École Polytechnique

CSC_52064 – **Compilation**

Jean-Christophe Filliâtre

introduction

9 blocks, from January 6 to March 10

- **lecture** 2:00–4:00 pm
  - slides / lecture notes on moodle
- **lab** 4:15–6:15 pm in lab rooms 32–33
  - with Wendlasida Ouedraogo and myself

- a **written exam** (March 17, 2:00–5:00 pm)
- a **project** = a tiny compiler to x86-64
  - during the labs (starting lab 4) and outside
  - alone or in pair
  - using Java or OCaml

$$\mathit{grade} = \frac{\mathit{exam} + \mathit{project}}{2}$$

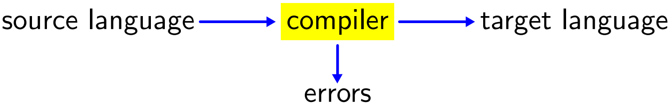
https://www.enseignement.polytechnique.fr/informatique/CSC_52064/

- archives of the exams
- various resources: tools, further reading, web sites, etc.

understand the mechanisms behind **compilation**,
that is, the translation from one language to another

understand the various aspects of **programming languages**
via their compilation

a compiler translates a "program" from a **source** language to a **target** language, possibly signaling errors



compilation typically evokes translating a high-level language (C, Java, OCaml, etc.) to some machine language

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

source `sum.c` ⟶ C compiler (gcc) ⟶ executable `sum`

```
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
...
```

in this lecture, 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

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

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

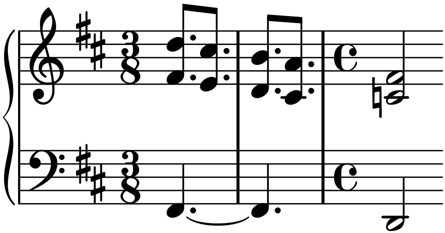
the interpreter performs a simpler task, but repeats it for every input

another difference: compiled code is typically more efficient than interpreted code

example of compilation and interpretation

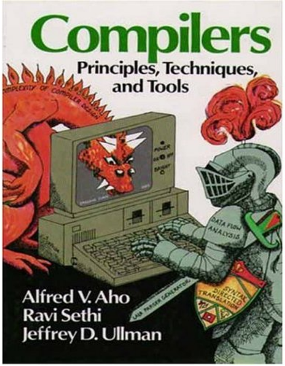
source ⟶ `lilypond` ⟶ PDF file ⟶ `evince` ⟶ image

```
\new PianoStaff <<
  \new Staff { \clef "treble" \key d \major \time 3/8
               <<d8. fis,8.>> <<cis'8. e,8.>> | ... }
  \new Staff { \clef "bass" \key d \major
               fis,,4. ~ | fis4. | \time 4/4 d2 }
>>
```



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)

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

source

↓

$\boxed{\text{lexical analysis}}$

↓

stream of tokens

↓

$\boxed{\text{parsing}}$

↓

abstract syntax tree

↓

$\boxed{\text{semantic analysis}}$

↓

abstract syntax + symbol table

backend

abstract syntax

↓

code production (several phases)

↓

assembly code

↓

assembler (`as`)

↓

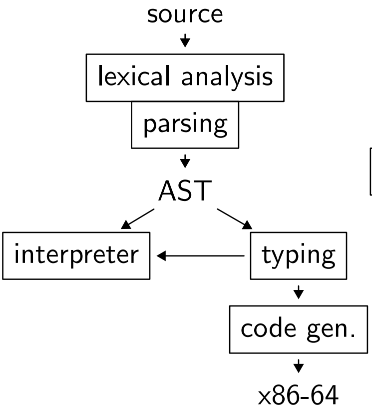
machine language

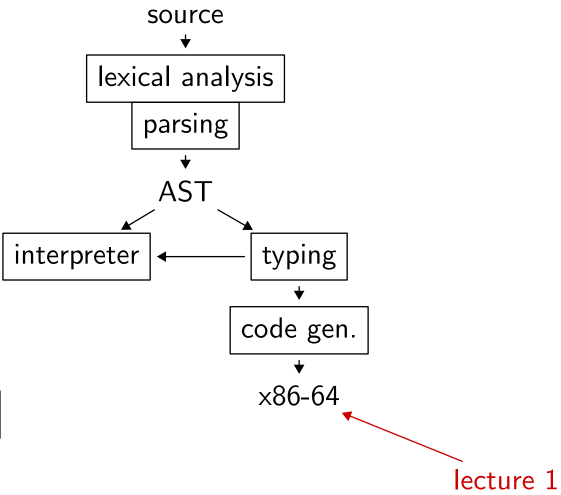
↓

linking (`ld`)

↓

executable


source
lexical analysis
parsing
AST
interpreter
typing
code gen.
x86-64


source
lexical analysis
parsing
AST
interpreter
typing
code gen.
x86-64
lecture 1