

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

```



```

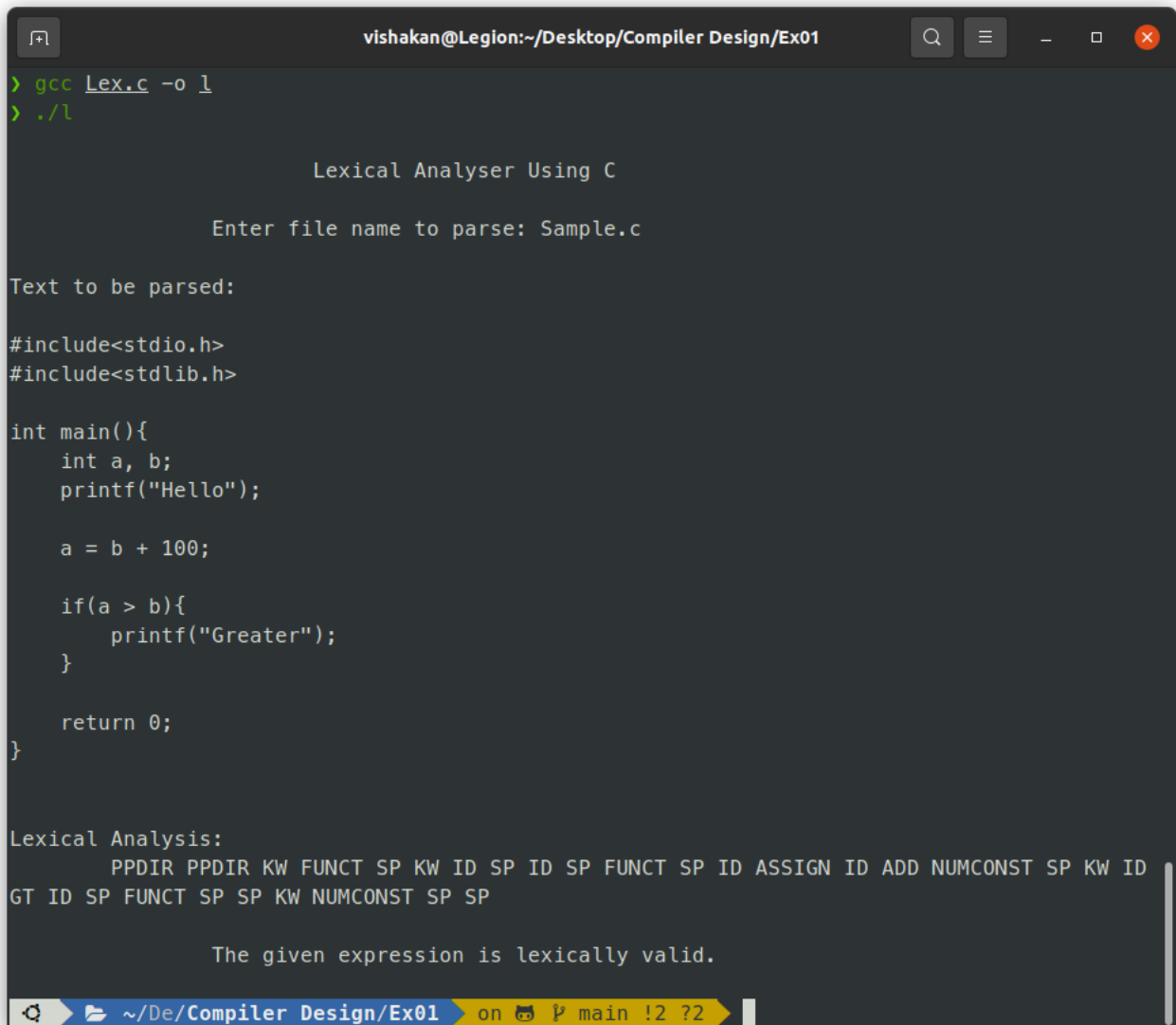
342         if(isKeyword(sub) == 1){
343             //Check if the function is a keyword based function, like
"if" & "for"
344             printf("KW ");
345             left = right;
346             continue; //Go ahead with the next check
347         }
348
349         //Otherwise, its some other function, 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");
    }

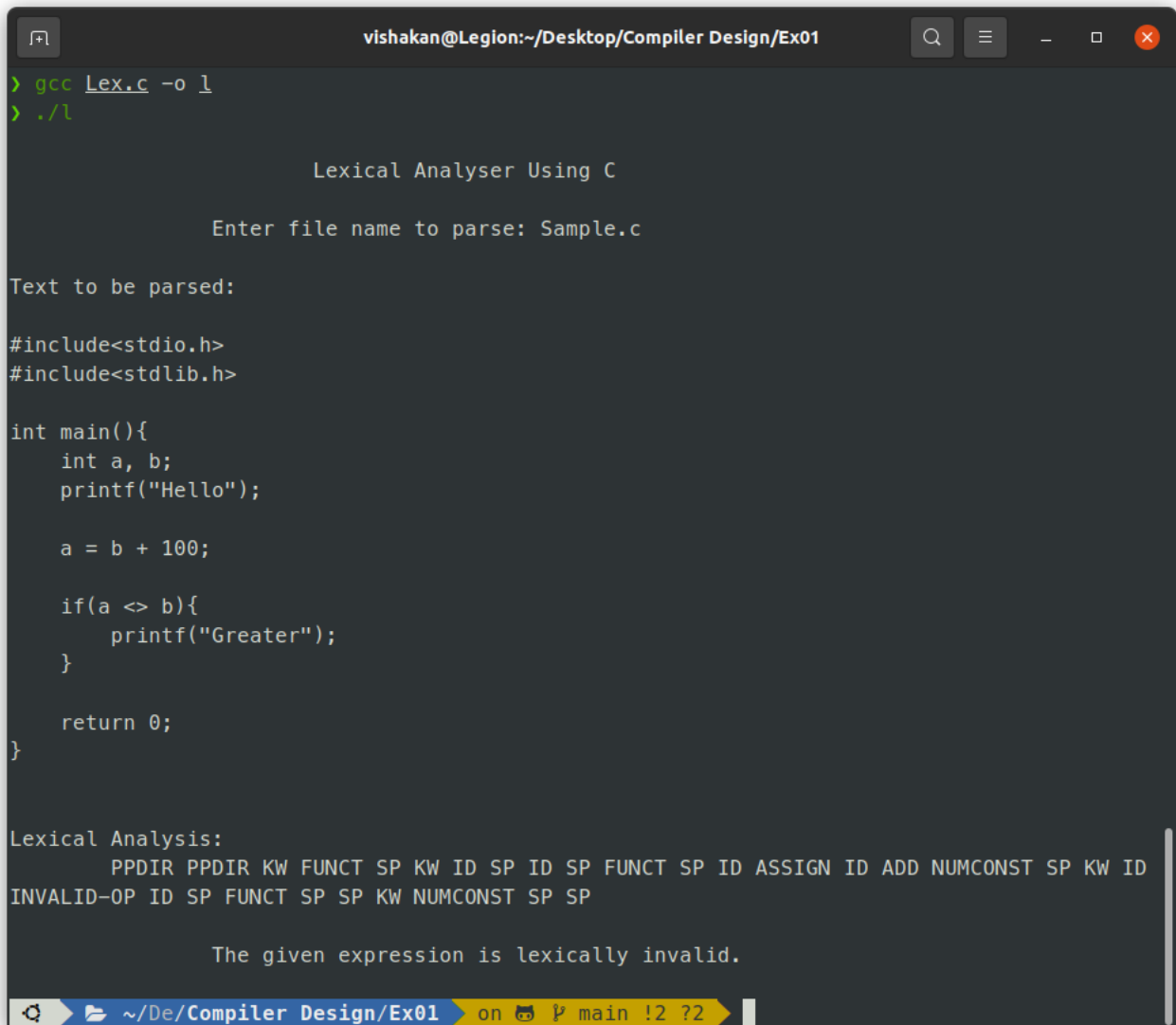
    return 0;
}

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.