Introduction.

A *compiler* is a program that'll given some code, return either an error or some other code in another language. However, compilation is not *only* about code generation: a large number of compilation techniques are not linked to assembly production. Moreover, languages can be:

- ▷ interpreted (*e.g.* Python sometimes);
- ▷ compiled into an intermediate language that will be interpreted (*e.g.* Java);
- ▷ compiled into another high level language (*e.g.* OCaml can be compiled into JavaScript);
- ▷ compiled "on the fly" (*e.g.* Julia or Python sometimes);
- ▷ several of the above.

A *compiler* will translate a program P into a program Q such that, for all entry, the output of Q is the same as the output of P. An *interpreter* is a program that, given a program P and an entry x, computes the output of P on x. This can be seen, in a way, we swap two quantifiers:

$$\underbrace{\forall P, \exists Q, \forall x, \dots}_{\text{compiler}} \qquad \text{and} \qquad \underbrace{\forall P, \forall x, \exists s, \dots}_{\text{interpreter}}.$$

The quality of a compiler can be measured on multiple factors: its correctness, the efficiency of the generated code, its own efficiency. We will also touch on program analysis.

The goal of the labs will be to write a compiler for the RISCV architecture. This part will be done in Python.

1 The RISCV architecture.

RISCV is an open-source architecture that is extensible. One of the basic components of RISCV are *registers*. We can manipulate registers with operations such as `add`[1] or `addi`.[2]

We can also do *branching* in RISCV in two kinds:

- ▷ *unconditional branching* ($\rightsquigarrow$ jump and link) is done with `jal`;
- ▷ *test and branch* ($\rightsquigarrow$ branch if lower than) is one with `blt`.

The first one is used to implement functions, and the other one (and variants) is used to implement an `if`.

All the details of the RISCV operations can be found at:

https://github.com/Drup/cap-lab25/blob/main/course/riscv_isa.pdf.

We have an assembly language. This will be the last part of our compiler.

2 Lexical Analysis.

Lexical Analysis breaks down the code in tokens, known as *lexems*. Here, we use *regular expressions*. In our case, the tool we will use for our compiler is ANTLR.

[1] Adds the data from two registers into another register.
[2] Adds the data from one register with a constant, into another register