COMPILER PROJECT I 2020

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 C programming language with the following lexical specifications;

<Lexical specifications>

- ✓ **Variable type**
  - int for a signed integer
  - char for a literal string
  - bool for a Boolean string
  - float for a floating-point number
- ✓ **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)

  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")
- ✓ **Boolean string:** true and false
- ✓ **Floating-point number**
  - A sequence that meets the following conditions:
    1) It starts with or without a negative sign symbol
    2) . (a decimal point) appears only once

3) Scientific/exponential symbols like E are not allowed

4) Both left and right side of the decimal point must not be empty sequence

5) The left side of a decimal point must be a single digit 0 or a non-empty sequence starting from a non-zero digit

6) The right side of a decimal point must be a single digit 0 or a non-empty sequence terminating with a non-zero digit

(e.g., 0.5, 0.0, -10.0, 100.00001, …)

- ✓ **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
  - for for for-loop statement
  - return for return statement

- ✓ **Arithmetic operators**: +, -, *, and /
- ✓ **Bitwise 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 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 C 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).

NOTE: you MUST build regular expressions, NFAs, and DFAs by hand

(Do not use a program like Lex for this procedure)

For the implementation, you can use C, C++, JAVA, or Python as you want (it is recommended to implement it on Linux or other Unix-like OS, but it's not mandatory). However, your lexical analyzer should work as follows;

- ✓ **The execution command of your lexical analyzer:** lexical_analyzer <input_file_name>
- ✓ **Input:** A program written in a simplified C programming language

  (You don't need to think about the syntax of the program yet)
- ✓ **Output:** <intput_file_name.out>
  - ■ (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; }		INT	
		ID	func
		LPAREN	(
		…	…

Term-project schedule and submission

- ✓ **Deadline: 5/9, 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