Yet Another Language for the Compiler Course
Introduction to language theory and compiling
Project – Part 3

Gilles Geeraerts   Arnaud Leponce

2025-2026

Work description

For this third and last part of the project, we ask you to augment the recursive-descent LL(1) parser you have written[1] during the first and second parts to let it *generate code* that corresponds to the semantics of the yaLcc 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 is 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.

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

[1]If you are not satisfied with your own parser you can use the correction that will be provided on the UV after the deadline for Part 2.

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 23rd, 2025, 23:59**, Brussels Grand-Place time.

A Informal semantics of the yaLcc 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 <NT> evaluates will be denoted by ⟦NT⟧, *e.g.* ⟦ExprArith⟧.

- The code represented by <Program> should be the result of the processing of <Code> (in other words, the `Prog` [ProgName] `Is`, and `End` markers are just markers).
- <Code> is a `;`-separated list of instructions <Instruction>, which should be executed sequentially.
- <Assign>: [VarName] `=` <ExprArith> means the program should store ⟦ExprArith⟧ in the variable VarName (which should be stored in a memory location, not simply in an LLVM variable).
- <If>: `If {` <Cond> `} Then` <Code> `End` means that if the condition computed by <Cond> (*i.e.* ⟦Cond⟧) is true, then <Code> should be executed, otherwise the program should go to the next instruction.
- <If>: `If {` <Cond> `} Then` <Code1> `Else` <Code2> `End` means that if ⟦Cond⟧ is true, then <Code1> should be executed, otherwise <Code2> should be executed instead.
- <While>: `While {` <Cond> `} Do` <Code> `End` means that the program should test <Cond>, then execute <Code> if ⟦Cond⟧ is true and then repeat, otherwise it should do nothing[2].
- <Output>: `Print(`[VarName]`)` should print the value of [VarName] to `stdout`.
- <Input>: `Input(`[VarName]`)` should read an integer from `stdin`, and store it in the corresponding [VarName].
- <ExprArith>: those are the semantics of usual arithmetic expressions written in infix notation, with the conventional precedence of operators (given in Table 1).
- <Cond>: the semantic of comparisons is the usual semantics. The `|` symbol acts as parenthesis for Boolean expressions. The semantics of the `->` implication operator are given in its truth table in Table 2; recall however that it is right-associative, so $\bot$`->`$\top$`->`$\bot$ should evaluate to $\top$.

[2]Giving formal semantics to `while` is actually not trivial.

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 (`-`).

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

Table 2: Truth table for the implication operator (`->` in yaLcc).

p	q	p `->` q
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$