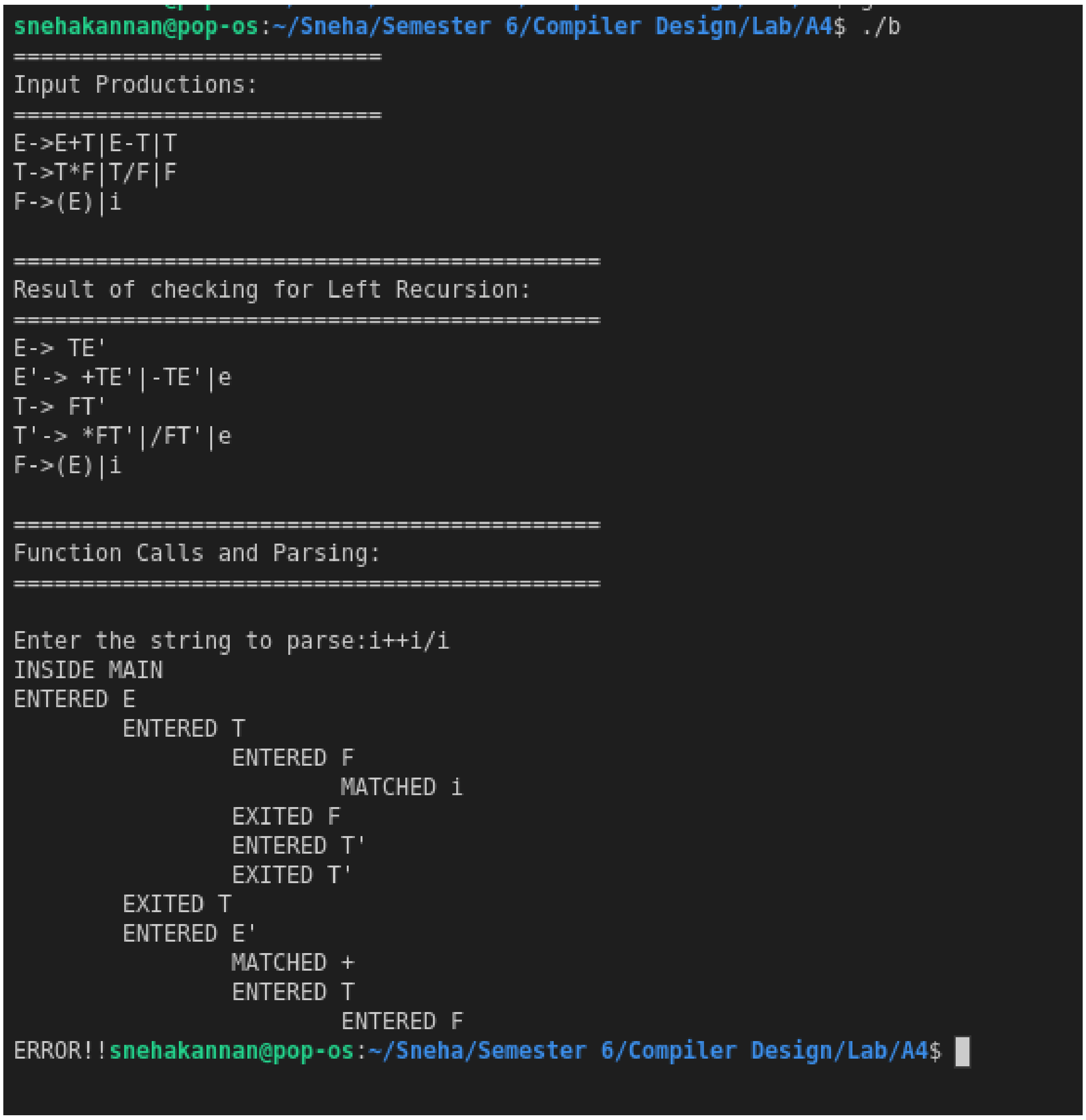
```
snehakannan@pop-os:~/Sneha/Semester 6/Compiler Design/Lab/A4$ ./a
===========================
Input Productions:
===========================
E->E+T|T
T->T*F|F
F->i

===========================================
Result of checking for Left Recursion:
===========================================
E-> TE'
E'-> +TE'|e
T-> FT'
T'-> *FT'|e
F->i

===========================================
Function Calls and Parsing:
===========================================

Enter the string to parse:i+i*i$
INSIDE MAIN
ENTERED E
        ENTERED T
                ENTERED F
                        MATCHED i
                EXITED F
                ENTERED T'
                EXITED T'
        EXITED T
        ENTERED E'
                MATCHED +
                ENTERED T
                        ENTERED F
                                MATCHED i
                        EXITED F
                        ENTERED T'
                                MATCHED *
                                ENTERED F
                                        MATCHED i
                                EXITED F
                                ENTERED E'
                                EXITED E'
                        EXITED T'
                EXITED T
                ENTERED E'
                EXITED E'
        EXITED E'
EXITED E

snehakannan@pop-os:~/Sneha/Semester 6/Compiler Design/Lab/A4$
```

Figure 2: Output for grammar 1 failure case



```
===========================
Input Productions:
===========================
E->E+T|E-T|T
T->T*F|T/F|F
F->(E)|i

===========================================
Result of checking for Left Recursion:
===========================================
E-> TE'
E'-> +TE'|-TE'|e
T-> FT'
T'-> *FT'|/FT'|e
F->(E)|i

===========================================
Function Calls and Parsing:
===========================================

Enter the string to parse:i/i+i-i$
INSIDE MAIN
ENTERED E
        ENTERED T
                ENTERED F
                        MATCHED i
                EXITED F
                ENTERED T'
                        MATCHED /
                        ENTERED F
                                MATCHED i
                        EXITED F
                        ENTERED T'
                        EXITED T'
                EXITED T'
        EXITED T
        ENTERED E'
                MATCHED +
                ENTERED T
                        ENTERED F
                                MATCHED i
                        EXITED F
                        ENTERED T'
                        EXITED T'
                EXITED T
                ENTERED E'
                        MATCHED -
                        ENTERED T
                                ENTERED F
                                        MATCHED i
                                EXITED F
                                ENTERED T'
                                EXITED T'
                        EXITED T
                        ENTERED E'
                        EXITED E'
                EXITED E'
        EXITED E'
EXITED E

snehakannan@pop-os:~/Sneha/Semester 6/Compiler Design/Lab/A4$
```

Figure 3: Output for grammar 2 success case



```
snehakannan@pop-os:~/Sneha/Semester 6/Compiler Design/Lab/A4$ ./b
===========================
Input Productions:
===========================
E->E+T|E-T|T
T->T*F|T/F|F
F->(E)|i

===========================================
Result of checking for Left Recursion:
===========================================
E-> TE'
E'-> +TE'|-TE'|e
T-> FT'
T'-> *FT'|/FT'|e
F->(E)|i

===========================================
Function Calls and Parsing:
===========================================

Enter the string to parse:i++i/i
INSIDE MAIN
ENTERED E
        ENTERED T
                ENTERED F
                        MATCHED i
                EXITED F
                ENTERED T'
                EXITED T'
        EXITED T
        ENTERED E'
                MATCHED +
                ENTERED T
                        ENTERED F
ERROR!!snehakannan@pop-os:~/Sneha/Semester 6/Compiler Design/Lab/A4$ 
```

Figure 4: Output for grammar 2 failure case