COMPILER PROJECT II 2019

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

CFG G:

01: CODE → VDECL CODE | FDECL CODE | ϵ
02: VDECL → vtype id semi
03: FDECL → vtype id lparen ARG rparen lbrace BLOCK rbrace
04: ARG → vtype id MOREARGS | ϵ
05: MOREARGS → comma vtype id MOREARGS | ϵ
06: BLOCK → STMT BLOCK | ϵ
07: STMT → VDECL | id assign RHS semi
08: STMT → if lparen COND rparen lbrace BLOCK rbrace else lbrace BLOCK rbrace
09: STMT → while lparen COND rparen lbrace BLOCK rbrace | FCALL semi
10: RHS → EXPR | FCALL | literal
11: EXPR → TERM addsub EXPR | TERM
12: TERM → FACTOR multdiv TERM | FACTOR
13: FACTOR → lparen EXPR rparen | id | num
14: FCALL → id lparen ARG rparen
15: COND → FACTOR comp FACTOR

✓ **Terminals (18)**

1. **vtype** for the types of variables and functions
2. **num** for signed integers
3. **literal** for literal strings
4. **id** for the identifiers of variables and functions
5. **if, else, while,** and **return** for if, else, while, and return statements respectively
6. **addsub** for + and - arithmetic operators
7. **multdiv** for * and / arithmetic operators

8. **assign** for assignment operators

9. **comp** for comparison operators

10. **semi** and **comma** for semicolons and colons respectively

11. **lparen, rparen, lbrace,** and **rbrace** for (,), {, and } respectively

- ✓ **Non-terminals (13)**

  CODE, VDECL, FDECL, ARG, MOREARGS, BLOCK, STMT, RHS, EXPR, TERM, FACTOR, COND, FCALL

- ✓ **Start symbol:** CODE

Descriptions

- ✓ The given CFG G is not ambiguous and non-left recursive.

  But, left factoring is required if you want to implement a top-down parser
- ✓ Source codes include zero or more declarations of functions and variables (CFG line 1)
- ✓ Variables are always declared without initialization (CFG line 2)
- ✓ Functions can have zero or more input arguments (CFG line 3 ~ 5)
- ✓ Function blocks include zero or more statements (CFG line 6)
- ✓ There are five types of statements: 1) variable declarations, 2) assignment operations, 3) if-else statements, 4) while statements, and 5) function calls (CFG line 7 ~ 9)
- ✓ if-else statements without else are not allowed (CFG line 8)
- ✓ The right hand side of assignment operations can be classified into three types; 1) arithmetic operations (expressions), 2) function calls, and 3) literal strings (CFG line 10 ~ 14)
- ✓ Arithmetic operations are the combinations of +, -, *, / operators (CFG line 11 ~ 13)

Based on this CFG, you can implement 1) a top-down parser or 2) a bottom-up parser.

- ✓ **If you want to implement a top-down parser**, then you are required 1) to do left factoring, 2) to compute first and follow sets, 3) to construct a LL(1) parsing table, and 4) to implement a LL(1) parser.

- ✓ **If you want to implement a bottom-up parser,** then you are required 1) to construct an NFA for recognizing viable prefixes of G, 2) to convert the NFA into a DFA, 3) to compute follow sets, 4) to construct a SLR parsing table, and 5) to implement a SLR parser.

  NOTE: if you implement a correct bottom-up parser, you will get an additional 5 points

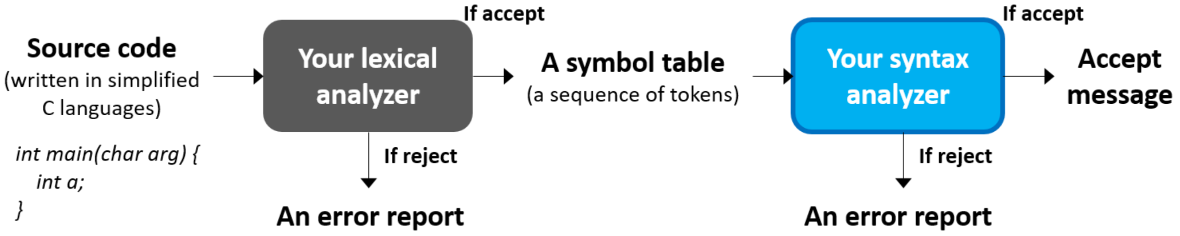
For the implementation, you can use C, C++, JAVA, or Python as you want. However, your syntax analyzer should work as follows;

- ✓ **The execution flow of your syntax analyzer:**

  lexical_analyzer <input_file_name>

  syntax_analyzer <output_of_your_lexical_analyzer>

- ✓ **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: 6/9, 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 syntax analyzer with detailed comments
    - ◆ The executable binary file of your syntax analyzer
    - ◆ Documentation (the most important thing!)
      - ● If you implemented a top-down parser, it must include 1) your re-written (left-factored) CFG and 2) your LL(1) parsing table
      - ● If you implemented a bottom-up parser, it must include 1) your DFA transition graph or table for recognizing viable prefixes of the CFG G and 2) your SLR parsing table
      - ● In both cases, It must include any changes 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 hskimhello@cau.ac.kr