

*Genial Imperative Language for Learning
and the Enlightenment of Students*
Introduction to language theory and compiling
Project – Part 3

Gilles GEERAERTS

Mrudula BALACHANDER

Mathieu SASSOLAS

Info-F403 – 2024-2025

Work description

For this third and last part of the project, we ask you to augment the recursive-descent LL(1) parser you have written¹ during the first and second parts to let it *generate code* that corresponds to the semantics of the GILLES program that is being compiled (those semantics are informally provided in Appendix A). The output code must be LLVM intermediary language (LLVM IR), which you studied during the practicals. This code must be accepted by the `llvm-as` tool, so that it can be converted to machine code. It is **not allowed** to first compile to another language (*e.g.* C) and then use the language compiler to get LLVM IR code.

When generating the code for arithmetic and boolean expressions, pay extra attention to the associativity and priority of the operators.

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 `LexicalAnalyzer.flex` for the scanner;
- all necessary java files, either generated (`LexicalAnalyzer.java`), provided (and maybe modified, such as `LexicalUnit.java`, `Symbol.java...`), or written on your own.
- a `Main.java` that reads the file given as argument and writes on the standard output stream the LLVM intermediary code. **Do not write any output from Part 1 or 2 on the standard output stream; only your LLVM code!**

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

The PDF report 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.

¹If you are not satisfied with your own parser you can use the correction provided on the UV.

test/ Contains all your example GILLES files. You must provide enough examples to convince of the correctness of your compiler on the whole GILLES language.

dist/ Contains an executable JAR called `part3.jar`. The command for running your executable should therefore be: `java -jar part3.jar [FILE]`. Remember that it should only output the LLVM code and nothing else!

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: `Part3_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 **December 22st, 2024, 23:59**, Brussels Grand-Place time.

Bonus

For this last part of the project, you can enrich GILLES with several features:

- syntactic sugar/simple extensions, *e.g.* FOR loops;
- functions (you can for example introduce new keywords `CALL` and allow them to appear multiple times);
- additional types: boolean, float numbers and strings, or even arrays, lists, ...
- recursive functions;

They are sorted by increasing difficulty, but you can choose any, or all. If you would like to add a feature that is not in the list, tell us first. You can also provide compiling optimizations (*dead code elimination, inlining, ...*), but this is less rewarding for you² since such optimizations are most likely provided by LLVM.

You have entire freedom of implementation for these bonuses; in particular, you can enrich the keywords and syntax, as long as GILLES is a subset of your language (any GILLES program must compile correctly). You are however required to explain what you did and how you did it in your report (we are not supposed to guess how your program works, nor what bonus you implemented), and you must provide test files to demonstrate your additional features. If you have any questions, please send us an email.

This bonus can get you *up to five point*. However, if you obtain more than 100/100, it will *not* be passed on to your exam grade. Also, note that this is only a *bonus*: it is better to provide a working GILLES compiler than a buggy Mega-GILLES one.

²In terms of interest, not in terms of grading: this will give you as many bonus points as the previous ones.

A Informal semantics of the GILLES language

We only provide an informal description of the semantics, since a formal one would needlessly complicate the matter for such a simple language. There is nothing surprising here, since those semantics are similar to the ones of everyday languages. The value to which a nonterminal $\langle NT \rangle$ evaluates will be denoted by $\llbracket NT \rrbracket$, *e.g.* $\llbracket \text{ExprArith} \rrbracket$.

- The code represented by $\langle \text{Program} \rangle$ should be the result of the processing of $\langle \text{Code} \rangle$ (in other words, the LET [ProgName] BE, and END markers are just markers).
- $\langle \text{Code} \rangle$ is a $:-$ -separated list of instructions $\langle \text{Instruction} \rangle$, which should be executed sequentially.
- $\langle \text{Assign} \rangle$: $[\text{VarName}] = \langle \text{ExprArith} \rangle$ means the program should store $\llbracket \text{ExprArith} \rrbracket$ in the variable VarName (which should be stored in a memory location, not simply in an LLVM variable).
- $\langle \text{If} \rangle$: IF { $\langle \text{Cond} \rangle$ } THEN $\langle \text{Code} \rangle$ END means that if the condition computed by $\langle \text{Cond} \rangle$ (*i.e.* $\llbracket \text{Cond} \rrbracket$) is true, then $\langle \text{Code} \rangle$ should be executed, otherwise the program should go to the next instruction.
- $\langle \text{If} \rangle$: IF { $\langle \text{Cond} \rangle$ } THEN $\langle \text{Code1} \rangle$ ELSE $\langle \text{Code2} \rangle$ END means that if $\llbracket \text{Cond} \rrbracket$ is true, then $\langle \text{Code1} \rangle$ should be executed, otherwise $\langle \text{Code2} \rangle$ should be executed instead.
- $\langle \text{While} \rangle$: WHILE { $\langle \text{Cond} \rangle$ } DO $\langle \text{Code} \rangle$ END means that the program should test $\langle \text{Cond} \rangle$, then execute $\langle \text{Code} \rangle$ if $\llbracket \text{Cond} \rrbracket$ is true and then repeat, otherwise it should do nothing³.
- $\langle \text{Output} \rangle$: OUT([VarName]) should print the value of [VarName] to `stdout`.
- $\langle \text{Input} \rangle$: IN([VarName]) should read an integer from `stdin`, and store it in the corresponding [VarName].
- $\langle \text{ExprArith} \rangle$: those are the semantics of usual arithmetic expressions written in infix notation, with the conventional precedence of operators (given in Table 1).
- $\langle \text{Cond} \rangle$: the semantic of comparisons is the usual semantics. The $|$ symbol acts as parenthesis for boolean expressions. The semantics of the \rightarrow implication operator are given in its truth table in Table 2; recall however that it is right-associative, so $\perp \rightarrow \top \rightarrow \perp$ should evaluate to \top .

³Giving formal semantics to `while` is actually not trivial.

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

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

p	q	$p \rightarrow q$
\top	\top	\top
\top	\perp	\perp
\perp	\top	\top
\perp	\perp	\top

Table 2: Truth table for the implication operator (\rightarrow in GILLES).