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:

```c
/*  C Program that performs a basic lexical analysis of a given string  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>

int isOperator(char ch);
int isDelimiter(char ch);
int isValidIdentifier(char *str);
int isInteger(char *str);
int isKeyword(char *str);
int isPreprocessorDirective(char ch);
char *subString(char *str, int start, int end);
int printOperator(char ch1, char ch2);
int lexicalParse(char *str);

int main(void){
    int status = 0, len, fp;
    char text[10000], file[100];

    printf("\n\t\t\tLexical Analyser Using C\n");
    printf("\n\t\tEnter file name to parse: ");
    scanf("%[^\n]", file);

    fp = open(file, O_RDONLY);

    if(fp < 0){
        printf("\nError: File does not exist.\n");
        return 0;
    }

    len = read(fp, text, 10000);
    close(fp);

    printf("\nText to be parsed:\n\n%s\n", text);

    status = lexicalParse(text);

    if(status){
        printf("\n\n\t\tThe given expression is lexically valid.\n");
    }

    else{
        printf("\n\n\t\tThe given expression is lexically invalid.\n");
```

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

```
99          }
100     }
101
102     return 1;
103 }
104
105 int isKeyword(char *str){
106     //Checks if the string is a valid keyword
107
108     if(!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while")
        ||
109         !strcmp(str, "for") || !strcmp(str, "do") || !strcmp(str, "break")
        ||
110         !strcmp(str, "switch") || !strcmp(str, "continue") || !strcmp(str,
        "return") ||
111         !strcmp(str, "case") || !strcmp(str, "default") || !strcmp(str, "
    void") ||
112         !strcmp(str, "int") || !strcmp(str, "char") || !strcmp(str, "bool"
    ) ||
113         !strcmp(str, "struct") || !strcmp(str, "goto") || !strcmp(str, "
    typedef") ||
114         !strcmp(str, "unsigned") || !strcmp(str, "long") || !strcmp(str, "
    short") ||
115         !strcmp(str, "float") || !strcmp(str, "double") || !strcmp(str, "
    sizeof")){
116             return 1;
117         }
118
119     return 0;
120 }
121
122 int isPreprocessorDirective(char ch){
123     //Checks if the string is a valid preprocessor directive
124
125     if(ch == '#'){
126         //Basic check, works for header files, macros and const
    declarations
127         return 1;
128     }
129     return 0;
130 }
131
132 char *subString(char *str, int start, int end){
133     //Get a substring from the given string
134     int i = 0;
135     char *sub = (char *)malloc(sizeof(char) * (end - start + 2));
136
137     for(i = start; i <= end; i++){
138         sub[i - start] = str[i];
139     }
140
```

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

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

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

```
293             //If we do not encounter a delimiter, keep moving forward
294             //"right" points to the next character
295             right++;
296         }
297 
298         else if(isDelimiter(str[right]) == 1 && left == right){
299             //If it is a delimiter, and we haven't parsed it yet
300 
301             if(isOperator(str[right]) == 1){
302                 //Check if the delimiter is an operator
303                 if((right + 1) <= len && isOperator(str[right + 1]) == 1){
304                     //Check if the next character is also an operator
305                     status = status & printOperator(str[right], str[right
    + 1]);
306                     right++;
307                 }
308 
309                 else{
310                     //Next character is not an operator
311                     status = status & printOperator(str[right], ' ');
312                 }
313 
314                 //printf("\n\t\t'%c' is an operator.", str[right]);
315             }
316 
317             right++;
318             left = right;
319         }
320 
321         else if(str[right] == '(' && left != right || (right == len &&
    left != right)){
322             //Special case, to check for functions
323 
324             char *sub = subString(str, left, right - 1);
325 
326             if(isKeyword(sub) == 1){
327                 //Check if the function is a keyword based function, like
    "if" & "for"
328                 printf("KW ");
329                 left = right;
330                 continue;   //Go ahead with the next check
331             }
332 
333             //Otherwise, its some other function, parse it.
334 
335             for(i = right + 1; i < len; i++){
336                 if(str[i] == ')'){
337                     //Finish parsing till the end of the block and break
338                     printf("FUNCT ");
339                     right = i + 2;
340                     left = right;
```

```
                    status = status & 1;
                    break;
                }
            }
        }

        else if(isDelimiter(str[right]) == 1 && left != right || (right == len && left != right)){
            //We encountered a delimiter in the "right" position, but left != right
            //thus a chunk of unparsed characters exist between left and right

            //Make a substring of the unparsed characters
            char *sub = subString(str, left, right - 1);

            if(isInteger(sub) == 1){
                //Check if substring is an integer
                printf("NUMCONST ");
            }
            else if(isKeyword(sub) == 1){
                //Check if substring is a keyword
                printf("KW ");
            }
            else if(isValidIdentifier(sub) == 1){
                //Check if substring is a valid identifier
                printf("ID ");
            }
            else if(isValidIdentifier(sub) == 0 && isDelimiter(str[right - 1]) == 0){
                //Otherwise, print that it is not a valid identifier
                status = status & 0;
                printf("INVALID-ID");
            }

            left = right;   //We have parsed the chunk, thus "left" = "right"
        }

    }

    return status;
}
```

Output - Valid Case:

```
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 KW ID ID FUNCT ID ASSIGN ID ADD NUMCONST KW ID GT
    ID FUNCT KW NUMCONST

        The given expression is lexically valid.
```

Output - Invalid Case:

```
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 KW ID ID FUNCT ID ASSIGN ID INVALID-OP NUMCONST
   KW ID GT ID FUNCT KW NUMCONST

        The given expression is lexically invalid.
```