INF564 – Compilation

Jean-Christophe Filliâtre

introduction


8 blocks, from January 9 to March 6

- **lecture** 2pm–4pm
  - slides / lecture notes on moodle

- **lab** 4pm15–6pm15
  - with Wendlasida Ouedraogo and myself
• a **written exam** (March 13, 2pm–5pm)

• a **project** = a mini C to x86-64 compiler
  • during the labs (starting lab 4) and outside
  • alone or in pair
  • using Java or OCaml

\[
\text{grade} = \frac{\text{exam} + \text{project}}{2}
\]
we use **slack** for communication

- inf564-2023.slack.com (you have received an invitation)
- for the labs, on channel #td
- otherwise channel #général or #projet
- direct messages if needed

you can also use slack between students (using direct messages)
e.g. when working in pair on the project
https://www.enseignement.polytechnique.fr/informatique/INF564/

- 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

diagram:

source language → compiler → target language

down arrow:

errors
compilation to machine language

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

```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);
}
```

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

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

```
0010011110111101111111111111100000
1010111110111111000000000000010100
101011111010010000000000000010000
101011111010101000000000000100100
10101111101001000000000000011000
10101111101000000000000000011100
10101111101011100000000000011100
...```
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 $P(x)$

$$\forall P \exists Q \forall x...$$

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

$$\forall P \forall x \exists s...$$
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.>> | ... }\n\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 ↓

lexical analysis

stream of tokens ↓

parsing

abstract syntax tree ↓

semantic analysis

abstract syntax + symbol table
abstract syntax
↓
code production (several phases)
↓
assembly code
↓
assembler (as)
↓
machine language
↓
linking (ld)
↓
executable
overview of the course

source

lexical analysis

parsing

AST

interpreter

typing

instr. sel.

RTL

ERTL

LTL

x86-64

source

lexical analysis

parsing

AST

interpreter

typing

instr. sel.

RTL

ERTL

LTL

x86-64