

Department of CSE SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VI

22 January 2021

UCS 1602 - Compiler Design

Exercise 1: Lexical Analyser Using C

Aim:

To write a program using C to perform the basic functionalities of a **Lexical Analyser**.

Code:

```
1 /* C Program that performs a basic lexical analysis of a given string */
2
3 #include <stdio.h>
4 #include <string.h>
5 #include <stdlib.h>
6 #include <ctype.h>
7 #include <unistd.h>
8 #include <fcntl.h>
9
10 int isOperator(char ch);
11 int isSeparator(char ch);
12 int isDelimiter(char ch);
13 int isValidIdentifier(char *str);
14 int isInteger(char *str);
15 int isKeyword(char *str);
16 int isPreprocessorDirective(char ch);
17 char *subString(char *str, int start, int end);
18 int printOperator(char ch1, char ch2);
19 int lexicalParse(char *str);
20
21 int main(void){
22     int status = 0, len, fp;
23     char text[10000], file[100];
24
25     printf("\n\t\t\tLexical Analyser Using C\n");
26     printf("\n\t\t\tEnter file name to parse: ");
27     scanf("%[^\n]", file);
28
29     fp = open(file, O_RDONLY);
30
31     if(fp < 0){
32         printf("\nError: File does not exist.\n");
33         return 0;
34     }
35
36     len = read(fp, text, 10000);
37     close(fp);
38
39     printf("\nText to be parsed:\n\n%s\n", text);
40
41     status = lexicalParse(text);
42
43     if(status){
44         printf("\n\n\t\tThe given expression is lexically valid.\n");
45     }
46
47     else{
```

```

48     printf("\n\n\t\tThe given expression is lexically invalid.\n");
49 }
50
51 return 0;
52 }
53
54 int isOperator(char ch){
55     //Checks if the character is a valid operator
56
57     if (ch == '+' || ch == '-' || ch == '*' ||
58         ch == '/' || ch == '>' || ch == '<' ||
59         ch == '=' || ch == '%' || ch == '!'){
60         return 1;
61     }
62
63     return 0;
64 }
65
66 int isSeparator(char ch){
67     //Checks if the character is a valid separator
68
69     if (ch == ';' || ch == '{' || ch == '}' || ch == ','){
70         return 1;
71     }
72
73     return 0;
74 }
75
76 int isDelimiter(char ch){
77     //Checks if the character is a valid delimiter
78
79     if (ch == ' ' || ch == '(' || ch == ')',
80         || isSeparator(ch) == 1 || isOperator(ch) == 1){
81         return 1;
82     }
83
84     return 0;
85 }
86
87 int isValidIdentifier(char *str){
88     //Checks if the character is a valid identifier
89
90     if(isdigit(str[0]) > 0 || isDelimiter(str[0]) == 1){
91         //First character shouldn't be a digit or a special character
92         return 0;
93     }
94
95     return 1;
96 }
97
98 int isInteger(char *str){

```

```

99     //Checks if the string is a valid integer
100
101     int i = 0, len = strlen(str);
102
103     if(!len){
104         return 0;
105     }
106
107     for(i = 0; i < len; i++){
108         if(!isdigit(str[i])){
109             return 0;
110         }
111     }
112
113     return 1;
114 }
115
116 int isKeyword(char *str){
117     //Checks if the string is a valid keyword
118
119     if(!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while")
120        ||
121        !strcmp(str, "for") || !strcmp(str, "do") || !strcmp(str, "break")
122        ||
123        !strcmp(str, "switch") || !strcmp(str, "continue") || !strcmp(str,
124        "return") ||
125        !strcmp(str, "case") || !strcmp(str, "default") || !strcmp(str, "
126        void") ||
127        !strcmp(str, "int") || !strcmp(str, "char") || !strcmp(str, "bool"
128        ) ||
129        !strcmp(str, "struct") || !strcmp(str, "goto") || !strcmp(str, "
130        typedef") ||
131        !strcmp(str, "unsigned") || !strcmp(str, "long") || !strcmp(str, "
132        short") ||
133        !strcmp(str, "float") || !strcmp(str, "double") || !strcmp(str, "
134        sizeof")){
135         return 1;
136     }
137
138     return 0;
139 }
140
141 int isPreprocessorDirective(char ch){
142     //Checks if the string is a valid preprocessor directive
143
144     if(ch == '#'){
145         //Basic check, works for header files, macros and const
146         declarations
147         return 1;
148     }
149
150     return 0;

```

```

141 }
142
143 char *subString(char *str, int start, int end){
144     //Get a substring from the given string
145     int i = 0;
146     char *sub = (char *)malloc(sizeof(char) * (end - start + 2));
147
148     for(i = start; i <= end; i++){
149         sub[i - start] = str[i];
150     }
151
152     sub[end - start + 1] = '\0';
153
154     return sub;
155 }
156
157 int printOperator(char ch1, char ch2){
158     //Print the details of the parsed operator
159
160     switch(ch1){
161         case '+':
162             if(ch2 == '='){
163                 printf("ASSIGN ");
164             }
165             else if(ch2 == ' '){
166                 printf("ADD ");
167             }
168             else{
169                 printf("INVALID-OP ");
170                 return 0;
171             }
172             break;
173
174
175         case '-':
176             if(ch2 == '='){
177                 printf("SUB-ASSIGN ");
178             }
179             else if(ch2 == ' '){
180                 printf("SUB ");
181             }
182             else{
183                 printf("INVALID-OP ");
184                 return 0;
185             }
186             break;
187
188         case '*':
189             if(ch2 == '='){
190                 printf("PRODUCT-ASSIGN ");
191             }

```

```

192     else if(ch2 == ' '){
193         printf("PRODUCT ");
194     }
195     else{
196         printf("INVALID-OP");
197         return 0;
198     }
199     break;
200
201 case '/':
202     if(ch2 == '='){
203         printf("DIVISION-ASSIGN ");
204     }
205     else if(ch2 == ' '){
206         printf("DIVISION ");
207     }
208     else{
209         printf("INVALID-OP ");
210         return 0;
211     }
212     break;
213
214 case '%':
215     if(ch2 == '='){
216         printf("MODULO-ASSIGN ");
217     }
218     else if(ch2 == ' '){
219         printf("MODULO ");
220     }
221     else{
222         printf("INVALID-OP ");
223         return 0;
224     }
225     break;
226
227 case '=':
228     if(ch2 == '='){
229         printf("EQUALITY ");
230     }
231     else if(ch2 == ' '){
232         printf("ASSIGN ");
233     }
234     else{
235         printf("INVALID-OP ");
236         return 0;
237     }
238     break;
239
240 case '>':
241     if(ch2 == '='){
242         printf("GT-EQ ");

```

```

243     }
244     else if(ch2 == ' '){
245         printf("GT ");
246     }
247     else{
248         printf("INVALID-OP ");
249         return 0;
250     }
251     break;
252
253     case '<':
254         if(ch2 == '='){
255             printf("LT-EQ ");
256         }
257         else if(ch2 == ' '){
258             printf("LT ");
259         }
260         else{
261             printf("INVALID-OP ");
262             return 0;
263         }
264         break;
265
266     case '!':
267         printf("NOT ");
268         break;
269
270     default:
271         printf("INVALID-OP ");
272         return 0;
273 }
274
275 return 1;
276 }
277
278 int lexicalParse(char *str){
279     //Parse the given string to check for validity
280     int left = 0, right = 0, len = strlen(str), status = 1, i;
281
282     printf("\nLexical Analysis:\n\t");
283
284     while(right <= len && left <= right){
285         //While we are within the valid bounds of the string, check:
286
287         while(isPreprocessorDirective(str[right]) == 1){
288             //Check if string is preprocessor directive
289             printf("PPDIR ");
290
291             for(right; str[right] != '\n'; right++);
292             right++;
293             left = right;

```

```

294     }
295
296     for(i = right; i < len; i++){
297         //Clearing linebreaks & tabs to spaces for efficient
processing
298         if(str[i] == '\n' || str[i] == '\t'){
299             str[i] = ' ';
300         }
301     }
302
303     if(isDelimiter(str[right]) == 0){
304         //If we do not encounter a delimiter, keep moving forward
305         //"right" points to the next character
306         right++;
307     }
308
309     else if(isDelimiter(str[right]) == 1 && left == right){
310         //If it is a delimiter, and we haven't parsed it yet
311
312         if(isSeparator(str[right]) == 1){
313             //Check if the delimiter is a separator
314             printf("SP ");
315         }
316
317         else if(isOperator(str[right]) == 1){
318             //Check if the delimiter is an operator
319             if((right + 1) <= len && isOperator(str[right + 1]) == 1){
320                 //Check if the next character is also an operator
321                 status = status & printOperator(str[right], str[right
+ 1]);
322                 right++;
323             }
324
325             else{
326                 //Next character is not an operator
327                 status = status & printOperator(str[right], ' ');
328             }
329
330             //printf("\n\t\t%c' is an operator.", str[right]);
331         }
332
333         right++;
334         left = right;
335     }
336
337     else if(str[right] == '(' && left != right || (right == len &&
left != right)){
338         //Special case, to check for functions
339
340         char *sub = subString(str, left, right - 1);
341

```



```

342         if(isKeyword(sub) == 1){
343             //Check if the function is a keyword based function, like
"if" & "for"
344             printf("KW ");
345             left = right;
346             continue; //Go ahead with the next check
347         }
348
349         //Otherwise, its some other function, parse it.
350
351         for(i = right + 1; i < len; i++){
352             if(str[i] == ')'){
353                 //Finish parsing till the end of the block and break
354                 printf("FUNCT ");
355                 right = i + 1;
356                 left = right;
357                 status = status & 1;
358                 break;
359             }
360         }
361     }
362
363     else if(isDelimiter(str[right]) == 1 && left != right || (right ==
len && left != right)){
364         //We encountered a delimiter in the "right" position, but left
!= right
365         //thus a chunk of unparsed characters exist between left and
right
366
367         //Make a substring of the unparsed characters
368         char *sub = subString(str, left, right - 1);
369
370         if(isInteger(sub) == 1){
371             //Check if substring is an integer
372             printf("NUMCONST ");
373         }
374         else if(isKeyword(sub) == 1){
375             //Check if substring is a keyword
376             printf("KW ");
377         }
378         else if(isValidIdentifier(sub) == 1){
379             //Check if substring is a valid identifier
380             printf("ID ");
381         }
382         else if(isValidIdentifier(sub) == 0 && isDelimiter(str[right -
1]) == 0){
383             //Otherwise, print that it is not a valid identifier
384             status = status & 0;
385             printf("INVALID-ID");
386         }
387

```

```
388         left = right;    //We have parsed the chunk, thus "left" = "  
    right"  
389     }  
390  
391 }  
392  
393     return status;  
394 }
```

Output - Valid Case:

Figure 1: Console Output for a Valid Program.

```
vishakan@Legion:~/Desktop/Compiler Design/Ex01
> gcc Lex.c -o l
> ./l

          Lexical Analyser Using C

Enter file name to parse: Sample.c

Text to be parsed:

#include<stdio.h>
#include<stdlib.h>

int main(){
    int a, b;
    printf("Hello");

    a = b + 100;

    if(a > b){
        printf("Greater");
    }

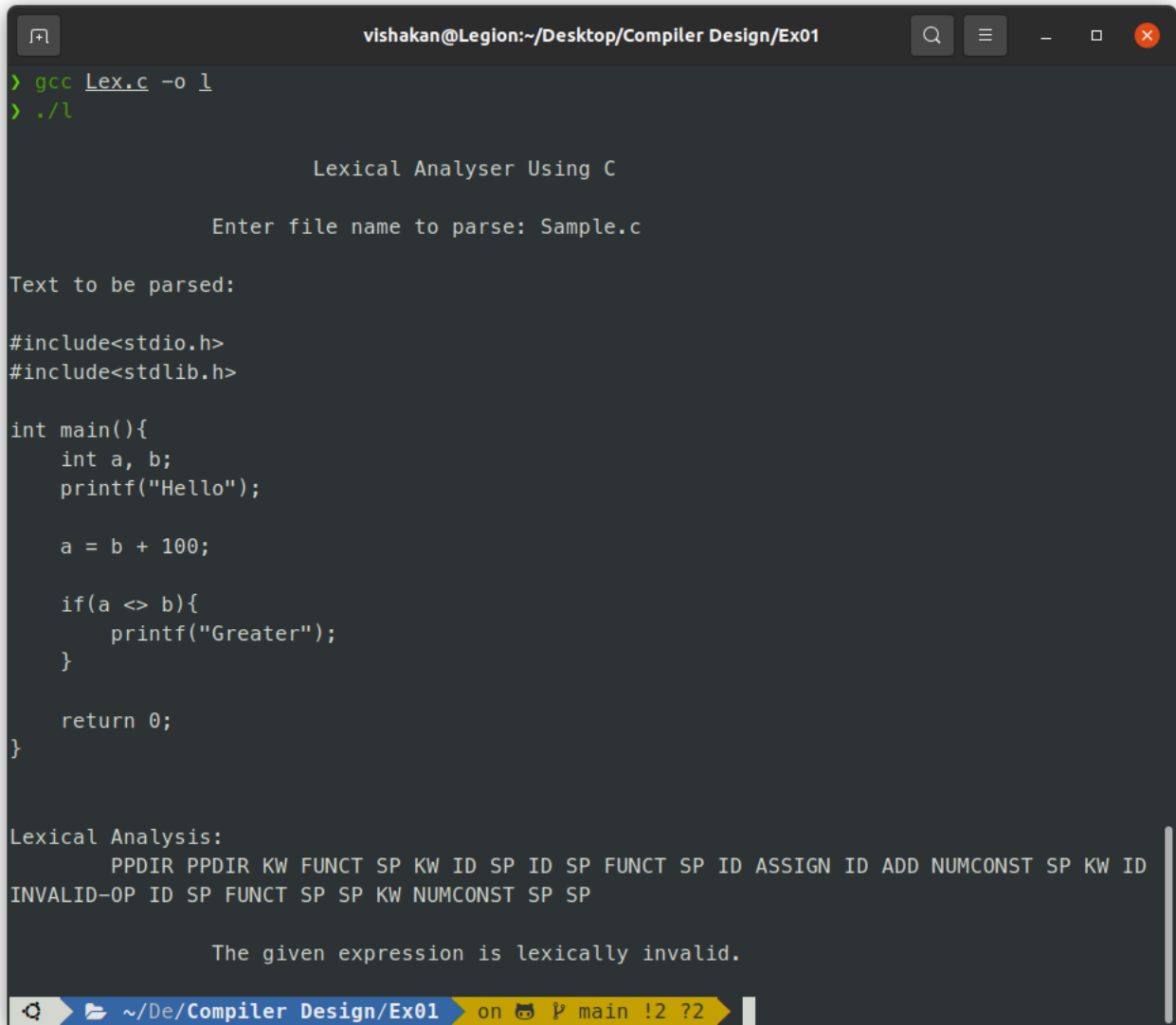
    return 0;
}

Lexical Analysis:
      PPDIR PPDIR KW FUNCT SP KW ID SP ID SP FUNCT SP ID ASSIGN ID ADD NUMCONST SP KW ID
GT ID SP FUNCT SP SP KW NUMCONST SP SP

          The given expression is lexically valid.
```

Output - Invalid Case:

Figure 2: Console Output for an Invalid Program.



```
vishakan@Legion:~/Desktop/Compiler Design/Ex01
> gcc Lex.c -o l
> ./l

          Lexical Analyser Using C

Enter file name to parse: Sample.c

Text to be parsed:

#include<stdio.h>
#include<stdlib.h>

int main(){
    int a, b;
    printf("Hello");

    a = b + 100;

    if(a <> b){
        printf("Greater");
    }

    return 0;
}

Lexical Analysis:
PPDIR PPDIR KW FUNCT SP KW ID SP ID SP FUNCT SP ID ASSIGN ID ADD NUMCONST SP KW ID
INVALID-OP ID SP FUNCT SP SP KW NUMCONST SP SP

          The given expression is lexically invalid.
```

Learning Outcome:

- From the experiment, I understood how a basic **Lexical Analyser** works.
- I was able to formulate ideas on how to implement recognition of specific tokens in programs for identification by the Lexical Analyser.
- I was able to implement simple regular expressions in C.
- I learnt how to parse a program for lexical validity, utilising the concept of **lexemes**.
- I was able to visualize the complexity that goes behind the compilation process and the significance of a Lexical Analyser phase in the compilation flow.

Department of CSE

SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VI

12 February 2021

UCS 1602 - Compiler Design

Exercise 2: Lexical Analyser Using Lex Tool

Aim:

To write a program using Lex to perform the basic functionalities of a **Lexical Analyser**, and to form a symbol table on the parsed program.

Code:

```
1 /* Lexical Analyser Using Lex Tool */
2
3 /*Definitions*/
4
5 %{
6 #include<stdio.h>
7 #include<stdlib.h>
8 #include<string.h>
9
10 struct symbol{
11     char type[10];
12     char name[20];
13     char value[100];
14 }; //For Symbol Table
15
16 typedef struct symbol sym;
17
18 sym sym_table[1000];
19 int cur_size = -1;
20 char current_type[10];
21 %}
22
23 number_const [-+]?[0-9]+(\.[0-9]+)?
24 char_const \'.\'
25 string_const \".*\"
26 identifier [a-zA-Z_][a-zA-Z0-9_]*
27 function [a-zA-Z_][a-zA-Z0-9]*([.][.][.])
28 keyword (int|float|char|unsigned|typedef|struct|return|continue|break|if|
        else|for|while|do|extern|auto|case|switch|enum|goto|long|double|sizeof|
        void|default|register)
29 pp_dir ~[#].*[>]$
30 rel_ops (<|>|<=|>=|==|!=)
31 assign_ops (=|+=|-=|%=|/=|*=)
32 arith_ops (+|-|%|/|*)
33 single_cmt [/][/].*
34 multi_cmt ([/][/].*)|([/][*](.[\n\r])*[*][/])
35 spl_chars [{ } ( ) , ; \ [ \ ]
36
37 /*Rules*/
38
39 %%
40
41 {pp_dir} {
42     printf("PPDIR ");
43     strcpy(current_type, "INVALID");
44 }
45
```

```

46 {keyword} {
47     printf("KW ");
48
49     if(strcmp(yytext, "int") == 0){
50         strcpy(current_type, "int");
51     }
52     else if(strcmp(yytext, "float") == 0){
53         strcpy(current_type, "float");
54     }
55     else if(strcmp(yytext, "double") == 0){
56         strcpy(current_type, "double");
57     }
58     else if(strcmp(yytext, "char") == 0){
59         strcpy(current_type, "char");
60     }
61     else{
62         strcpy(current_type, "INVALID");
63     }
64 }
65
66 {function} {
67     printf("FUNCT ");
68 }
69
70 {identifier} {
71     printf("ID ");
72
73     if(strcmp(current_type, "INVALID") != 0){
74         cur_size++;
75         strcpy(sym_table[cur_size].name, yytext);
76         strcpy(sym_table[cur_size].type, current_type);
77
78         if(strcmp(current_type, "char") == 0){
79             strcpy(sym_table[cur_size].value, "NULL");
80         }
81         else if(strcmp(current_type, "int") == 0){
82             strcpy(sym_table[cur_size].value, "0");
83         }
84         else{
85             strcpy(sym_table[cur_size].value, "0.0");
86         }
87     }
88 }
89
90 {single_cmt} {
91     printf("SCMT ");
92 }
93
94 {multi_cmt} {
95     printf("MCMT ");
96 }

```



```

97
98 {number_const} {
99     printf("NUM_CONST ");
100
101     if(strcmp(current_type, "INVALID") != 0){
102         strcpy(sym_table[cur_size].value, yytext);
103     }
104 }
105
106 {char_const} {
107     printf("CHAR_CONST ");
108
109     if(strcmp(current_type, "char") == 0){
110         strcpy(sym_table[cur_size].value, yytext);
111     }
112 }
113
114 {string_const} {
115     printf("STR_CONST ");
116 }
117
118 {rel_ops} {
119     printf("REL_OP ");
120 }
121
122 {arith_ops} {
123     printf("ARITH_OP ");
124 }
125
126 {assign_ops} {
127     printf("ASSIGN_OP ");
128 }
129
130 {spl_chars} {
131     if(strcmp(yytext, ";") == 0){
132         strcpy(current_type, "INVALID");
133     }
134
135 }
136
137 \n {
138     printf("\n");
139 }
140
141 [ \t] { }
142
143
144 %%
145
146 int yywrap(void){
147     return 1;

```

```

148 }
149
150
151 /*User Subroutines*/
152
153 int main(int argc, char *argv[]){
154     int i = 0;
155
156     yyin = fopen(argv[1], "r");
157     yylex();
158
159     printf("\n\t-----\n");
160
161     printf("\n\t\t\tSYMBOL TABLE");
162     printf("\n\t\tNAME\tTYPE\tVALUE\n");
163     for(i = 0; i <= cur_size; i++){
164         printf("\t\t%s\t%s\t%s\n", sym_table[i].name, sym_table[i].type,
sym_table[i].value);
165     }
166
167     printf("\t-----\n");
168
169     return 0;
170 }

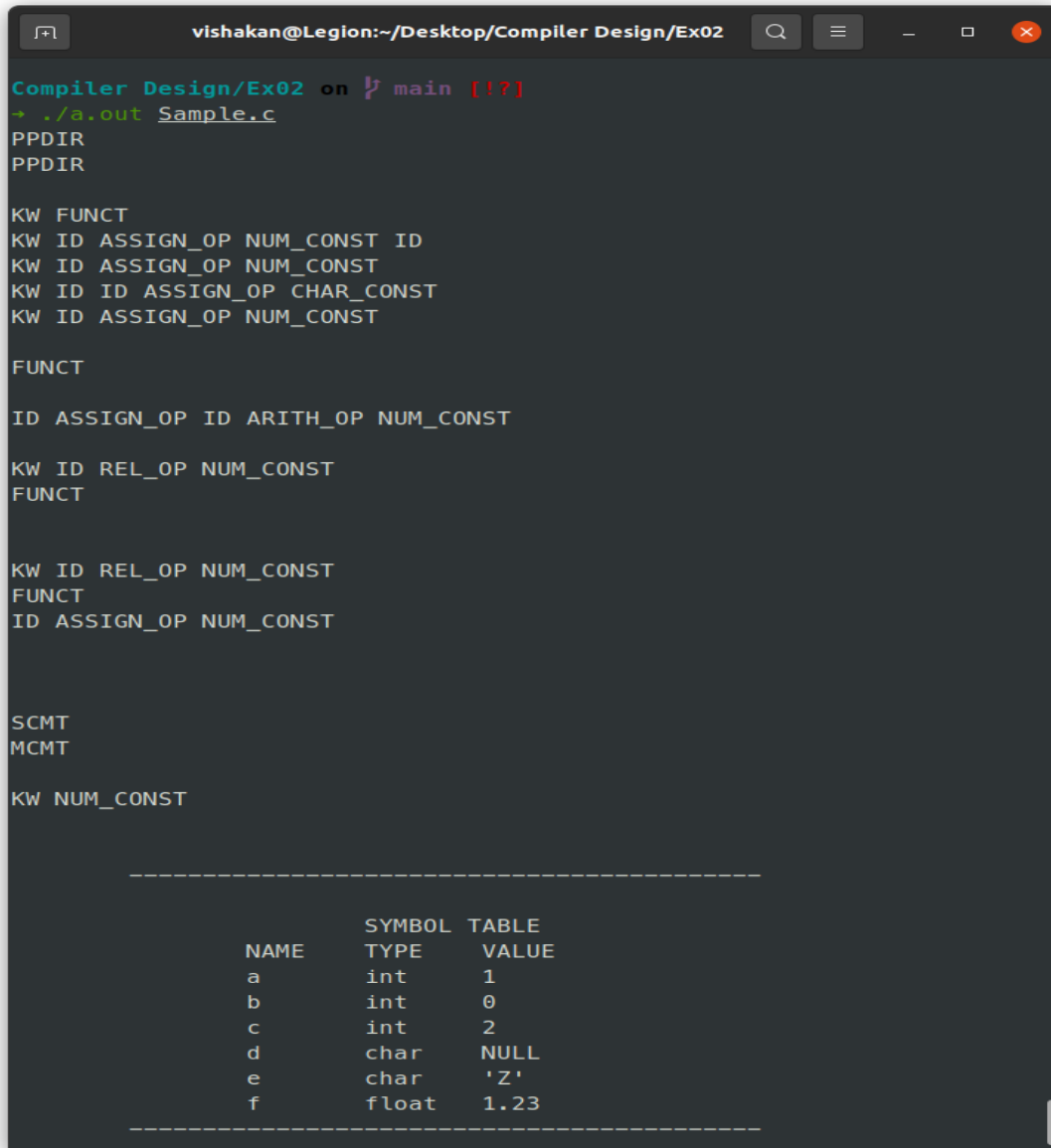
```

Parsed C Code:

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main(){
5     int a = 1, b;
6     int c = 2;
7     char d, e = 'Z';
8     float f = 1.23;
9
10    printf("Hello to %d", c);
11
12    a = b + 100;
13
14    if (c > 100){
15        printf("Greater");
16    }
17
18    while (c > 0) {
19        printf("Hello to Lex!");
20        c -= 1;
21    }
22
23
24    //a is GREATER than b!
25    /* Multi-line
26    comment */
27
28    return 0;
29 }
```

Output:

Figure 1: Console Output



```
vishakan@Legion:~/Desktop/Compiler Design/Ex02
Compiler Design/Ex02 on main [!?]
→ ./a.out Sample.c
PPDIR
PPDIR

KW FUNCT
KW ID ASSIGN_OP NUM_CONST ID
KW ID ASSIGN_OP NUM_CONST
KW ID ID ASSIGN_OP CHAR_CONST
KW ID ASSIGN_OP NUM_CONST

FUNCT

ID ASSIGN_OP ID ARITH_OP NUM_CONST

KW ID REL_OP NUM_CONST
FUNCT

KW ID REL_OP NUM_CONST
FUNCT
ID ASSIGN_OP NUM_CONST

SCMT
MCMT

KW NUM_CONST

-----
              SYMBOL TABLE
        NAME  TYPE  VALUE
        a    int   1
        b    int   0
        c    int   2
        d    char  NULL
        e    char  'Z'
        f    float 1.23
-----
```

Learning Outcome:

- From the experiment, I understood the basics of Lex tool.
- I was able to implement recognition for regular expressions using Lex terminology.
- I understood the working of a Lex program.
- I learnt about the three sections of a Lex program, namely, definitions, rules and user subroutines.
- I learnt to implement a basic symbol table using Lex on the parsed C program.
- I understood that Lex tool is more powerful and easy-to-use for Lexical Analysis task compared to conventional C programming.

Department of CSE

SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VI

20 February 2021

UCS 1602 - Compiler Design

Exercise 3: Elimination of Left Recursion Using C

Aim:

Write a program in C to find whether the given grammar is **Left Recursive** or not. If it is found to be left recursive, convert the grammar in such a way that the left recursion is removed.

Code:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(){
6     /*
7         Sample Input Format:      E->E+T|T
8                                   T->T*F|F
9                                   F->i
10    */
11
12    char productions[100][100], sub_prods[100][100];
13    char non_terminal;
14    int num_prods, i, j, k, flag = 0;
15
16    printf("\n\t\tElimination of Left Recursion\n");
17    printf("\nEnter the number of Productions: ");
18    scanf("%d", &num_prods);
19
20    printf("\nEnter the Grammar:\n");
21
22    for(i = 0; i < num_prods; i++){
23        //Getting Input
24        scanf("%s", productions[i]);
25    }
26
27    printf("\nGiven Grammar:\n");
28
29    for(i = 0; i < num_prods; i++){
30        //Printing the Grammar, and checking for left recursions
31        printf("%s\n", productions[i]);
32
33        if(productions[i][0] == productions[i][3]){
34            flag = 1;
35        }
36    }
37
38    if(flag == 0){
39        //If Grammar is not left recursive, exit
40        printf("\nGrammar is not Left Recursive.");
41        return 0;
42    }
43
44    //Otherwise, Grammar is left recursive, parse and remove it
45    printf("\nGrammar is Left Recursive.");
46    printf("\n\nGrammar after removal of Left Recursion:");
47
```

```

48     for(i = 0; i < num_prods; i++){
49         //Parse each production one by one
50         non_terminal = productions[i][0];
51
52         char *split, production[100];
53         flag = 0;
54
55         //Store the RHS of the production alone
56         for(j = 0; productions[i][j + 3] != '\0'; j++){
57             production[j] = productions[i][j + 3];
58         }
59
60         production[j] = '\0';
61         j = 0;
62
63         //Split at the sub-expression level when there is an OR operator
64         split = strtok(production, "|");
65
66         while(split != NULL){
67             //Store the subexpression in a new productions array
68             strcpy(sub_prods[j], split);
69
70             if(split[0] == non_terminal && flag == 0){
71                 //Seeing an immediate left recursion, with no other
72                 productions //for the same non-terminal
73                             //This type of Left Recursion cannot be removed
74                             flag = 1;
75             }
76             else if(split[0] != non_terminal && flag == 1){
77                 //Already seen a left recursion, but now we have seen
78                 //another production with some terminal symbol
79                 //for the same non-terminal
80                 flag = 2;
81             }
82
83             j++;
84             split = strtok(NULL, "|");
85             //split and loop till all productions are parsed
86         }
87
88         if(flag != 2){
89             //flag == 0 => no LR
90             //flag == 1 => LR of the form A->Ab which cannot be removed
91             printf("%s\n", productions[i]);
92         }
93
94         if(flag == 2){
95             //Remove the left recursion if there's another production with
96             terminal symbol
97             printf("\n");

```



```

97         flag = 0;
98
99         for(k = 0; k < j; k++){
100             if(sub_prods[k][0] != non_terminal){
101                 //Loop until the non-terminal causing the LR is not
found, for 1st production rule
102                 if(flag != 0){
103                     //Removed the LR by starting with the other non-
terminal/ID,
104                     //thus add the remaining sub-productions
105                     printf("|s%c'", sub_prods[k], non_terminal);
106                 }
107                 else{
108                     //No left recursion with that particular sub-
production
109                     //thus make it as a new production with a new non-
terminal
110                     flag = 1;
111                     printf("%c->s%c'", non_terminal, sub_prods[k],
non_terminal);
112                 }
113             }
114         }
115         printf("\n");
116         flag = 0;
117
118         for(k = 0; k < j; k++){
119             if(sub_prods[k][0] == non_terminal){
120                 //Loop until the non-terminal causing the LR is found,
for 2nd production rule
121                 if(flag != 0){
122                     //Add the remaining sub-productions, since the LR
has been removed
123                     printf("|s%c'", sub_prods[k] + 1, non_terminal);
124                 }
125                 else{
126                     //k sub-production contains the LR causing term,
thus first print the
127                     //next sub-production followed by a new non-
terminal as a new production
128                     //2D Array Manipulation, sub_prods[k] + 1
essentially prints
129                     //the string sub_prods[k][1] till sub_prods[k][n]
130                     flag = 1;
131                     printf("%c'->s%c'", non_terminal, sub_prods[k]
+ 1, non_terminal);
132                 }
133             }
134         }
135         printf("|e\n");
136     }

```

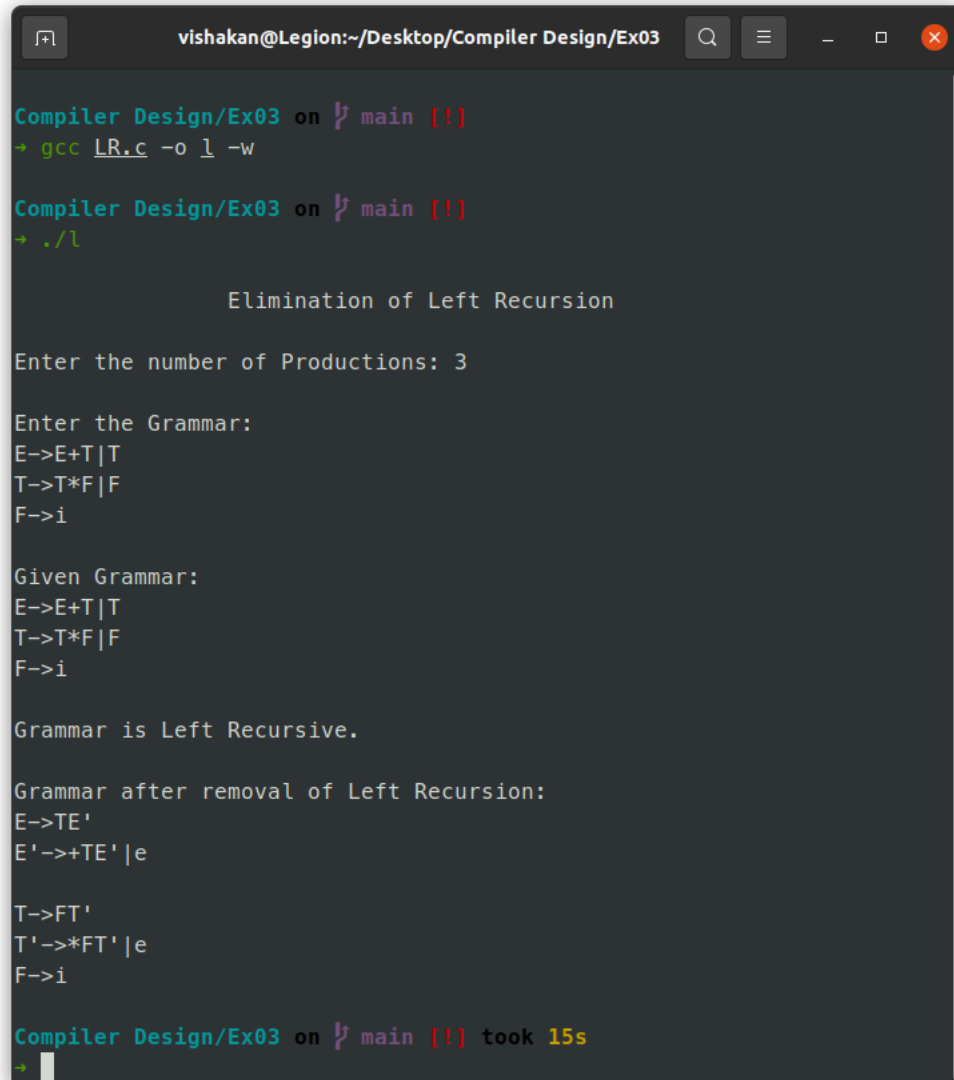
```

137
138     }
139
140
141     return 0;
142 }
143
144 /*
145 OUTPUT:
146
147 gcc LR.c -o l -w
148 ./l
149
150         Elimination of Left Recursion
151
152 Enter the number of Productions: 3
153
154 Enter the Grammar:
155 E->E+T|T
156 T->T*F|F
157 F->i
158
159 Given Grammar:
160 E->E+T|T
161 T->T*F|F
162 F->i
163
164 Grammar is Left Recursive.
165
166 Grammar after removal of Left Recursion:
167 E->TE'
168 E'->+TE'|e
169
170 T->FT'
171 T'->*FT'|e
172 F->i
173
174 */

```

Output - Left Recursive Grammar:

Figure 1: Console Output for a Left Recursive Grammar.



```
vishakan@Legion:~/Desktop/Compiler Design/Ex03
Compiler Design/Ex03 on 1 main [!]
+ gcc LR.c -o 1 -w
Compiler Design/Ex03 on 1 main [!]
+ ./1

      Elimination of Left Recursion

Enter the number of Productions: 3

Enter the Grammar:
E->E+T|T
T->T*F|F
F->i

Given Grammar:
E->E+T|T
T->T*F|F
F->i

Grammar is Left Recursive.

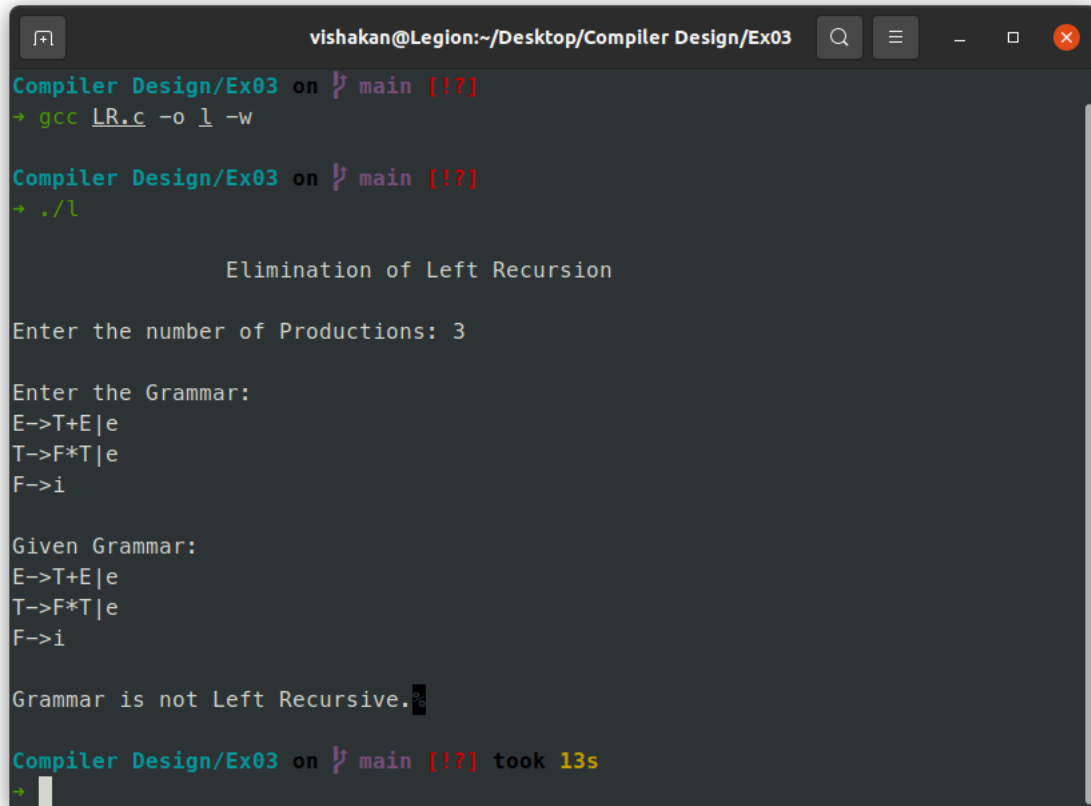
Grammar after removal of Left Recursion:
E->TE'
E'->+TE'|e

T->FT'
T'->*FT'|e
F->i

Compiler Design/Ex03 on 1 main [!] took 15s
+ 
```

Output - Non Left Recursive Grammar:

Figure 2: Console Output for a Non Left Recursive Grammar.



```
vishakan@Legion:~/Desktop/Compiler Design/Ex03
Compiler Design/Ex03 on  main [!?]
+ gcc LR.c -o l -w

Compiler Design/Ex03 on  main [!?]
+ ./l

          Elimination of Left Recursion

Enter the number of Productions: 3

Enter the Grammar:
E->T+E|e
T->F*T|e
F->i

Given Grammar:
E->T+E|e
T->F*T|e
F->i

Grammar is not Left Recursive.

Compiler Design/Ex03 on  main [!?] took 13s
+ 
```

Learning Outcome:

- I understood about left recursive grammars.
- I understood the need for this type of conversion, as top-down parsers cannot handle left recursive grammars.
- I was able to perform a check of whether or not a grammar is left recursive using C.
- I implemented a conversion in C which converts left recursive grammar to non left recursive grammar.
- I refreshed my 2D-char array manipulation concepts in C.

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:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow i \end{aligned}$$

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

G2:

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow (E) \mid i \end{aligned}$$

Code - Grammar 1:

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<string.h>
4
5 /*Recursive Descent Parser*/
6
7 /*
8 Grammar: G: E->E+T|T
9           T->T*F|F
10          F->i
11 */
12
13 /*
14 Removed Left Recursion
15 Grammar G': E->TE'
16             E'->+TE'|e
17             T->FT'
18             T'->*FT'|e
19             F->i
20 */
21
22 struct parse_struct{
23     char str[100];
24     int pos;
25     int len;
26 };
27
28 typedef struct parse_struct parser;
29
30 parser E(parser p);
31 parser T(parser p);
32 parser EPrime(parser p);
33 parser F(parser p);
34 parser TPrime(parser p);
35 parser parse(parser p, char s);
36
37 int main(void){
38     parser p;
39
40     printf("\n\t\tRecursive Descent Parser\n");
41     printf("\nEnter a string to parse: ");
42     scanf("%s", p.str);
43
44     p.len = strlen(p.str);
45     p.pos = 0;
46
47     p = E(p);
```

```

48
49     if(p.pos == p.len){
50         //All characters have been parsed
51         printf("\nParse Success!\n");
52     }
53
54     else{
55         //Some characters haven't been parsed, but returned to main
56         printf("\nError parsing at Position %d!\n", p.pos);
57     }
58
59
60     return 0;
61 }
62
63 parser E(parser p){
64     //printf("\nAt E");
65     p = T(p);
66     p = EPrime(p);
67
68     return p;
69 }
70
71 parser T(parser p){
72     //printf("\nAt T");
73     p = F(p);
74     p = TPrime(p);
75
76     return p;
77 }
78
79 parser EPrime(parser p){
80     //printf("\nAt EPrime");
81     if(p.str[p.pos] == '+'){
82         p = parse(p, '+');
83         p = T(p);
84         p = EPrime(p);
85     }
86
87     return p;
88 }
89
90 parser TPrime(parser p){
91     //printf("\nAt TPrime");
92     if(p.str[p.pos] == '*'){
93         p = parse(p, '*');
94         p = F(p);
95         p = TPrime(p);
96     }
97
98     return p;

```



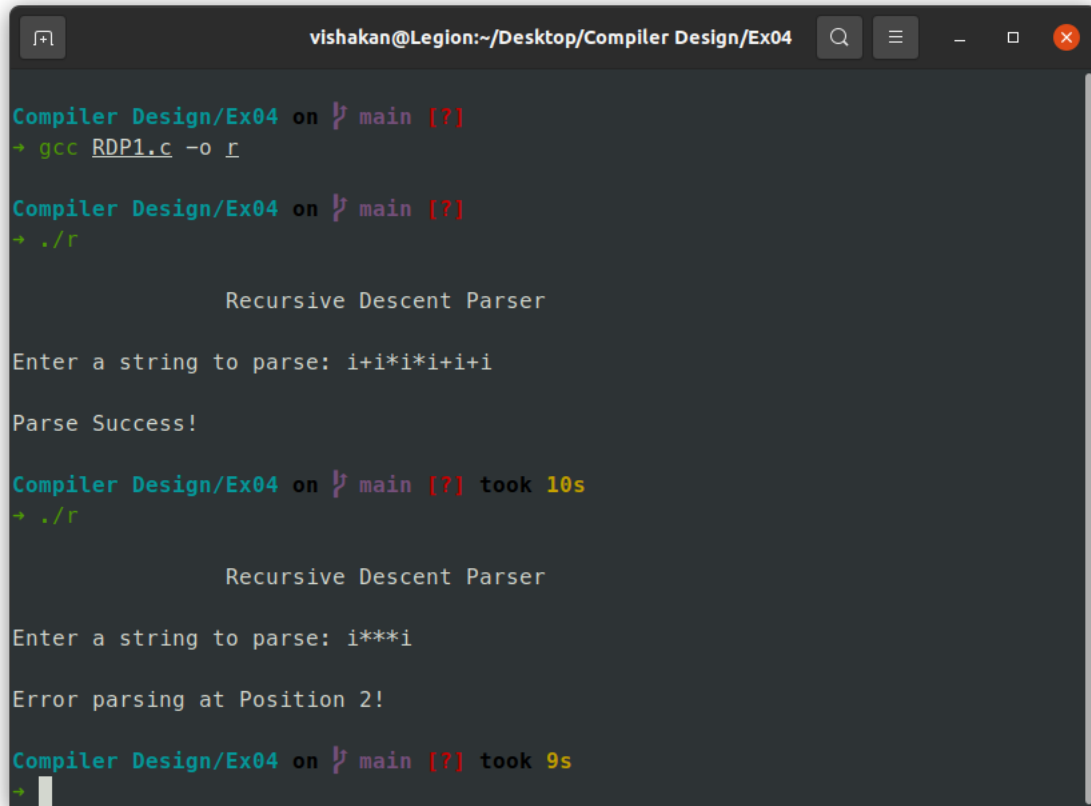
```

99 }
100
101 parser F(parser p){
102     //printf("\nAt F");
103     if(p.str[p.pos] == 'i'){
104         p = parse(p, 'i');
105     }
106     else{
107         printf("\nError parsing at Position %d!\n", p.pos);
108         exit(0);
109     }
110
111     return p;
112 }
113
114 parser parse(parser p, char s){
115     if(p.str[p.pos] != s){
116         printf("\nError parsing at Position %d!\n", p.pos);
117         exit(0);
118     }
119     else{
120         p.pos++;
121     }
122
123     return p;
124 }

```

Output - Grammar 1:

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



```
vishakan@Legion:~/Desktop/Compiler Design/Ex04

Compiler Design/Ex04 on  main [?]
→ gcc RDP1.c -o r

Compiler Design/Ex04 on  main [?]
→ ./r

Recursive Descent Parser

Enter a string to parse: i+i*i+i+i

Parse Success!

Compiler Design/Ex04 on  main [?] took 10s
→ ./r

Recursive Descent Parser

Enter a string to parse: i***i

Error parsing at Position 2!

Compiler Design/Ex04 on  main [?] took 9s
→
```

Code - Grammar 2:

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<string.h>
4
5 /*Recursive Descent Parser*/
6
7 /*
8 Grammar: G: E->E+T|E-T|T
9           T->T*F|T/F|F
10          F->(E)|i
11 */
12
13 /*
14 Removed Left Recursion
15 Grammar G': E->TE'
16             E'->+TE'|-TE'|e
17             T->FT'
18             T'->*FT'|/FT'|e
19             F->(E)|i
20 */
21
22 struct parse_struct{
23     char str[100];
24     int pos;
25     int len;
26 };
27
28 typedef struct parse_struct parser;
29
30 parser E(parser p);
31 parser T(parser p);
32 parser EPrime(parser p);
33 parser F(parser p);
34 parser TPrime(parser p);
35 parser parse(parser p, char s);
36
37 int main(void){
38     parser p;
39
40     printf("\n\t\tRecursive Descent Parser\n");
41     printf("\nEnter a string to parse: ");
42     scanf("%s", p.str);
43
44     p.len = strlen(p.str);
45     p.pos = 0;
46
47     p = E(p);
```

```

48
49     if(p.pos == p.len){
50         //All characters have been parsed
51         printf("\nParse Success!\n");
52     }
53
54     else{
55         //Some characters haven't been parsed, but returned to main
56         printf("\nError parsing at Position %d!\n", p.pos);
57     }
58
59     return 0;
60 }
61
62 parser E(parser p){
63     //printf("\nAt E");
64     p = T(p);
65     p = EPrime(p);
66
67     return p;
68 }
69
70 parser T(parser p){
71     //printf("\nAt T");
72     p = F(p);
73     p = TPrime(p);
74
75     return p;
76 }
77
78 parser EPrime(parser p){
79     //printf("\nAt EPrime");
80     if(p.str[p.pos] == '+'){
81         p = parse(p, '+');
82         p = T(p);
83         p = EPrime(p);
84     }
85     else if(p.str[p.pos] == '-'){
86         p = parse(p, '-');
87         p = T(p);
88         p = EPrime(p);
89     }
90
91     return p;
92 }
93
94 parser TPrime(parser p){
95     //printf("\nAt TPrime");
96     if(p.str[p.pos] == '*'){
97         p = parse(p, '*');
98         p = F(p);

```

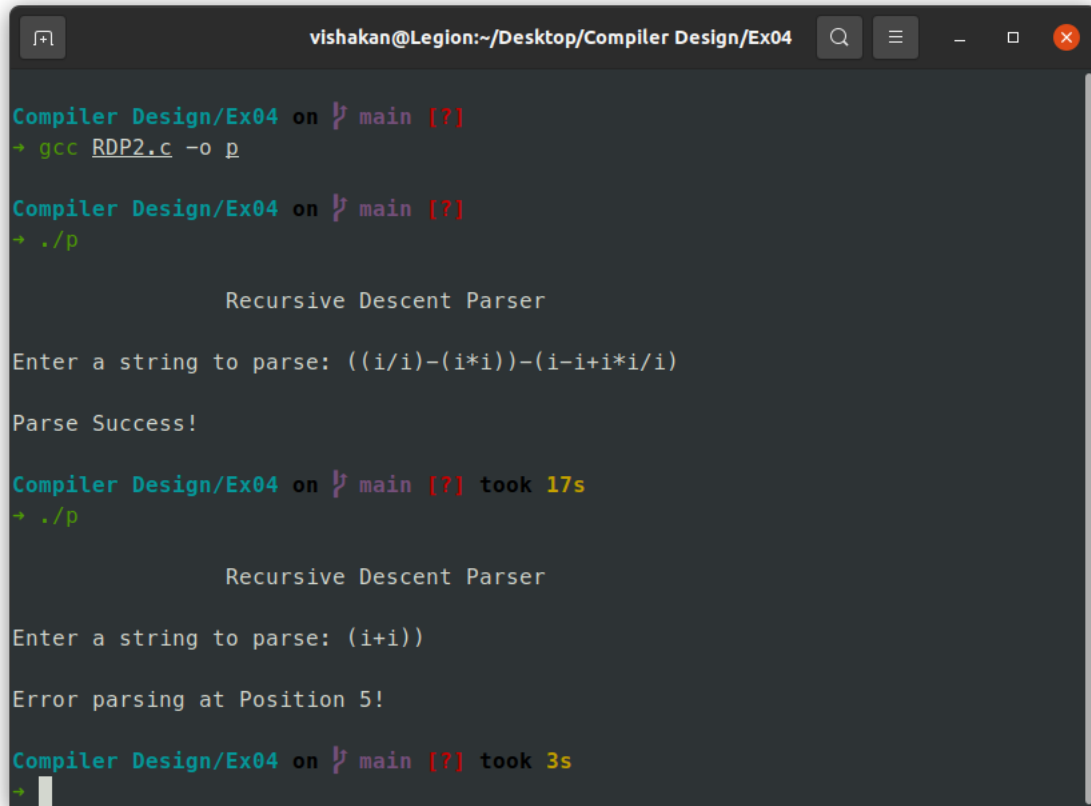
```

99         p = TPrime(p);
100     }
101     else if(p.str[p.pos] == '/'){
102         p = parse(p, '/');
103         p = F(p);
104         p = TPrime(p);
105     }
106
107     return p;
108 }
109
110 parser F(parser p){
111     //printf("\nAt F");
112     if(p.str[p.pos] == '('){
113         p = parse(p, '(');
114         p = E(p);
115         p = parse(p, ')');
116     }
117     else if(p.str[p.pos] == 'i'){
118         p = parse(p, 'i');
119     }
120     else{
121         printf("\nError parsing at Position %d!\n", p.pos);
122         exit(0);
123     }
124
125     return p;
126 }
127
128 parser parse(parser p, char s){
129     if(p.str[p.pos] != s){
130         printf("\nError parsing at Position %d!\n", p.pos);
131         exit(0);
132     }
133     else{
134         p.pos++;
135     }
136
137     return p;
138 }

```

Output - Grammar 2:

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



```
Compiler Design/Ex04 on main [?]
→ gcc RDP2.c -o p

Compiler Design/Ex04 on main [?]
→ ./p

Recursive Descent Parser

Enter a string to parse: ((i/i)-(i*i))-(i-i+i*i/i)

Parse Success!

Compiler Design/Ex04 on main [?] took 17s
→ ./p

Recursive Descent Parser

Enter a string to parse: (i+i))

Error parsing at Position 5!

Compiler Design/Ex04 on main [?] took 3s
→
```

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.