Interpretation and Compilation of Languages

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

March 10, 2025

Lecture 1

based on lectures by Jean-Christophe Filliâtre and Léon Gondelman
previous editions by João Costa Seco, Luís Caires, and Bernardo Toninho

Before we begin

Meta-language for the lectures: English or Portuguese?

All provided documents are written in English.

Who am I?

- Mário Pereira `mjp.pereira@fct.unl.pt`
- Office hours: Wednesday at 2pm – office 243, Ed. II
- This is my field of expertise. Do talk to me and put questions!
  - There are many MSc opportunities in this field
  - Some of them with funding

Course Organization

13 blocks, every Thursday from March 11 to June 3

- lectures: 9:00 - 11:00
- lab sessions: 11:00 - 18:00
  - practical lab sessions
  - whiteboard/handwritten exercises
  - support with the project (these will be majority)

The plan

Evaluation plan:

Midterm: 26th April (Saturday)

Final test: 2nd June (Monday)

Project: 7th June (Saturday)

The project:

A compiler from a subset of a programming language into machine code (assembly)

- I am not going to lie: this project is a considerable amount of work.
- That is why the majority of lab sessions are project support.
- I am here to help you!

Evaluation components:

1. *teórico-prática* (TP): one midterm ($T1$) + one final test ($T2$)
2. *prática* (P): one project (groups of two)
3. final exam (Ex)

Evaluation formula:

$$\begin{array}{lcll} TP & = & (T1+T2)/2 \ \text{ OR } \ Ex & \\ F & = & TP, & \text{if } TP < 9.5 \\ F & = & 0.7\,TP + 0.3P & \text{if } TP \geq 9.5 \end{array}$$

Frequência:

$$\begin{array}{lcl} TP & \geq & 9.5 \\ P & \geq & 9.5 \end{array}$$

Course Website

`https://icl-2025.github.io/`

- lectures slides, lab sessions problems & solutions
- various resources: tools, further reading, web sites, etc.

Discord: announcements and quick communication.

Today: Course Overview

1. What is a compiler?
2. Frontend of a compiler
3. Backend of a compiler
4. Interpretation vs Compilation
5. OCaml for the design and implementation of compilers

Course Objectives

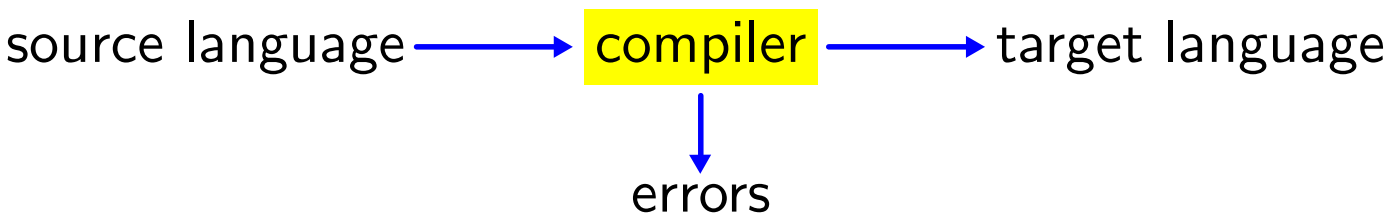
Understand the mechanisms behind compilation, that is, the translation from one language to another.

Understand the various aspects of programming languages via their compilation.

Earn practical skills how to use existing compilation tools that help to write a compiler.

Compilation Overview

A compiler ranslates a "program" from a source language to a target language, possibly signaling errors.



Frontend and Backend

Typically, the compiler decomposes into

a frontend

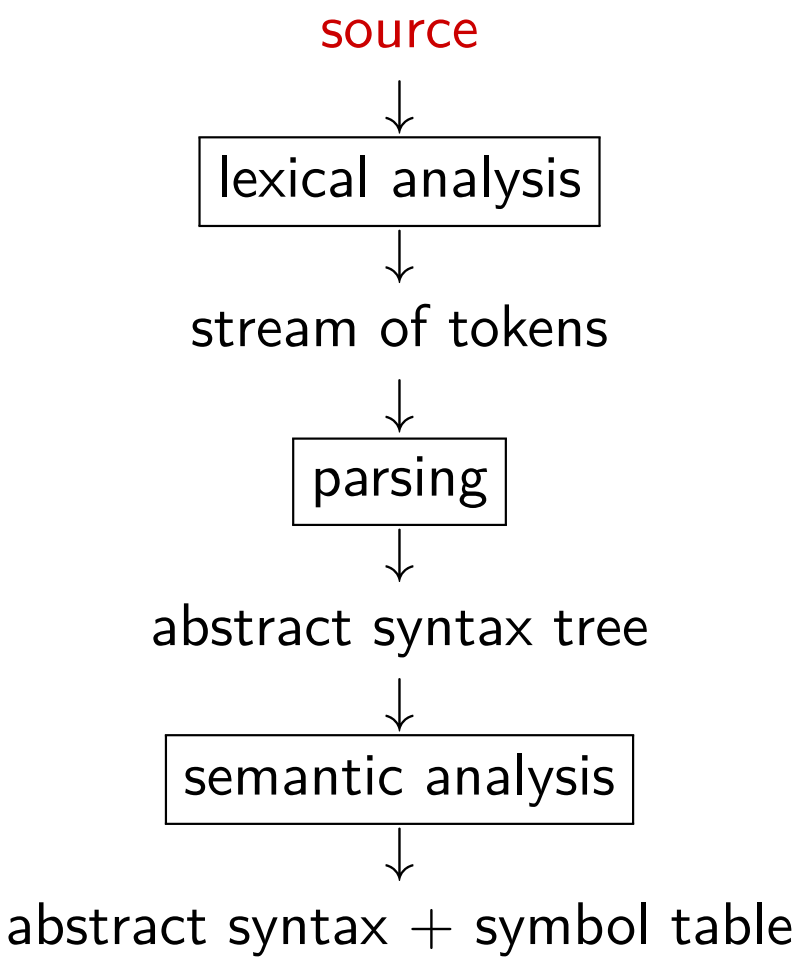
- recognizes the program and its meaning
- signals errors and thus can fail
  (syntax errors, scoping errors, typing errors, etc.)

and a backend

- produces the target code
- uses many intermediate languages
- must not fail

Overview of the frontend

(first part of this course)



source code = sequence of characters

Intuitively, the texts

```
2*(x+1)
```

and

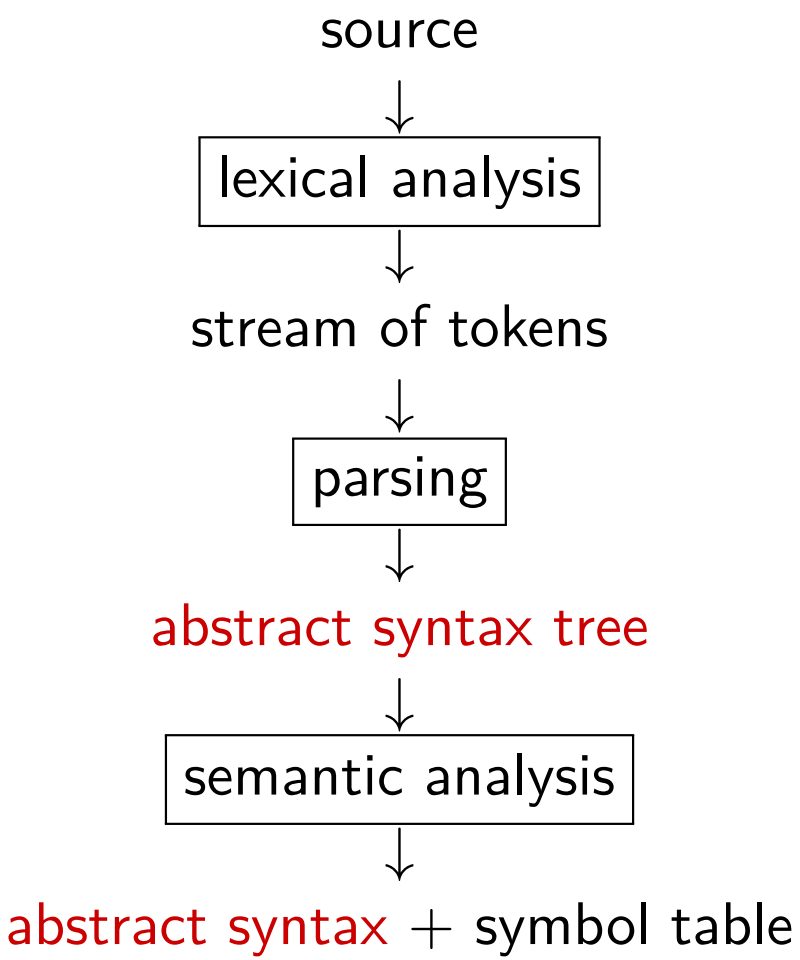
```
(2 * ((x) + 1))
```

and

```
2 * /* I double */ ( x + 1 )
```

all map to the same "program" we have in mind that computes $2(x+1)$

What is a program?



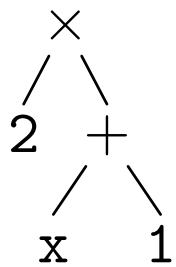
As a syntactic object (sequence of characters), a program is too complex to apprehend

That is why we switch to abstract syntax of a given programming language

We say that all of the

```
2*(x+1)     2*((x) + 1)     ...
```

all map to the same abstract syntax tree.



We define an abstract syntax using a grammar

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{addition} \\
 & | & e \times e & \textit{multiplication} \\
 & | & \ldots &
\end{array}
$$

It reads "an expression, noted e, is

- either a constant c,
- either a variable x,
- either the addition of two expressions,
- etc."

From Source to Abstract Syntax

A grammar

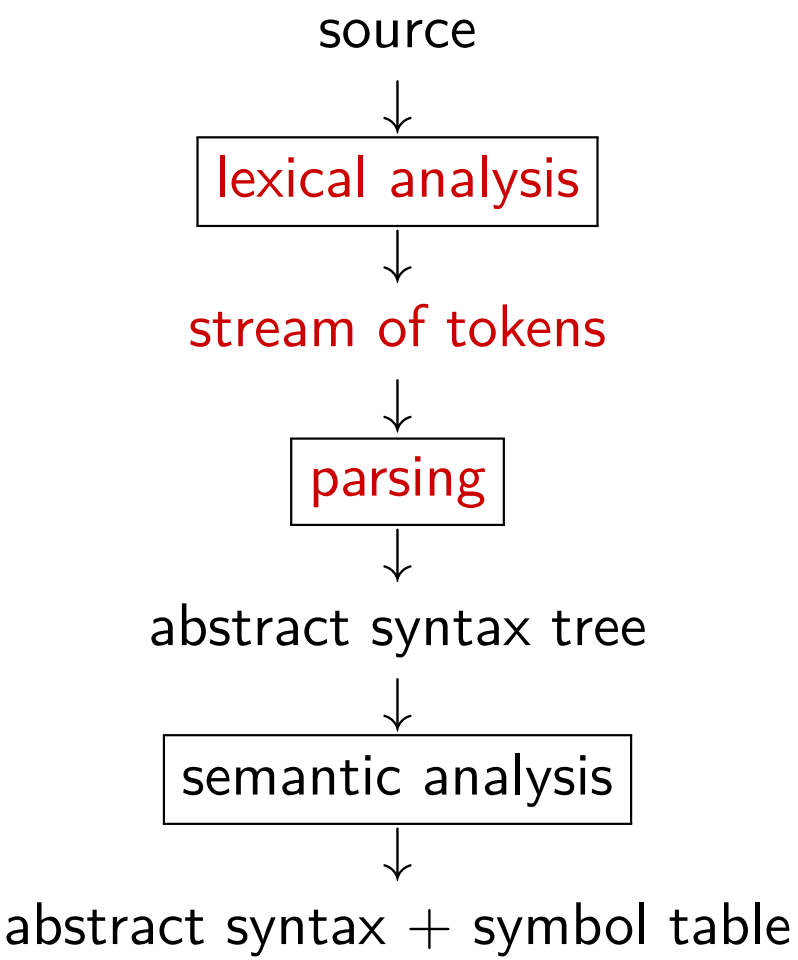
- helps us to define rules of the source language of compiler
- can itself be encoded within a given programming language

But how to go

- from the source text of a concrete given program
- to its abstract syntax tree representation?

For this, we need to master the art of parsing.

Parsing



Identifies the programs that belong to the syntax of the language.

Its input is a sequence of characters and its output is abstract syntax.

Parsing is split into two steps

- lexical analysis, which splits the input in "words" called tokens
- syntax analysis, which recognizes legal sequences of tokens

Which Tools?

To implement lexical analysis, we are going to use

- regular expressions to describe tokens
- finite automata to recognize them

To implement syntax analysis, we are going to use

- a context-free grammar to define the syntax
- a pushdown automaton to recognize it

Overview of the backend

(second part of this course)

Compilation to Machine Language

Compilation typically involves translating a high-level language (C, JAVA, OCAML, etc.) to some machine language.

```
% gcc -o sum sum.c
```


```
int main(int argc, char **argv) {
  int i, s = 0;
  for (i = 0; i <= 100; i++) s += i*i;
  printf("0*0+...+100*100 = %d\n", s);
}
```

⟶

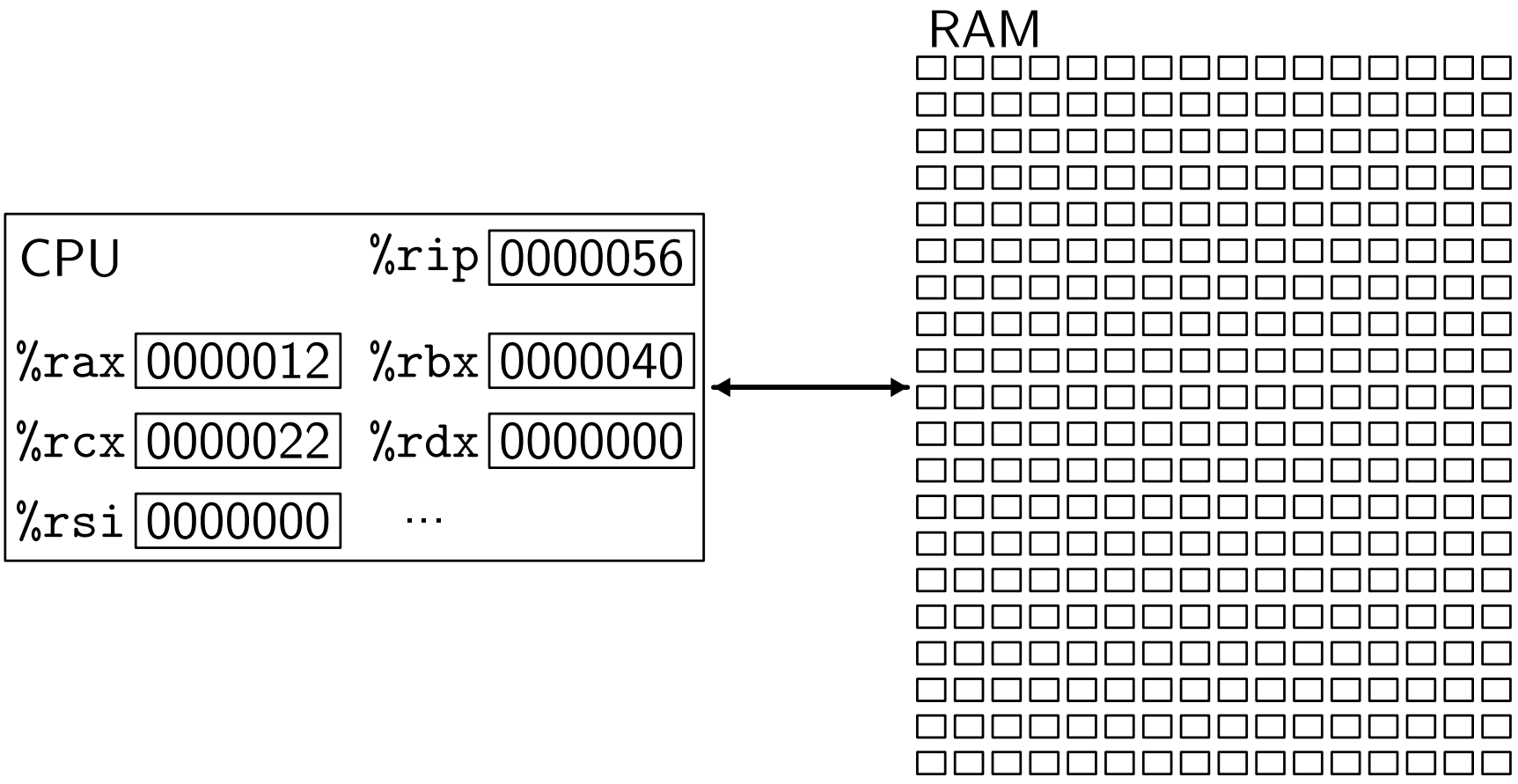
```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
...
```

A Little bit of Architecture

Roughly speaking, a computer is composed

- of a CPU, containing
  - few integer and floating-point registers
  - some computation power
- memory (RAM)
  - composed of a large number of bytes (8 bits)
    for instance, 1 GiB $= 2^{30}$ bytes $= 2^{33}$ bits, that is $2^{2^{33}}$ possible states
  - contains data and instructions

A Little bit of Architecture



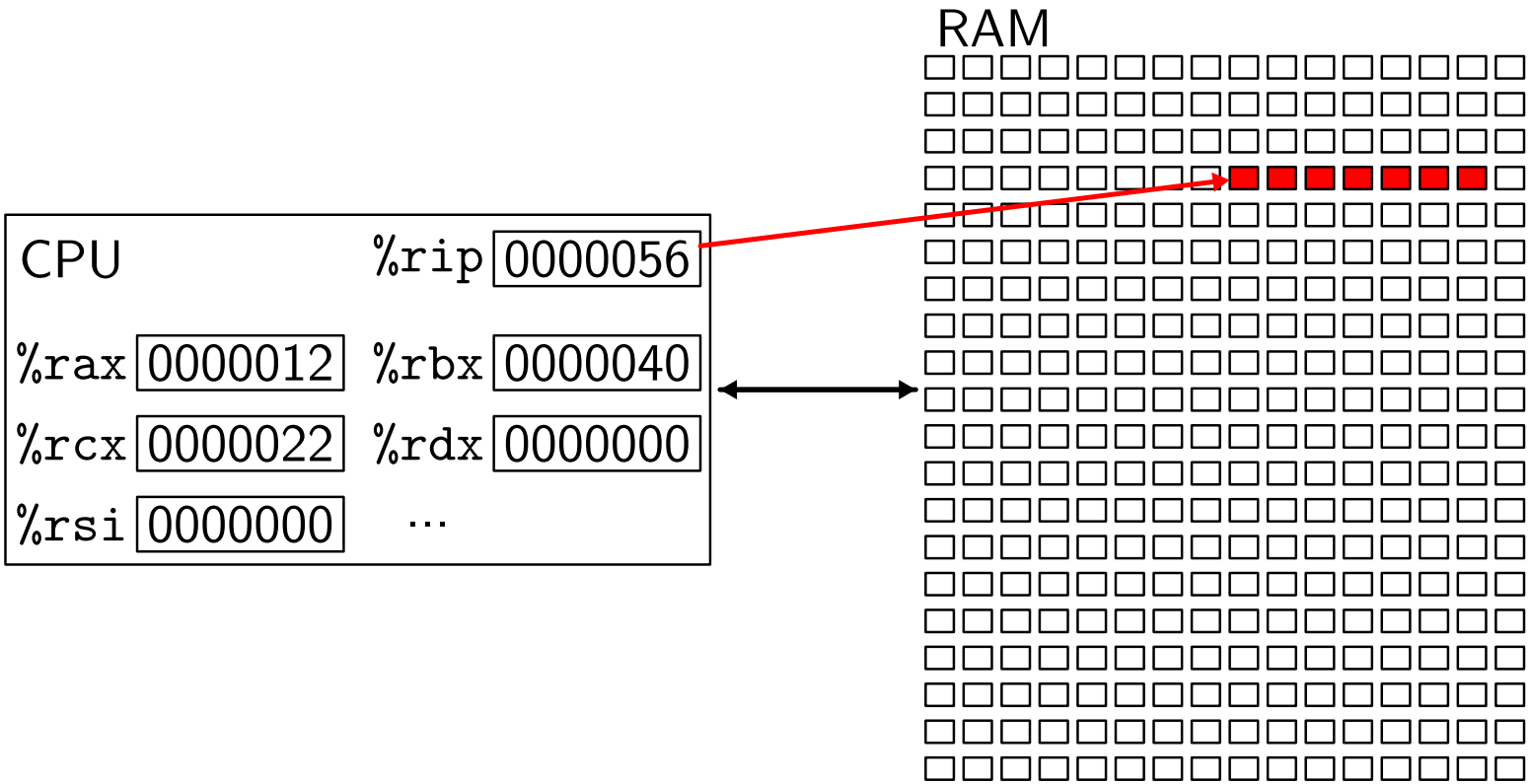
accessing memory is costly (at one billion instructions per second, light only travels 30 centimeters!)

Execution Principle

execution proceeds according to the following:

- a register (`%rip`) contains the address of the next instruction to execute
- we read one or several bytes at this address (*fetch*)
- we interpret these bytes as an instruction (*decode*)
- we execute the instruction (*execute*)
- we modify the register `%rip` to move to the next instruction (typically the one immediately after, unless we `jump`)

Execute Principle



instruction:	48	c7	c0	2a	00	00	00
decoding:		movq	%rax	42			

i.e. store 42 into register %rax

Two Main Families of Microprocessors

- CISC (*Complex Instruction Set*)
  - many d'instructions
  - many addressing modes
  - many instructions read / write memory
  - few registers
  - examples: VAX, PDP-11, Motorola 68xxx, Intel x86
- RISC (*Reduced Instruction Set*)
  - few instructions
  - few instructions read / write memory
  - many registers
  - examples: Alpha, Sparc, MIPS, ARM

Machine Code and Assembly

Machine code can be executed only on the microprocessor it was compiled for (e.g., Intel x86-64, Apple M1, ...)

We do not compile directly into machine code, but into an assembly language providing several facilities

- symbolic names
- allocation of global data

Assembly language is turned into machine code by a program called an assembler (a compiler)

Target Language

In this course, we are going to consider compiling to assembly, indeed, but this is only one aspect of compilation.

Many techniques used in compilers are not related to the production of assembly code.

Some languages are instead

- compiled into some intermediate language, which is then interpreted (JAVA, Python, OCAML, Scala, etc.)
- interpreted (Basic, COBOL, Ruby, etc.)
- just-in-time compiled (Julia, etc.)
- compiled into another high-level language

Difference Between a Compiler and an Interpreter

A compiler translates a program P into a program Q such that for any input x, the output of $Q(x)$ is identical to that of $P(x)$

$$\forall P \; \exists Q \; \forall x \ldots$$

An interpreter is a program that, given a program P and some input x, computes the output s of $P(x)$

$$\forall P \; \forall x \; \exists s \ldots$$

Difference Between a Compiler and an Interpreter

Said otherwise,

the compiler performs a more complex task only once, to produce a code that accepts any input.

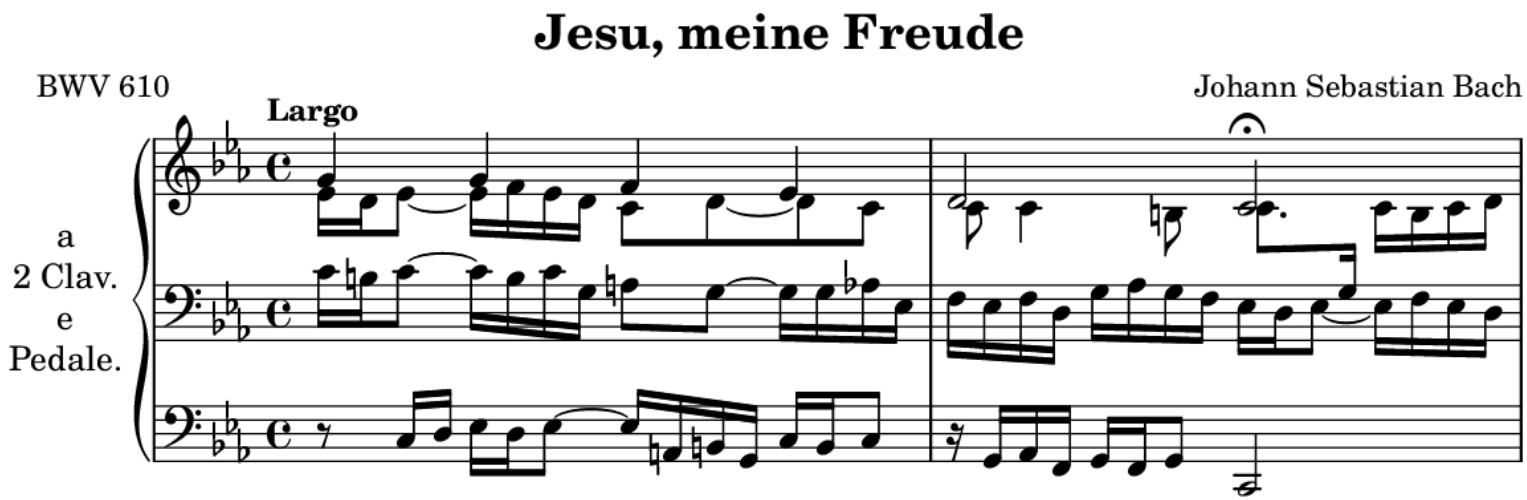
The interpreter performs a simpler task, but repeats it for every input.

Another difference: compiled code is typically more efficient than interpreted code.

Example of Compilation and Interpretation

source → `lilypond` → PDF file → `evince` → image

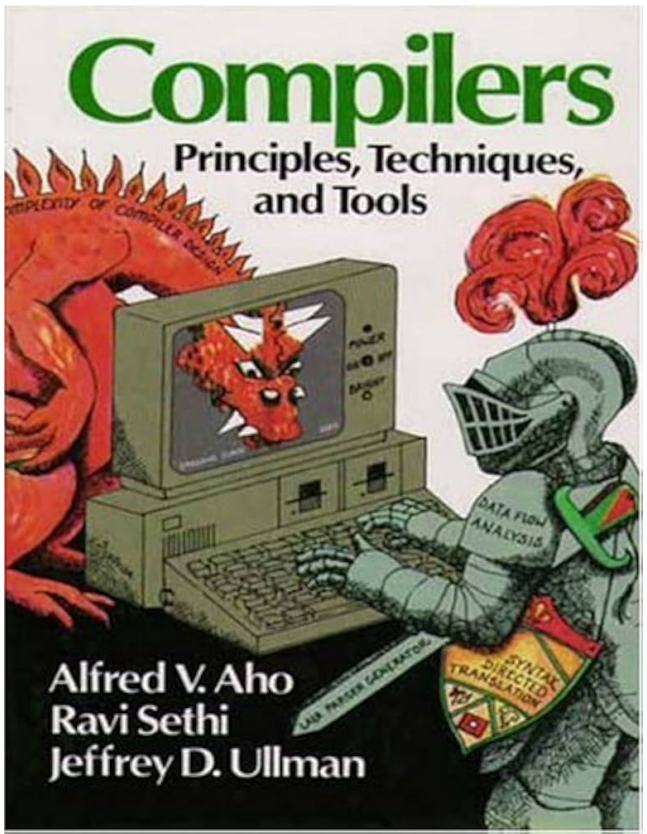
```
\new PianoStaff
   { \set PianoStaff.instrumentName = \markup { \large \center-column {
       "a" "" "2 Clav." "" "e"  ""  "Pedale." } \hspace #0.5 }
   <<
      \context Staff = right
      {\context Voice = right \right}
      \context Staff = left {\context Voice = left \left}
      \context Staff = pedal {\context Voice = pedal \pedal}
   >>
```



Quality of a Compiler

How can we evaluate the quality of a compiler?

- its soundness
- the performance of the compiled code
- the performance of the compiler itself



"Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct."

(Dragon Book, 2006)

How to Implement a Compiler

A Compiler is a Program Itself

We must choose a programming language in which we will implement a compiler, *i.e.*, a language in which we can:

- model the abstract syntax of the source language
- use existing/implement parsing tools (lexer, parser)
- generate the target language code
- write an interpreter, type checker, static analyser, etc.

In this ourse we will use OCaml programming language which is well-suited for programming language design.

Why OCaml?

First preliminary note: there will be support material written in JAVA.

That said, JAVA is not the best tool to build compilers.

Functional programming languages are the right tool for the job.
here, we choose OCAML.

Do not believe me? Ask your colleagues from last year.

OCaml comes with

- imperative style features and control structures (e.g. loops, exceptions)
- garbage collection for automatic memory management (similar to JAVA)
- static type-checking to increase performance and reduce the number of runtime errors, as found in JAVA and C#

But it is also and foremost a functional programming language:

- first-class functions that an be passed around like ordinary values, as seen in JavaScript, Common Lisp.
- algebraic data types and pattern matching to define and manipulate complex data structures, also available in Scala, F#, or Haskell
- immutable programming, i.e., programming without making destructive updates to data structures, as seen in languages like Scheme
- parametric polymorphism to onstruct abstractions that work across different data types, akin to generics in JAVA and C# and templates in C++

Your choice!

You choose how to write your compilers, but if you do it in OCaml:

- lectures are in OCaml
- labs are in OCaml (you can reuse code for the project)
  - Do not panic just yet: there are also lab material in Java:)
- OCaml is arguably good to write compilers (have I said this before?)
- you master a new programming language/way of programming.

there are no good programming languages,
only good programmers

(there are bad programming languages, tough...)

Let's make a deal, shall we?

I hear and I forget. I see and I remember. I do and I understand.
Confucius

Simply want to succeed? Lecture notes and exercise sheets are enough.

Want to excel? Go beyond lectures and lab sessions!

- read the books
- do more exercises on your own
- talk to the teacher
- use the Discord server

Let's avoid the passive attitude during lectures:

- I will write a lot on the board. Bring your notebook.
- I will do many live demos. Bring and use your laptop.

That said, I am planning on organizing many extra hands-on sessions.

Homework

For next week, your homework is
to read the introductory overview slides about OCaml

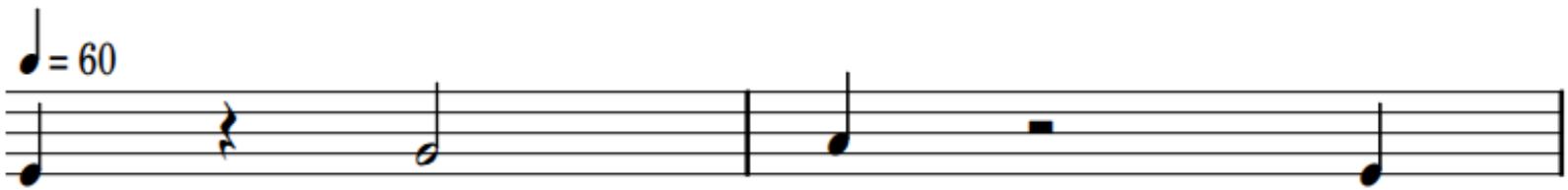
https://icl-2025.github.io/intro_ocaml.pdf

and go **through the second chapter of the following book**

https://usr.lmf.cnrs.fr/lpo/lpo.pdf

- You do not need to read/solve every single example/exercise
- focus on getting familiar with the concepts of each section
- start with the first 60 pages
- next week, the remaining 60 pages

It's Demo Time!



Our Symbols

Two kinds of symbols:

- Notes: pitch on the score + duration
  - Sixteenth
  - Eighth
  - Quarter
  - Half
  - Full
- Silence: duration

An extra ingredient: the tempo

- the number of quarter notes per minute

Frequency of Notes

To play the notes, we need to know their frequency (in Hz).

Two pieces of information:

- main note:

Frequency of Notes

To play the notes, we need to know their frequency (in Hz).

Two pieces of information:

- main note: *Do, Re, Mi, Fa, Sol, La, Si*
- the octave: 0, 1, 2, 3, etc.

Compute frequency:

$$f = f_0 \times 2^o$$

Note	Frequency (Hz) for octave 0
Do	32.70
Re	36.71
Mi	41.20
Fa	43.65
Sol	49.
La	55.
Si	61.74

Playing Sounds

To play a single note, we will use the `play` command from `SoX` package.

From OCaml, we will be using the `Unix` library to

- create a process
- interact with the operative system

```
play -r 44100 -n synth D sin F
```

D `duration`

F `frequency`

Bibliography

- Learn Programming with OCaml – Algorithms and Data Structures, S. Conchon & J-C. Filliâtre (translated by U. Nair), 2025. `https://usr.lmf.cnrs.fr/lpo/lpo.pdf`