Yet Another Language for the Compiler Course
Introduction to language theory and compiling
Project – Part 2

Gilles Geeraerts Arnaud Leponce

2025-2026

1 Work description

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

1. Transform the yaLcc grammar (see Table 2 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. Also note that while the characters "`(`" and "`)`" act as regular parentheses in arithmetic expressions, the character "`|`" has the same role but for boolean expressions (conditions).
   (c) Remove left-recursion and apply factorization 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

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

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

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`.

Remark on `->`: The implication operator can be found in the literature to follow several conventions regarding its associativity. Here we chose the most common right associativity, which means that p`->`q`->`r is to be understood as (and therefore parsed as) p`->`(q`->`r).

2 Expected outcome

You must hand in all files required to compile and evaluate your code, as well as the proper documentation, including a **PDF report**. This must be structured into five folders as follows, and a `Makefile` must be provided.

`src/` Contains all source files required to evaluate and compile your work:

- the JFlex source file (yours or the solution) `LexicalAnalyzer.flex` for the scanner;
- the generated scanner file `LexicalAnalyzer.java`;
- provided files `Symbol.java`, `LexicalUnit.java`, and `ParseTree.java` (which can be modified);
- the source code of your parser in a Java source file called `Parser.java`, as well as all the auxiliary classes you created;
- a `Main.java` file calling the lexical analyzer that reads the file given as argument and writes on the standard output stream the leftmost derivation (and optionally produce the parse tree).

`doc/` Contains the JAVADOC and the PDF report.

The PDF report should contain the modified grammar (remember to enumerate the rules!) and action table. It should present your work, with all the necessary justifications, choices and hypotheses, as well as descriptions of your example files. Such report will be particularly useful to get you partial credit if your tool has bugs.

`test/` Contains all your example YALCC files. It is necessary to provide at least one example files of your own.

`dist/` Contains an executable JAR called `part2.jar`. The command for running your executable should therefore be: `java -jar part2.jar [OPTION] [FILE]`. In practice that would be one of the following:

- `java -jar part2.jar sourceFile.ycc`
- `java -jar part2.jar -wt sourceFile.tex sourceFile.ycc`

`more/` Contains all other files.

`Makefile` At the root of your project (i.e. not in any of the aforementioned folders). There is an example Makefile on the UV.

You will compress your root folder (in the *zip* format—no *rar* or other format), **which is named 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); you are allowed to work in a group of maximum two students.

The *zip* file shall be submitted on Université Virtuelle before **November 21st, 2025, 23:59**, Brussels Grand-Place time.

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 yaLcc. Ask us before, just to check that the feature in question is both relevant and doable.

[1]	<Program>	→ `Prog` [ProgName] `Is` <Code> `End`
[2]	<Code>	→ <Instruction> `;` <Code>
[3]		→ ε
[4]	<Instruction>	→ <Assign>
[5]		→ <If>
[6]		→ <While>
[7]		→ <Call>
[8]		→ <Output>
[9]		→ <Input>
[10]	<Assign>	→ [VarName] `=` <ExprArith>
[11]	<ExprArith>	→ [VarName]
[12]		→ [Number]
[13]		→ `(` <ExprArith> `)`
[14]		→ `-` <ExprArith>
[15]		→ <ExprArith> <Op> <ExprArith>
[16]	<Op>	→ `+`
[17]		→ `-`
[18]		→ `*`
[19]		→ `/`
[20]	<If>	→ `If {` <Cond> `} Then` <Code> `End`
[21]		→ `If {` <Cond> `} Then` <Code> `Else` <Code> `End`
[22]	<Cond>	→ <Cond> `->` <Cond>
[23]		→ `|`<Cond>`|`
[24]		→ <ExprArith> <Comp> <ExprArith>
[25]	<Comp>	→ `==`
[26]		→ `<=`
[27]		→ `<`
[28]	<While>	→ `While {`<Cond>`} Do` <Code> `End`
[29]	<Output>	→ `Print(`[VarName]`)`
[30]	<Input>	→ `Input(`[VarName]`)`

Table 2: The yaLcc original grammar.