Compiler Project 2 - VC Parser Report

Be Trong Nghia **Pham Duy Minh Quan** **Vuong Vu Duc Hoang**

1. Requirements

The project and documentation must:

- Identify and comprehend the current grammar of VC language; reason if the language is LL(1) compatible; if not, convert the grammar to an appropriate, compatible variant with FIRST and FOLLOW.
- Build an appropriate state table using the syntax variant above & apply it in the code. In effect, the project program should receive an input source code(*.vc) and output the AST form of the code (*.vcps), which is formatted with normal brackets "()" outside every node of the tree.

Bonus will be given for submission with:

- External state table, in form of (*.dat) that would be used by the program.
- Detailed comment/documentation w.r.t program source code.

2. Self-evaluation

Our solution has met all the criteria specified in the project requirements.

- Identifying and Comprehending the Current Grammar:

We have analyzed the current grammar of the VC language to understand its structure and rules. After careful examination, we determined that the language is not LL(1) compatible due to **left recursion** and **ambiguities** (notably **the dangling else**).

Example of left recursions and the dangling else:

```
rel-expr -> additive-expr
            | rel-expr "<" additive-expr
            | rel-expr "<=" additive-expr
            | rel-expr ">" additive-expr
            | rel-expr ">=" additive-expr

stmt -> compound-stmt
        | if-stmt
        | for-stmt
        | while-stmt
        | break-stmt
        | continue-stmt
        | return-stmt
        | expr-stmt
if-stmt -> if "(" expr ")" stmt ( else stmt )?
```

- Converting the Grammar to an Appropriate Variant:

To make the grammar LL(1) compatible, we performed the necessary transformations to eliminate the conflicts using the FIRST and FOLLOW sets. Also, we guarded the `else` with `{ }`.

- Building and using the external State Table:

Using the modified grammar, we constructed an appropriate state table in an external format (bonus). After that, we implement a parser with a `TableHandler` class to put that file into use. The parser's structure will be shown in the following part.

However, something has gone wrong with the parsing process, and we don't have time to find the problem (we have to submit this report late and there is no more time to spend).
Example: The division 1/2/3 should be ((1/2)/3), but our parser returns (1/(2/3)).

- Detailed Comment/Documentation:

We have extensively commented on our code to make sure every part is explained thoroughly. The code is also formatted with Black Formatter to ensure aesthetics. For the documentation: this document and the README file were written with much useful information related to this project.

Overall, we tried to fulfill all the requirements, but the parser is still buggy and we are late for the deadline. But still, we tried our best!

3. The parser's structure

The parser has 4 parts:

- The lexer: Use Lark, a parsing toolkit for Python, to do the tokenization part. A .lark file is defined for the terminal rules, then it would be loaded into the program to do lexical analysis on a .vc source code file.
- The table handler: Handle the parsing table, defined as a .csv file (but the extension has been changed into .dat to fit this project's requirement).
- The AST: Generate the AST from the parsing process, and write it down a .vcps file.
- The parser: Read arguments and perform parsing on a single file, a folder, or the example folder. Usage is written in the README file.

Because the code is commented very thoroughly this time, we think it wouldn't be necessary to include explanations for each module here.

4. Some used resources

These are some honorable mentions:

- https://duongoku.dev/archive/2023/LL_Parser/
- https://mdaines.github.io/grammophone/#/
- http://smlweb.cpsc.ucalgary.ca/