AlCOl
Introduction to language theory and compiling
Project – Part 2

Gilles GEERAERTS Raphaël BERTHON Sarah WINTER

October 26, 2021

For this second part of the project, you will write the *parser* of your AlCOl compiler. More precisely, you must:

1. Transform the AlCOl grammar (see Figure 2 at the end of the statement) 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 need be;

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. We strongly recommend you to design a recursive descent LL(1) parser, since it is the easiest way; you can either code it by hand or write a parser generator which will generate it from the action table. You can also design a pushdown automaton for this, or any other solution you have in mind (for "uncommon" solutions, please ask us first), as long as it is as efficient – i.e. a linear-time algorithm – as a recursive descent LL(1) parser. Whatever the chosen solution, explain it in detail in the report. In all cases, 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).

   If your scanner from Part 1 worked correctly, your parser may use it in order to extract the sequence of tokens from the input. Otherwise, you can use the scanner which will be provided on the Université Virtuelle.

   For this part of the project, your program must output on `stdout` *the leftmost derivation* of the input string 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 (cf. Figure 1 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` if you wish but do not modify `LexicalUnit.java`.

Operators	Associativity
- (unary)	right
*, /	left
+, - (binary)	left
>, <, =	left
not	right

Table 1: Priority and associativity of the AlCOl 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, with all the necessary justifications, choices and hypotheses, as well as descriptions of your example files;
- *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 AlCOl. 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 AlCOl 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 five folders:

- `doc` contains the JAVADOC and the PDF report.
- `test` contains all your example AlCOl files.
- `dist` contains an executable JAR **that must be called** `part2.jar`.
- `src` contains your source files.
- `more` contains all other files.

Your implementation must contain:

1. Your scanner (from the first part of the project), or the correction if yours did not work properly;
2. Your parser;
3. An executable that reads the file given as argument and writes on the standard output stream the leftmost derivation, possibly with an options to write/print the parse tree (cf. Figure 1). The command for running your executable must be as follows: `java -jar part2.jar sourceFile.co` (cf. Figure 1 for the option).

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

```
java -jar part2.jar [OPTION] [FILE]
```
e.g., `java -jar part2.jar sourceFile.co`
e.g., `java -jar part2.jar -wt tree.tex sourceFile.co`

Figure 1: The generic form of the command for running your executable, as well as two examples.

`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, 23rd**. You are allowed to work in group of maximum two students.

[1] <Program> → `begin` <Code> `end`
[2] <Code> → ε
[3] → <InstList>
[4] <InstList> → <Instruction>
[5] → <Instruction> `;` <InstList>
[6] <Instruction> → <Assign>
[7] → <If>
[8] → <While>
[9] → <For>
[10] → <Print>
[11] → <Read>
[12] <Assign> → [VarName] `:=` <ExprArith>
[13] <ExprArith> → [VarName]
[14] → [Number]
[15] → `(` <ExprArith> `)`
[16] → `-` <ExprArith>
[17] → <ExprArith> <Op> <ExprArith>
[18] <Op> → `+`
[19] → `-`
[20] → `*`
[21] → `/`
[22] <If> → `if` <Cond> `then` <Code> `endif`
[23] → `if` <Cond> `then` <Code> `else` <Code> `endif`
[24] <Cond> → `not` <Cond>
[25] → <SimpleCond>
[26] <SimpleCond> → <ExprArith> <Comp> <ExprArith>
[27] <Comp> → `=`
[28] → `>`
[29] → `<`
[30] <While> → `while` <Cond> `do` <Code> `endwhile`
[31] <For> → `for` [VarName] `from` <ExprArith> `by` <ExprArith> `to` <ExprArith> `do` <Code> `endfor`
[32] <Print> → `print(`[VarName]`)`
[33] <Read> → `read(`[VarName]`)`

Figure 2: The AlCOl grammar.