FORTRESS
Introduction to language theory and compiling
Project – Part 2

Gilles GEERAERTS Léonard BRICE Sarah WINTER

October 26, 2022

For this second part of the project, you have to write the *parser* of your FORTRESS compiler. More precisely, you must:

1. Transform the FORTRESS grammar (see Figure 1 at the end of document) in order to: (a) Remove unproductive and/or unreachable variables, if any; (b) Make the grammar non-ambiguous by taking into account the priority and the associativity of the operators. Table 1 shows these priorities and associativities: operators are sorted by decreasing order of priority (with two operators in the same row having the same priority). Please note that you do not have to handle priority if there is no ambiguity. (c) Remove left-recursion and apply factorisation where needed;

2. Check your grammar is LL(1) and write the *action table* of an LL(1) parser for the transformed grammar. You must justify this table by giving the details of the computations of the relevant First and Follow sets.

3. Write, in Java, a parser for this grammar. Design a *recursive descent LL(1) parser*; you can either code it by hand or write a parser generator which will generate it from the action table. You must implement your parser *from scratch*, in the sense that you are not allowed to use tools such as `yacc`, or existing implementations (e.g. for pushdown automata).

   You can use your own scanner or the scanner which is be provided on the Université Virtuelle (see Project Part 1) in order to extract the tokens from the input file. If you have not already implemented it for Part 1, you can view on the Université Virtuelle (see Project Part 1) how to access the generated scanner from your `Main.java`.

   For this part of the project, your program must output on `stdout` *the leftmost derivation* of the input if it is correct; or an (explanatory) error message if there is a syntax error. The format for such leftmost derivation is a sequence of rule numbers (do not forget to number your rules accordingly in the report!) separated by a space. For instance, if your input string is part of the grammar, as witnessed by a successful derivation obtained by applying rules 1,2,4,9,3,10,3, your program must output `1 2 4 9 3 10 3`.

   Additionally, your program must build the parse tree (aka. the derivation tree) of the input string and, when called by adding `-wt filename.tex` to the command (see below for details), write it as a LaTeX file called `filename.tex`. To this end, a `ParseTree.java` class is provided on the Université Virtuelle. Use the method `toLatex()`. *You are free to modify* this class as you wish, or even design your own from scratch (as usual, explain what you did and why you did it in the report).

4. On another note, you are also free to modify `Symbol.java` and `LexicalUnit.java`.

Operators	**Associativity**
− (unary)	right
*, /	left
+, − (binary)	left
>, <, =	left

Table 1: Priority and associativity of the FORTRESS operators (operators are sorted in decreasing order of priority). Note the difference between *unary* and *binary* minus (-).

You must hand in:

- A PDF report containing the modified grammar, the action table, as well as descriptions of your example files; remember to enumerate the rules of your grammar;
- *Bonus:* For this part, no bonus is *a priori* specified, but recall that initiative is encouraged for this project: do not hesitate to explain why some features would be hard to add at this point of the project, or to add relevant features to FORTRESS. Ask us before, just to check that the feature in question is both relevant and doable;
- The source code of your parser in a JAVA source file called `Parser.java`, your `Main.java` file calling the parser, as well as all the auxiliary classes (`ParseTree.java`, etc.);
- The FORTRESS example files you have used to test your parser. It is necessary to provide at least one example file of your own.

You must structure your files in these folders:

- `doc` contains the JAVADOC and the PDF report.
- `test` contains all your example FORTRESS files.
- `dist` contains an executable JAR **that must be called** `part2.jar`.
- `src` contains your source files.
- `more` contains all other files.
- Additionally, a (working) `Makefile` must be provided.

Your implementation must contain:

1. Your scanner (from the first part of the project), or the sample solution;
2. Your parser;
3. An executable that reads the file given as argument and writes on the standard output stream the leftmost derivation, with an option to write/print the parse tree. The command for running your executable must be as follows:

   ```
   java -jar part2.jar [OPTION] [FILE]
   ```

   Concretely, you have, e.g., `java -jar part2.jar sourceFile.fs`, or `java -jar part2.jar -wt sourceFile.tex sourceFile.fs`.

You must compress your folder (in the *zip* format—no *rar* or other format), **which is called according to the following regexp**:

`Part2_Surname1(_Surname2)?.zip`

where `Surname1` and, if you are in a group, `Surname2` are the last names of the student(s) (in alphabetical order), and exactly **one group member** must submit it on the Université Virtuelle before the end of **November, 21st**. You are encouraged and allowed to work in group of maximum two students.

[1]	<Program>	→ `BEGIN` [ProgName] <Code> `END`
[2]	<Code>	→ <Instruction> `,` <Code>
[3]		→ ε
[4]	<Instruction>	→ <Assign>
[5]		→ <If>
[6]		→ <While>
[7]		→ <Print>
[8]		→ <Read>
[9]	<Assign>	→ [VarName] `:=` <ExprArith>
[10]	<ExprArith>	→ [VarName]
[11]		→ [Number]
[12]		→ `(` <ExprArith> `)`
[13]		→ `-` <ExprArith>
[14]		→ <ExprArith> <Op> <ExprArith>
[15]	<Op>	→ `+`
[16]		→ `-`
[17]		→ `*`
[18]		→ `/`
[19]	<If>	→ `IF (`<Cond>`) THEN` <Code> `END`
[20]		→ `IF (`<Cond>`) THEN` <Code> `ELSE` <Code> `END`
[21]	<Cond>	→ <ExprArith> <Comp> <ExprArith>
[22]	<Comp>	→ `=`
[23]		→ `>`
[24]		→ `<`
[25]	<While>	→ `WHILE (`<Cond>`) DO` <Code> `END`
[26]	<Print>	→ `PRINT (`[VarName]`)`
[27]	<Read>	→ `READ (`[VarName]`)`

Figure 1: The FORTRESS grammar.