# *Yet Another Language for the Compiler Course*
# Introduction to language theory and compiling
# Project – Part 1

Gilles Geeraerts Arnaud Leponce

2025-2026

## 1 Introduction

**Remark** This project can be made by groups of maximum two students.

In this project, you are requested to design and write a compiler for, well, *Yet Another Language for the Compiler Course*, aka yaLcc. The grammar of the language is given in Table 1 (page 2), where reserved keywords have been typeset using `typewriter font`. In addition, [ProgName], [VarName], and [Number] are lexical units, which are defined as follows (where the word *letter* is used for latin unaccented letter, i.e. A to Z). A [ProgName] identifies a program name, which is a string of letters and _ (the underscore character), starting with an uppercase letter (this is case sensitive). A [VarName] identifies a variable, which is a string of digits and letters, starting with a lowercase letter (this is case sensitive). A [Number] represents an integral numerical constant, and is made up of a string of digits only. The minus sign can be generated using rule [14].

Finally, comments are allowed in yaLcc. There are two kinds of comments: Short comments, which are all the symbols appearing after a dollar sign `$` up to the end of the line, and long comments, which are all the symbols occurring between two double exclamation marks `!!` keywords. There is no possibility of nested comments. Observe that comments do not occur in the rules of the grammar: they must be ignored by the scanner, and will not be transmitted to the parser.

Figure 1 shows an example of yaLcc program.

## 2 Assignment - Part 1

In this first part of the assignment, you must produce the *lexical analyzer* of your compiler, using the JFlex tool reviewed during the practicals.

*Please adhere strictly to the instructions given below, otherwise we might be unable to grade your project, as automatic testing procedures will be applied.*

### 2.1 Environment and provided files

The lexical analyzer will be implemented in JAVA. It must recognise the different lexical units of the language, and maintain a symbol table. To help you, several JAVA classes are provided on the UV:

- The `LexicalUnit` class contains an enumeration of all the possible lexical units;
- The `Symbol` class implements the notion of token. Each object of the class can be used to associate a value (a generic Java `Object`) to a `LexicalUnit`, and a line and column number (position in the file). The code should be self-explanatory. If not, do not hesitate to ask questions to the teacher, or to the teaching assistant.

| | | |
|---|---|---|
| [1] | <Program> | → `Prog` [ProgName] `Is` <Code> `End` |
| [2] | <Code> | → <Instruction> `;` <Code> |
| [3] | | → $\varepsilon$ |
| [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 1: The YALCC grammar.

```
!! Euclid's algorithm !!

Prog Euclidean_algorithm Is
  Input(a);
  Input(b);
  While {0 < b} Do
      c = b;
      While {b <= a} Do     $ Computation of modulo
        a = a-b;
      End;
      b = a;
      a = c;
  End;
  Print(a);
End
```

Figure 1: An example YALCC program.

## 2.2 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 `LexicalAnalyzer.flex` for your lexical analyzer;
- the generated file `LexicalAnalyzer.java`;
- provided files `Symbol.java` and `LexicalUnit.java` *without modification*.
- a `Main.java` file calling the lexical analyzer that reads the file given as argument and writes on the standard output stream the sequence of matched lexical units and the content of the symbol table (see below for details on the expected output).

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

Note that the documentation of your code will be taken into account for the grading, especially for Parts 2 and 3, so Part 1 is a good occasion to setup JAVADOC for your project.

The PDF report should contain all regular expressions, and 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 relevant example files of your own.

`dist/` Contains an executable JAR called `part1.jar`. The command for running your executable should therefore be: `java -jar part1.jar sourceFile.gls`

`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 (see the *Jflex practical* folder).

You will compress your root folder (in the *zip* format—no *rar* or other format), **which is named according to the following regexp**: `Part1_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 **October 20$^{\text{th}}$, 2025, 23:59**, Brussels Grand-Place time.

## 2.3 Output of the program

The format of the output of the program must be:

1. First, the sequence of matched lexical units. You must use the *toString()* method of the provided Symbol class to print individual tokens;
2. Then, the word *Variables*, to clearly separate the symbol table from the sequence of tokens;
3. Finally, the content of the symbol table, formatted as the sequence of all recognized variables, in lexicographical (alphabetical) order. There must be one variable per line, together with the number of the line of the input file where this variable has been encountered for the first time (the variable and the line number must be separated by at least one space).

By default, your program must output on `stdout` (the terminal), and not write into a file.

For instance, on the following input:

```
Input(b)
```

your executable must produce exactly, using the `toString()` method of the `Symbol` class, the following output for the sequence of tokens (an example for the symbol table is given hereunder):

```
token: Input          lexical unit: INPUT
token: (              lexical unit: LPAREN
token: b              lexical unit: VARNAME
token: )              lexical unit: RPAREN
```

Note that the *token* is the matched input string (for instance `b` for the third token) while the *lexical unit* is the name of the matched element from the `LexicalUnit` enumeration (`VARNAME` for the third token).

Also, for the example in Figure 1, the symbol table must be displayed as:

```
Variables
a 4
b 5
c 7
```

An example input YALCC file with the expected output is available on Université Virtuelle to test[1] your program.

## 3 Frequently Asked Questions

Here are some questions that we have gotten in the previous years which might help you during your project. If you have further questions please ask in the forum on the UV or via email.

- Q: What should happen if the YALCC file can not be correctly tokenized?

  A: Throw a (useful) error message such as "Unknown symbol detected" or "Long comment not closed" for example. You can assume that you will not encounter a YALCC file that has nested comments.

- Q: What about whitespaces?

  A: Whitespaces are not necessary beyond separating keywords, but they are nice for readability. All extra whitespaces must be ignored. For example:

```
(If abbb Then  x=3:
```

```
( If abbb Then x=3 ;
```

```
(
    If          abbb

    Then
        x = 3
    ;
```

must all yield the same matching[2] of lexical units, namely:

[1] It is by no means extensive!

[2] This is not syntactically correct YALCC code, but the lexing should nonetheless work.

```
token: (            lexical unit: LPAREN
token: If           lexical unit: IF
token: abbb         lexical unit: VARNAME
token: Then         lexical unit: THEN
token: x            lexical unit: VARNAME
token: =            lexical unit: ASSIGN
token: 3            lexical unit: NUMBER
token: ;            lexical unit: SEMI
```

However, there are cases where spaces are required to isolate a keyword: `abbbThenx` should be scanned as one variable name, and therefore, `(IF abbbThenx=3` would not produce the same result as above.

- Q: What is the use of a `Makefile`?

  A: You need to provide a `Makefile` because your executable might not run on our machine, because of different JAVA versions (unavoidable because some use Windows, Linux, MacOS). A working `Makefile` allows us to quickly produce an executable from your code. It takes a considerable amount of time if we need to figure out how to turn your code into an executable by hand. In addition, having a `Makefile` can streamline the testing and debugging of the code, so it is worthwhile to have for your sake as well.