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 */
3 #include <stdio.h>
4 #include <string.h>
5 #include <stdlib.h>
6 #include <ctype.h>
7 #include <unistd.h>
8 #include <fcntl.h>
int isOperator(char ch);
int isSeparator(char ch);
12 int isDelimiter(char ch);
int isValidIdentifier(char *str);
14 int isInteger(char *str);
int isKeyword(char *str);
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);
21 int main(void){
      int status = 0, len, fp;
      char text[10000], file[100];
24
      printf("\n\t\t\tLexical Analyser Using C\n");
      printf("\n\t\tEnter file name to parse: ");
26
      scanf("%[^\n]", file);
28
      fp = open(file, O_RDONLY);
30
      if(fp < 0){
31
          printf("\nError: File does not exist.\n");
32
          return 0;
33
      }
34
35
      len = read(fp, text, 10000);
36
37
      close(fp);
38
      printf("\nText to be parsed:\n\n%s\n", text);
39
      status = lexicalParse(text);
41
42
      if(status){
43
          printf("\n\n\t\tThe given expression is lexically valid.\n");
45
      else{
47
```

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

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

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

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

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

```
}
295
           for(i = right; i < len; i++){</pre>
296
                //Clearing linebreaks & tabs to spaces for efficient
297
      processing
                if(str[i] == '\n' || str[i] == '\t'){
208
                    str[i] = ' ';
299
                }
300
           }
301
           if(isDelimiter(str[right]) == 0){
303
                //If we do not encounter a delimiter, keep moving forward
304
                //"right" points to the next character
305
                right++;
307
           else if(isDelimiter(str[right]) == 1 && left == right){
309
                //If it is a delimiter, and we haven't parsed it yet
310
311
                if(isSeparator(str[right]) == 1){
312
                    //Check if the delimiter is a separator
313
                    printf("SP ");
314
                }
315
316
                else if(isOperator(str[right]) == 1){
317
                    //Check if the delimiter is an operator
318
                    if((right + 1) \le len \&\& isOperator(str[right + 1]) == 1){
319
                         //Check if the next character is also an operator
320
                         status = status & printOperator(str[right], str[right
      + 1]);
                         right++;
322
                    }
                    else{
325
                         //Next character is not an operator
326
                         status = status & printOperator(str[right], '');
327
                    }
328
329
                    //printf("\n\t\t'%c' is an operator.", str[right]);
330
                }
331
332
                right++;
333
                left = right;
334
           }
335
336
           else if(str[right] == '(' && left != right || (right == len &&
      left != right)){
                //Special case, to check for functions
338
                char *sub = subString(str, left, right - 1);
340
341
```

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

Output - Valid Case:

Figure 1: Console Output for a Valid Program.

```
vishakan@Legion:~/Desktop/Compiler Design/Ex01
                                                                         Q = -
 gcc <u>Lex.c</u> -o <u>l</u>
                        Lexical Analyser Using C
                Enter file name to parse: Sample.c
Text to be parsed:
#include<stdio.h>
#include<stdlib.h>
int main(){
    printf("Hello");
    a = b + 100;
    if(a > b){
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.

♦ ~/De/Compiler Design/Ex01 on 
P main !2 ?2
```

Output - Invalid Case:

Figure 2: Console Output for an Invalid Program.

```
vishakan@Legion:~/Desktop/Compiler Design/Ex01
                                                                  Q = -
 gcc <u>Lex.c</u> -o <u>l</u>
                      Lexical Analyser Using C
              Enter file name to parse: Sample.c
Text to be parsed:
#include<stdio.h>
#include<stdlib.h>
   printf("Hello");
   a = b + 100;
   if(a \ll b)
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.