Compiler Project #2, 2022

The goal of the second term-project is to implement **a bottom-up syntax analyzer (a.k.a., parser)** as we've learned. More specifically, you will implement the syntax analyzer for a simplified Java programming language with the following context free grammar G;

□ **CFG** G:

01: CODE → VDECL CODE | FDECL CODE | CDECL CODE | **ε**

02: VDECL → **vtype id semi** | **vtype** ASSIGN **semi**

03: ASSIGN → **id assign** RHS

04: RHS → EXPR | **literal** | **character** | **boolstr**

05: EXPR → EXPR **addsub** EXPR | EXPR **multdiv** EXPR

06: EXPR → **lparen** EXPR **rparen** | **id** | **num**

07: FDECL → **vtype id lparen** ARG **rparen lbrace** BLOCK RETURN **rbrace**

08: ARG → **vtype id** MOREARGS | **ε**

09: MOREARGS → **comma vtype id** MOREARGS | **ε**

10: BLOCK → STMT BLOCK | **ε**

11: STMT → VDECL | ASSIGN **semi**

12: STMT → **if lparen** COND **rparen lbrace** BLOCK **rbrace** ELSE

13: STMT → **while lparen** COND **rparen lbrace** BLOCK **rbrace**

14: COND → COND **comp** COND | **boolstr**

15: ELSE → **else lbrace** BLOCK **rbrace** | **ε**

16: RETURN → **return** RHS **semi**

17: CDECL → **class id lbrace** ODECL **rbrace**

18: ODECL → VDECL ODECL | FDECL ODECL | **ε**

□ **Terminals (21):**

1. **vtype** for the types of variables and functions
2. **num** for signed integers
3. **character** for a single character
4. **boolstr** for Boolean strings
5. **literal** for literal strings
6. **id** for the identifiers of variables and functions
7. **if**, **else**, **while**, and **return** for if, else, while and return statements, respectively
8. **class** for class declarations
9. **addsub** for + and - arithmetic operators

10. **multdiv** for * and / arithmetic operators
11. **assign** for assignment operators
12. **comp** for comparison operators
13. **semi** and **comma** for semicolons and commas, respectively
14. **lparen**, **rparen**, **lbrace**, and **rbrace** for (,), {, and }, respectively

□ **Non-terminals (15)**

CODE, VDECL, ASSIGN, RHS, EXPR, FDECL, ARG, MOREARGS, BLOCK, STMT, COND, ELSE, RETURN, CDECL, ODECL

□ **Start symbol:** CODE

□ **Descriptions**

- The given CFG G is non-left recursive, but **ambiguous**
- **Codes** include zero or more declarations of functions, variables, and classes (CFG 01)
- **Variables** are declared with or without initialization (CFG 02, 03)
- **The right-hand side (RHS) of assignment operators** can be classified into four types such as (1) arithmetic operators (expressions), (2) literal strings (CFG 13), (3) a single character, and (4) Boolean strings (CFG 04)
- **Arithmetic operations** are the combinations of +, -, *, / operators (CFG 05,06,07)
- **Functions** can have zero or more input arguments (CFG 08,09,10)
- **Function blocks** include zero or more statements (CFG 11)
- There are **four types of statements**: (1) variable declaration, (2) assignment operations, (3) if-else statements, and (4) while statements (CFG 12,13,14)
- **if** and **while** statements include a conditional operation which consists of Boolean strings and an condition operator (CFG 13,14,15)
- **if** statements can be used with or without an else statement (CFG 13 & 16)
- **return** statements return (1) the computation result of arithmetic operations, (2) literal strings, (3) a single character, or (4) Boolean strings (CFG 17)
- **class** is declared with zero or more declarations of functions and variables (CFG 18,19)

Based on this CFG, you should implement a bottom-up parser as follows:

- Discard an ambiguity in the CFG
- Construct a SLR parsing table for the non-ambiguous CFG through the following website: http://jsmachines.sourceforge.net/machines/slr.html
- Implement a SLR parsing program for the simplified Java programming language by using the constructed table

For the implementation, you can use C, C++, JAVA, or Python as you want. However, your syntax analyzer must run on Linux or Unix-like OS without any error.

Your syntax analyzer should work as follows:

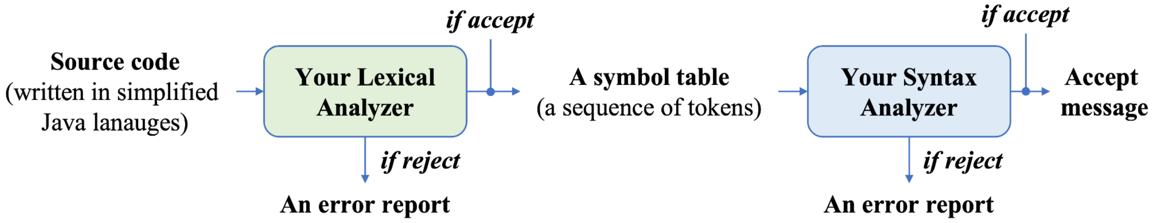
□ **The execution flow of your syntax analyzer:**

- ./lexical_analyzer <input_file_name>
- ./syntax_analyzer <output_of_your_lexical_analyzer>
  : *You can internally pass your input from your lexical analyzer to your syntax analyzer.*

□ **Input:** an output of your lexical analyzer program

□ **Output:** just an acceptance message

- (if an output is 'reject') please make an error report which explains why and where the error occurred (e.g., line number)



Term-project schedule and submission

- **Deadline:** 2022-06-21 (please use the e-Class)
  - For a delayed submission, you will lose 0.1 * your original project score per each delayed day
- **Submission file:** <your student ID>_<your_name>.tar.gz or .zip
  - The compressed file should include
    - The source code of **your syntax and lexical analyzer** with detailed comments
    - The executable binary files of **your syntax analyzer + lexical analyzer**
    - Documentation (the most important thing!)
      - It must include (1) your non-ambiguous CFG G and (2) your SLR parsing table
      - It must also include any change in the CFG G and all about how your syntax analyzer works for validating token sequences (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 syntax grammars.
- If there exist any error in the given CFG, please send an e-mail to kimjsung@cau.ac.kr