CME 3202 - Concepts of Programming Languages

Laboratory Worksheet#3

March 3, 2020

Laboratory Aim

In this laboratory section, you are expected to exercise lexical analysis.

Lexical Analysis

Lexical analyzers extract lexemes from a given input string by skipping comments and blanks, and produce the corresponding tokens. A lexeme is the lowest level syntactic unit of a language (e.g., *, sum, begin). A token is category of lexemes (e.g., identifier). The syntax of the lexemes and the corresponding tokens, which are internal codes for the grouping of the lexemes, can be understood from the BNF or EBNF of the programming language.

The EBNF consisting rules required for a simple lexical analyzer is shown below. Follow the following steps, and implement the lexical analyzer.

```
<expr> → <term> {<arith-op> <term>}
<term> → <ident>|<int-lit>
<ident> → <letter> {<letter>|<digit>}
<int-lit> → <digit> {<digit>}
<letter> → a|b|…|z
<digit> → 0|1|…|9
<arith-op> → +|-|*|/
```

1. *Step*

Launch Dev-C++ and open **lexical.c** file. Program includes uncompleted code fragments. After completing the program, for input: **a+2/5-temp1** the output should be as follows.

```
Input your string: a+2/5-temp1
 token: 4       lexeme: a       nextChar: +     inputIndex: 2   charClass: 2
 token: 2       lexeme: +       nextChar: 2     inputIndex: 3   charClass: 1
 token: 3       lexeme: 2       nextChar: /     inputIndex: 4   charClass: 2
 token: 2       lexeme: /       nextChar: 5     inputIndex: 5   charClass: 1
 token: 3       lexeme: 5       nextChar: -     inputIndex: 6   charClass: 2
 token: 2       lexeme: -       nextChar: t     inputIndex: 7   charClass: 0
 token: 4       lexeme: temp1   nextChar:       inputIndex: 11  charClass: -2
Press any key to continue . . .
```

2. *Step*

Complete the code by writing the required code fragments which are marked with the commented lines. Then copy these code fragments into the space below. (Please copy just case DIGIT: and case OPERATOR: parts)

```
void term() {
    int myTermId = termId;
    termId++;
    printf(" term starting %d\n", myTermId);

    factor();
    while (nextToken == AST_CODE || nextToken == SLASH_CODE) {
        printf(" lexeme %s\n", lexeme);
        lex();
        factor();
    }
```

```
    printf(" term ended %d\n", myTermId);
}
```

3. Step

Run your project, and paste the screenshot for input: 32*5/2y-t1 below.

```
Input your string: 32*5/2y-t1
 token: 3  lexeme: 32     nextChar: *     inputIndex: 3     charClass: 2
 token: 2  lexeme: *      nextChar: 5     inputIndex: 4     charClass: 1
 token: 3  lexeme: 5      nextChar: /     inputIndex: 5     charClass: 2
 token: 2  lexeme: /      nextChar: 2     inputIndex: 6     charClass: 1
 token: 3  lexeme: 2      nextChar: y     inputIndex: 7     charClass: 0
 token: 4  lexeme: y      nextChar: -     inputIndex: 8     charClass: 2
 token: 2  lexeme: -      nextChar: t     inputIndex: 9     charClass: 0
 token: 4  lexeme: t1     nextChar:       inputIndex: 11    charClass: -2
```

4. Step

What will be the values of the variables below after the execution of lex() when the input is "temp2-15%3" and the output lexeme is "%" and why?

nextToken: Since % is not defined as a valid operator in the given EBNF, it would be classified as UNKNOWN.
charClass: Since % does not match any predefined categories like LETTER, DIGIT, or OPERATOR, its classification would be UNKNOWN (-1).
lexLen: Since % is a single-character lexeme, its length would be 1.
The reason % is classified as UNKNOWN is that it is not included in the predefined arithmetic operators (+, -, *, /), and thus the lexer does not recognize it.

Syntax Analysis

Syntax analysis or parsing checks whether the input expression (program) is syntactically correct and also constructs a parse tree during the analysis. The syntax of the programming language is defined by the BNF or EBNF of the language.

The EBNF consisting rules required for a simple syntax analyzer is shown below. We need a lexical analyzer to be able to analyze syntactically. Follow the steps and implement the lexical analyzer and the syntax analyzer.

```
<expr> → <term> {(+|-) <term>}
<term> → <factor> {(*|/) <factor>}
<factor> → <ident>|<int-lit> | (<expr>)
<ident> → <letter> {<letter>|<digit>}
<int-lit> → <digit> {<digit>}
<letter> → a|b|…|z
```

```
<digit> → 0|1|…|9
```

5. Step

Open **syntax.c** program that includes uncompleted code fragments in Dev-C++. When code is completed, following output is obtained for input: **index*2+35/a.**

```
Input your string:
index*2+35/a
expr starting 0
 term starting 0
  factor starting 0
  lexeme index
  factor ended 0
 lexeme *
  factor starting 1
  lexeme 2
  factor ended 1
 term ended 0
lexeme +
 term starting 1
  factor starting 2
  lexeme 35
  factor ended 2
 lexeme /
  factor starting 3
  lexeme a
  factor ended 3
 term ended 1
expr ended 0
success
```

6. Step

Complete the code by writing the required code fragments which are marked with the commented lines and copy your code below.

```
void term() {
    int myTermId = termId;
    termId++;
    printf(" term starting %d\n", myTermId);

    factor();
    while (nextToken == AST_CODE || nextToken == SLASH_CODE) {
        printf(" lexeme %s\n", lexeme);
        lex();
        factor();
    }

    printf(" term ended %d\n", myTermId);
}
```

7. Step

Run your project and paste the screenshot for input: 12-8/2x-y below and **explain the result**.

```
lexeme 12
lexeme -
lexeme 8
lexeme /
lexeme 2
Syntax error
```

8. Step

Run your program for the input “total-25:12”. Do the lexemes “-”, “25”, “:” or “12” exist in the output? **Why?**

"total" as a valid identifier (IDENT).
"-" as a valid operator.
"25" as an integer literal (INT_LIT).
":" as an **unknown** character.
"12" may not be processed if the parser stops at ":".
Since ":" is not defined in the language grammar, it will be categorized as UNKNOWN, likely causing a syntax error before "12" is fully analyzed.