COMPILER PROJECT I 2021

The goal of the first term-project is to implement a lexical analyzer (a.k.a., scanner) as we've learned. More specifically, you will implement the lexical analyzer **for a simplified Java programming language** with the following lexical specifications;

<Lexical specifications>

- ✓ **Variable type**
  - **int** for a signed integer
  - **char** for a single character
  - **boolean** for a Boolean string
  - **String** for a literal string
- ✓ **Signed integer**
  - A single zero digit (e.g., 0)
  - A non-empty sequence of digits, starting from a non-zero digit
    (e.g., 1, 22, 123, 56, … any non-zero positive integers)
    (e.g., 001 is not allowed)
  - A non-empty sequence of digits, starting from a minus sign symbol and a non-zero digit
    (e.g., -1, -22, -123, -56, .. any non-zero negative integers)
- ✓ **Single character**
  - A single digit, English letter, block, or any symbol, starting from and terminating with a symbol ' (e.g., 'a', '1', ' ', '&')
- ✓ **Boolean string:** true and false
- ✓ **Literal string**
  - Any combination of digits, English letters, and blanks, starting from and terminating with a symbol " (e.g., "Hello world", "My student id is 12345678")
- ✓ **An identifier of variables and functions**

- A non-empty sequence of English letters, digits, and underscore symbols, starting from an English letter or a underscore symbol (e.g., i, j, k, abc, ab_123, func1, func_, __func_bar__)

- ✓ **Keywords for special statements**
  - **if** for if statement
  - **else** for else statement
  - **while** for while-loop statement
  - **class** for class statement
  - **return** for return statement
- ✓ **Arithmetic operators**: +, -, *, and /
- ✓ **Assignment operator**: =
- ✓ **Comparison operators**: <, >, ==, !=, <=, and >=
- ✓ **A terminating symbol of statements**: ;
- ✓ **A pair of symbols for defining area/scope of variables and functions**: { and }
- ✓ **A pair of symbols for indicating a function/statement**: (and)
- ✓ **A pair of symbols for using an array**: [and]
- ✓ **A symbol for separating input arguments in functions:** ,
- ✓ **Whitespaces**: a non-empty sequence of ₩t, ₩n, and blanks

Based on this specification, you will 1) define tokens (e.g., token names) for a simplified Java language, 2) make regular expressions which describe the patterns of the tokens, 3) construct a NFA for the regular expressions, 4) translate the NFA into a DFA, especially in the form of a table, and 5) implement a program which does a lexical analysis (recognizing tokens).

[IMPORTANT NOTE]

1. You MUST build regular expressions, NFAs, and DFAs by writing or drawing by hand.

(Do not use any kind of computer program like Lex for this procedure)

2. You can use C, C++, JAVA, or Python to implement your lexical analyzer.

3. Your lexical analyzer MUST run on Linux or Unix-like OS without any error.

(Do not use Windows-only APIs and please test your program on Linux machines before submission)

5. Your lexical analyzer should work as follows;

- ✓ **On a command line, your analyzer must run with the following command**

  lexical_analyzer <input_file_name>

- ✓ **Input:** A program written in a simplified Java programming language

  (You don't need to think about the syntax grammar of the program yet)

- ✓ **Output:** <input_file_name_output.txt>
  - ■ (If an input program has no error) A symbol table which stores the information of all tokens including their names and optional values
    - ◆ This output will be used as an input of your next term-project (syntax analyzer)
  - ■ (Otherwise) An error report which explains why and where the error occurred (e.g., line number)

Input: test.c		Output: test.out
int func(int a) { return 0; }		<table><tr><td>INT</td><td></td></tr><tr><td>ID</td><td>func</td></tr><tr><td>LPAREN</td><td>(</td></tr><tr><td>…</td><td>…</td></tr></table>

6. Do not include "WHITESPACE" tokens in the output token list

7. There will be some issues with the symbol -. Please consider the syntax of the given code to address this.

Term-project schedule and submission

- ✓ **Deadline: 4/18, 23:59 (through an e-class system)**
  - ■ For a delayed submission, you will lose 0.1 * your original project score per each delayed day
- ✓ Submission file: team_<your_team_number>.zip or .tar.gz
  - ■ The compressed file should contain
    - ◆ The source code of your lexical analyzer with detailed comments
    - ◆ The executable binary file of your lexical analyzer
    - ◆ Documentation (the most important thing!)
      - ● It must include 1) the definition of tokens and their regular expressions, 2) the DFA transition graph or table for recognizing the regular expressions, 3) all about how your lexical analyzer works for recognizing tokens (for example, overall procedures, implementation details like algorithms and data structures, working examples, and so on)
    - ◆ Test input files and outputs which you used in this project
      - ● The test input files are not given. You should make the test files, by yourself, which can examine all the token patterns.
- ✓ If there exist any error in the given lexical specification, please send an e-mail to hskimhello@cau.ac.kr