COMPILER PROJECT II 2020

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 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:   FDECL → vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
05:   ARG → vtype id MOREARGS | ε
06:   MOREARGS → comma vtype id MOREARGS | ε
07:   BLOCK → STMT BLOCK | ε
08:   STMT → VDECL | ASSIGN semi
09:   STMT → if lparen COND rparen lbrace BLOCK rbrace ELSE
10:   STMT → while lparen COND rparen lbrace BLOCK rbrace
11:   STMT → for lparen ASSIGN semi COND semi ASSIGN rparen lbrace BLOCK rbrace
12:   ELSE → else lbrace BLOCK rbrace | ε
13:   RHS → EXPR | literal
14:   EXPR → TERM addsub EXPR | TERM
15:   TERM → FACTOR multdiv TERM | FACTOR
16:   FACTOR → lparen EXPR rparen | id | num | float
17:   COND → FACTOR comp FACTOR
18:   RETURN → return FACTOR semi
```


✓ **Terminals (20)**

1. **vtype** for the types of variables and functions

2. **num** for signed integers

3. **float** for floating-point numbers

4. **literal** for literal strings

5. **id** for the identifiers of variables and functions

---

6. **if, else, while, for** and **return** for if, else, while, for and return statements respectively

7. **addsub** for + and - arithmetic operators

8. **multdiv** for * and / arithmetic operators

9. **assign** for assignment operators

10. **comp** for comparison operators

11. **semi** and **comma** for semicolons and commas respectively

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

✓ **Non-terminals (14)**

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

✓ **Start symbol:** CODE

Descriptions

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

✓ Source codes include zero or more declarations of functions and variables (CFG line 1)

✓ Variables are declared with or without initialization (CFG line 2 ~ 3)

✓ Functions can have zero or more input arguments (CFG line 4 ~ 6)

✓ Function blocks include zero or more statements (CFG line 7)

✓ There are five types of statements: 1) variable declarations, 2) assignment operations, 3) if-else statements, 4) while statements, and 5) for statements (CFG line 8 ~ 11)

✓ if statements can be used with or without an else statement (CFG line 8 & 12)

✓ The right hand side of assignment operations can be classified into two types; 1) arithmetic operations (expressions) and 2) literal strings (CFG line 13)

✓ Arithmetic operations are the combinations of +, -, *, / operators (CFG line 14 ~ 16)

Based on this CFG, you should implement a bottom-up parser.

✓ 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 the follow sets, 4) to construct a SLR parsing table, and 5) to
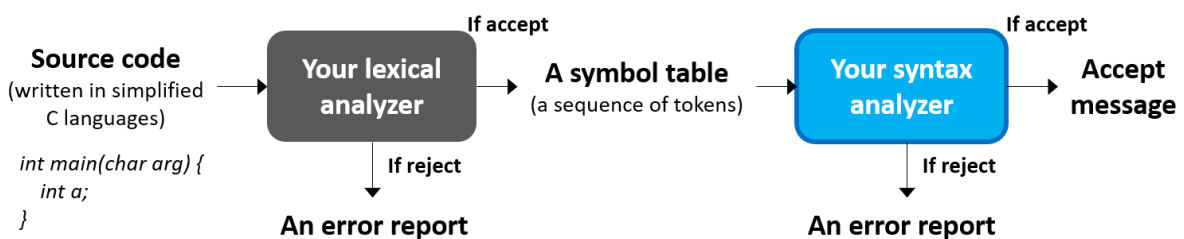
implement a SLR parser.

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)

Source code
(written in simplified
C languages)

int main(char arg) {
* int a;*
}

Your lexical analyzer

If accept → **A symbol table** (a sequence of tokens)

Your syntax analyzer

If accept → **Accept message**

If reject ↓ **An error report**

If reject ↓ **An error report**

Term-project schedule and submission

✓ **Deadline: 6/27, 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 and lexical analyzer** with detailed comments

◆ The executable binary file of **your syntax analyzer + lexical analyzer**

◆ Documentation (the most important thing!)

● It must include 1) your DFA transition graph or table for recognizing viable prefixes of the CFG G and 2) your SLR parsing table

● 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