COMPILER PROJECT 2024

The goal of the 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** C programming language with the following context free grammar G;

CFG G:

01: CODE → VDECL CODE | FDECL 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

✓ **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 (13)**

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

- ✓ **Start symbol:** CODE

Descriptions

- ✓ The given CFG G is non-left recursive, but **ambiguous**.
- ✓ Codes include zero or more declarations of functions and variables (CFG line 1)
- ✓ Variables are declared with or without initialization (CFG line 2 ~ 3)
- ✓ The right hand side of assignment operations can be classified into four types; 1) arithmetic operations (expressions), 2) literal strings, 3) a single character, and 4) Boolean strings (CFG 4)
- ✓ Arithmetic operations are the combinations of +, -, *, / operators (CFG line 5 ~ 6)
- ✓ Functions can have zero or more input arguments (CFG line 7 ~ 9)
- ✓ Function blocks include zero or more statements (CFG line 10)
- ✓ There are four types of statements: 1) variable declarations, 2) assignment operations, 3) if-else statements, and 4) while statements (CFG line 11 ~ 13)
- ✓ if and while statements include a conditional operation which consists of Boolean strings and condition operators (CFG line 12 ~ 14)

✓ if statements can be used with or without an else statement (CFG line 12 & 15)

✓ return statements return 1) the computation result of arithmetic operations, 2) literal strings, 3) a single character, or 4) Boolean strings (CFG line 16)

✓ This is not a CFG for C. This is for **simplified** C. So, you don't need to consider grammars and structures not mentioned in this specification.

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, please use C, C++, or Python (If you want to use . 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:**

syntax_analyzer <input file>

✓ **Input:** A sequence of tokens (terminals) written in the input file

e.g., vtype id semi vtype id lparen rparen lbrace if lparen boolstr comp boolstr rparen lbrace rbrace

✓ **Output**

- (If a parsing decision output is "accept") please construct a parse tree **(not abstract syntax tree)** for the input sequence
    - ◆ You can design the data structure to represent the tree as you want.
- (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 (if you implemented using a complied language)**
    - 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 hskimhello@cau.ac.kr