Introduction - CAP Course

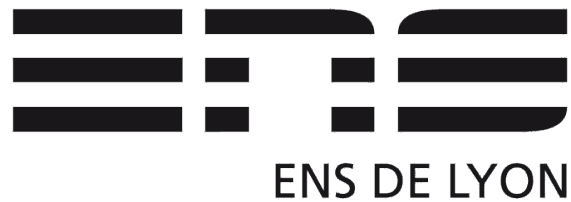
Laure Gonnord

http://laure.gonnord.org/pro/teaching/capM1.html

Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

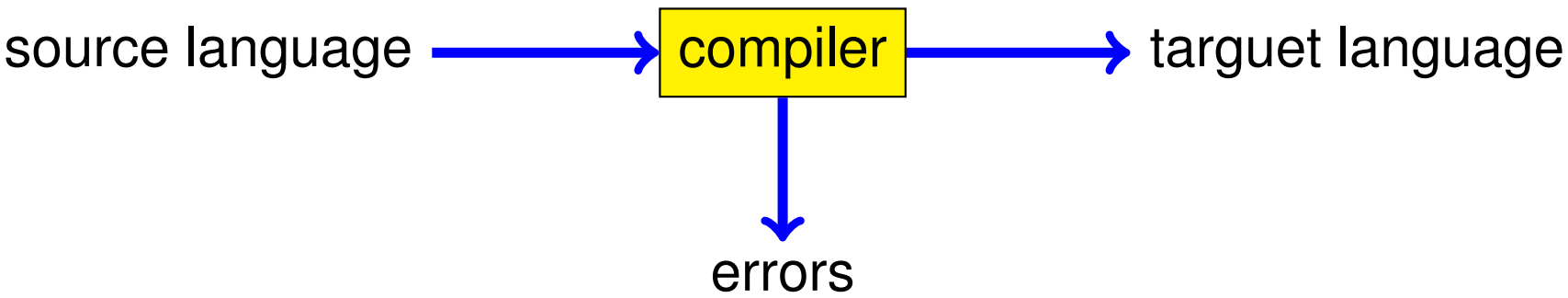
2018-2019



Credits

A large part of the compilation part of this course is inspired by the Compilation Course of JC Filliâtre at ENS Ulm who kindly offered the source code of his slides.

What's compilation ?



Compilation toward the machine language

We immediatly think of the translation of a high-level language (C,Java,OCaml) into the machine language of a processor (Pentium, PowerPC...)

```
% 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);
}
```

$\longrightarrow$

```
00100111101111011111111111000001010111110111111000000000010100
10101111010010000000000010000010101111010010100000000100100
10101111010000000000000001100010101111010000000000000011100
10001111010111000000000011100
```

Target Language

This aspect (compilation into assembly) will be presented in this course, but we will do more :

Compilation is not (only) code generation

A large number of compilation techniques are not linked to assembly code production.

Moreover, languages can be

- interpreted (Basic, COBOL, Ruby, Python, etc.)
- compiled into an intermediate language that will be interpreted (Java, OCaml, Scala, etc.)
- compiled into another high level language (or the same !)
- compiled "on the fly" (or just on time)

Compiler/ Interpreter

- A compiler translates a program P intro a program Q such that for all entry x, the output $Q(x)$ is the same as $P(x)$.

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

- An interpreter is a program that, given a program P and an entry x, computes the output of $P(x)$:

$$\forall P\ \forall x\ \exists s...$$

Compiler vs Interpreter

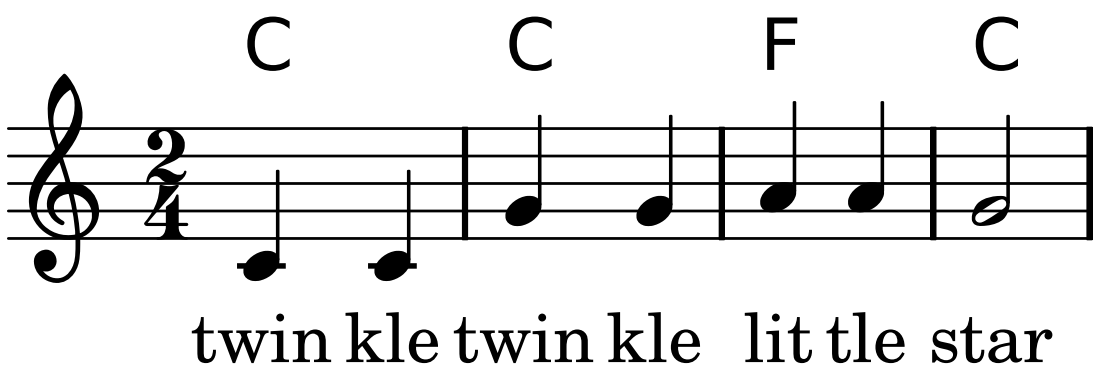
Or :

- The compler makes a complex work once, to produce a code for whatever entry.
- An interpreter makes a simpler job, but on every entry.

- In general the code after compilation is more efficient.

Example

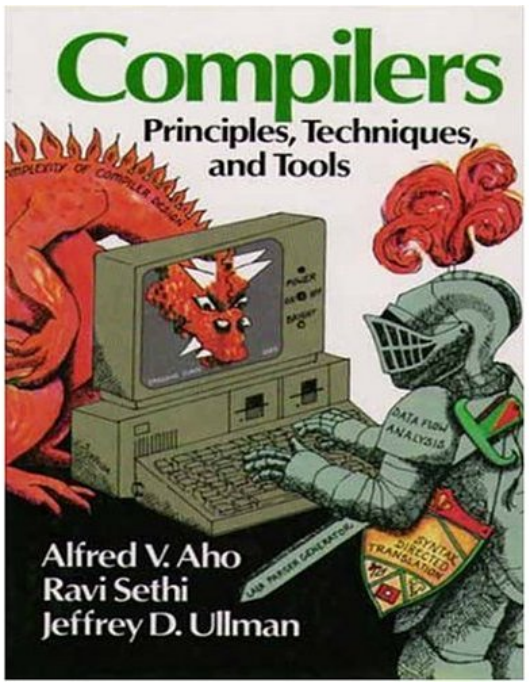

```
\chords { c2 c f2 c }
\new Staff \relative c' { \time 2/4 c4 c g'4 g a4 a g2 }
\new Lyrics \lyricmode { twin4 kle twin kle lit tle star2 }
```



Compiler Quality

Quality criteria ?

- correctness
- efficiency of the generated code
- its own efficiency



"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."

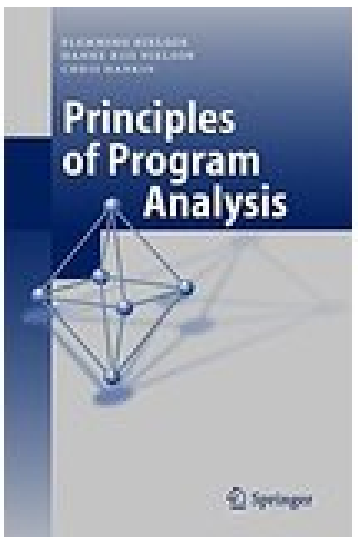
(Dragon Book, 2006)

Program Analysis

To prove :

- Correctness of compilers/optimisations phases.
- Correctnes of programs : invariants

. . . the second part of the course.



Course Objective

Be familiar with the mecanisms inside a (simple) compiler. Be familiar with basis of program analysis.

- And understand the links between them !

Course Content - Compilation Part

- Syntax Analysis : lexing, parsing, AST, types.
- Evaluators.
- Code generation.
- Code Optimisation.

Lab : a complete compiler for the TARGET18 architecture !

Support language : Python 3

Frontend infrastructure : ANTLR 4.

Course Content - Analysis Part

- Concrete semantics
- Abstract Interpretation
- A bit of verification : abstract interpretation, Hoare logic, ...

Labs : abstract interpretation. Support language : Python / Ocaml

Course Organization

- 13 + 1 course slots : Laure Gonnord.
- 14 lab slots : Matthieu Moy and Remy Grunblatt.

The official URL :

`http://laure.gonnord.org/pro/teaching/capM1.html` Github

for labs : `https://github.com/lauregonnord/cap-labs18`

Evaluation

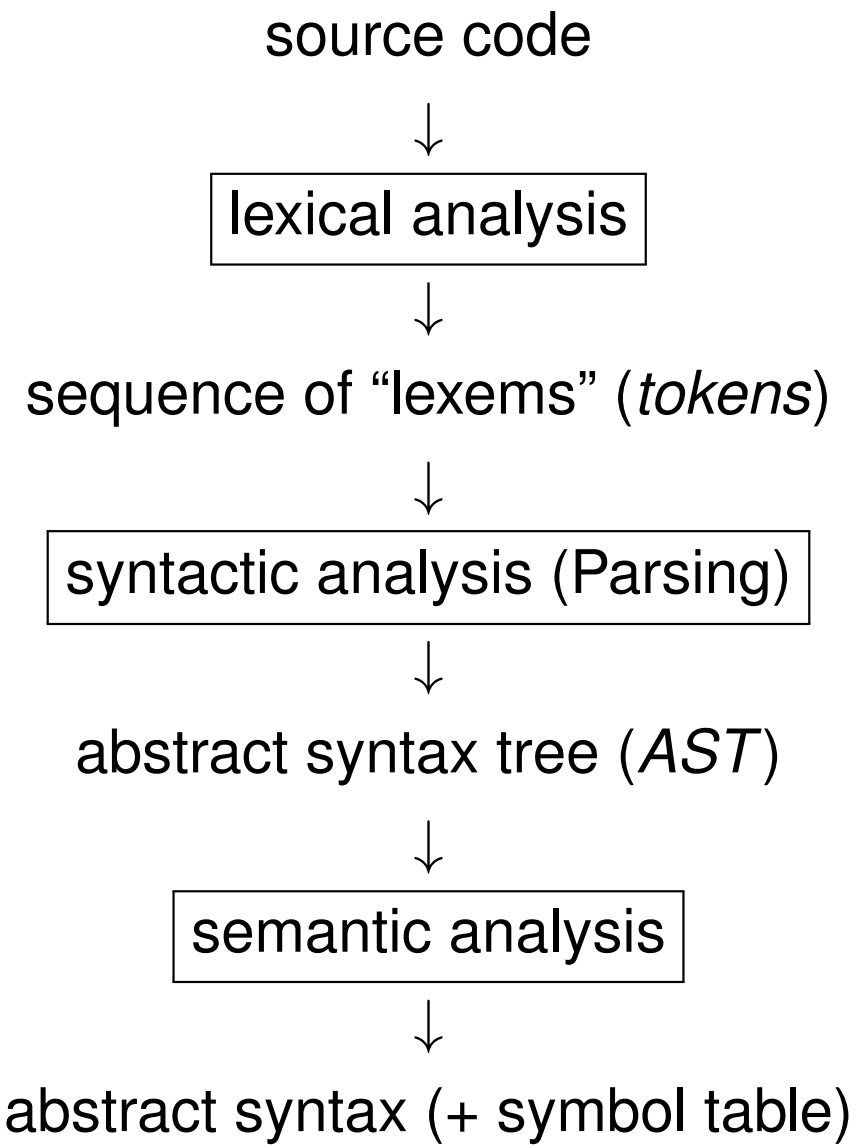
- One partial exam.
- Labs.
- A final exam.

Compiler phases

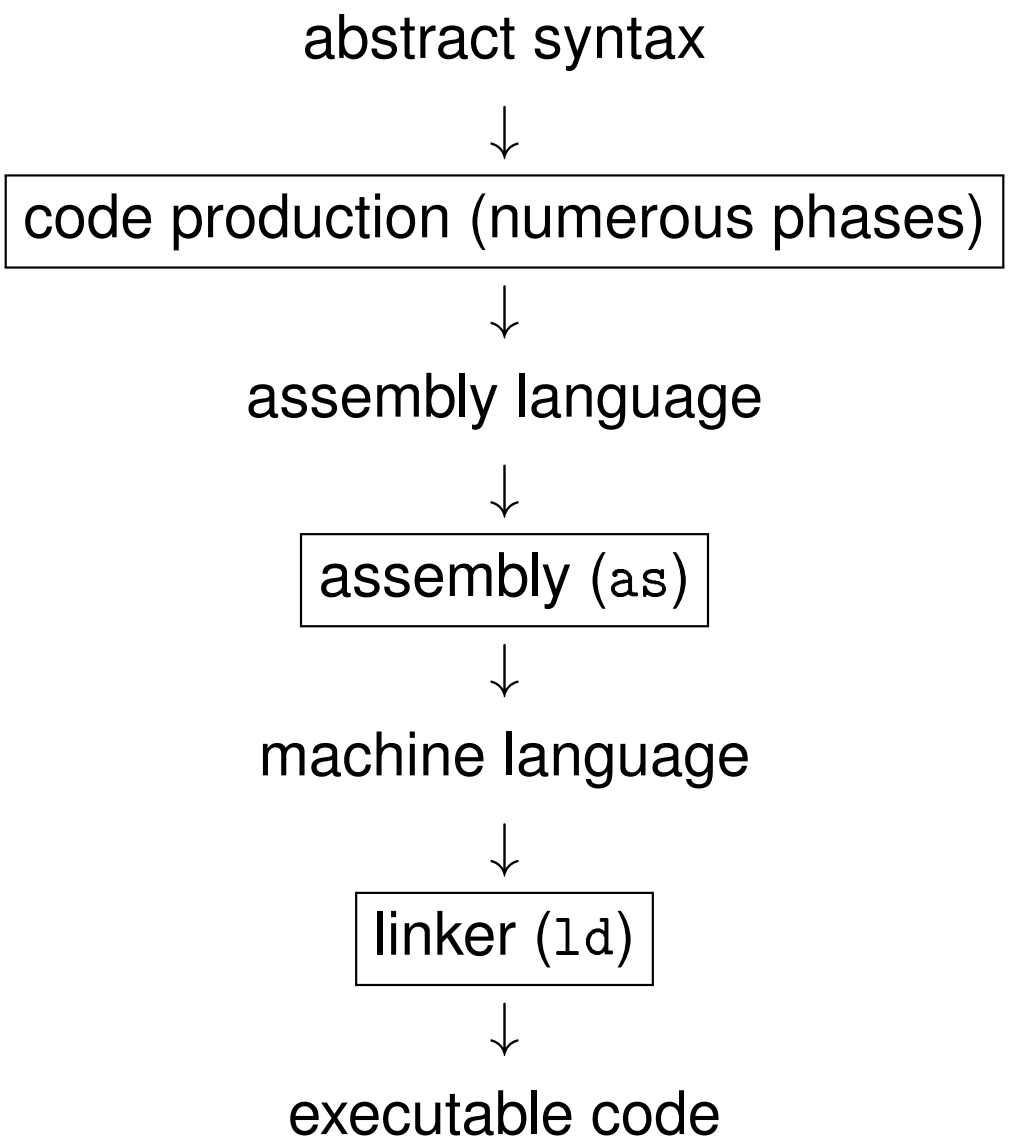
Usually, we distinguish two parts in the design of a compiler :

- an *analysis phase* :
  - recognizes the program to translate and its meaning.
  - launch errors (syntax, scope, types . . .)
- Then a *synthesis phase* :
  - produces a target file.
  - sometimes optmises.

Analysis Phase



Synthesis Phase



Today

assembly

Our target machine : TARGET18

- Memory : adressed by bit.
- Instructions do not have a constant size.
- Registers **PC**, **IR**, **SP**. 8 general purpose registers **R0,...,R7**, 2 special registers to access memory **A0, A1**.

We have a assembler and a simulator (see Lab 1)

TARGET18 ISA

See companion document.

Example : ADD instructions (subset)

- `add3 dr sr1 sr2`, does `dr <- sr1 + sr2`.
  - ↪ All operands are registers.
  - ↪ Example : `add3 r1 r2 r3` executes `r1 <- r2 + r3`.

- `add3 dr sr1 imm`, does `dr <- sr + imm`.
  - ↪ The last operand is an immediate value (on xxbits) encoded in the instruction.
  - ↪ Example : `add3i r1 r2 5` executes `r1 <- r2 + 5`.

TARGET18 ADD3/ADD3i : encoding

A bit specifies the addressing mode :

class	action	encoding
add3 reg reg reg	$r_d \leftarrow r_i + r_j$	`1110010|<rd(3bits)>|<ri(3bits)>|<rj(3bits)>`
add3i reg reg cte	$r_d \leftarrow r_i + \text{zeroext}(j)$	`1110011|<rd(3bits)>|<ri(3bits)>|<cte>`

The size and the encoding of the constant depends on its value :

0 + 1 bit	constant 0 ou 1
10 + 8 bits	byte
110 + 32 bits	
111 + 64 bits	

Example : assemble `add3i r1 r2 5`

TARGET18 : branching

Unconditional branching :

- `jump c`, does `PC <- a + c` where a is the current instruction address.

class	action	encoding
jump	jump to relative address	`1010<addr>`

Test and branch :

- `jumpif cond add`, same if the current flag is true

class	action	encoding
jump if cond	(depends on flag)	`1011<cond><addr>`

- see the encoding of `addr` in the TARGET18 ISA !

TARGET18 Branching 2/2 : Test and jump

Common usage of the “jumpif” instruction, example

```
cmp r1 0
jumpif eq endloop
```

- The label `endloop` is assembled into the adequate offset of the jump.
- See the list of operators in the companion sheet.

TARGET18 Memory access instructions

In the TARGET18 machine, stores and loads use special registers `a0` and `a1` have side-effects.

- read in memory N bits from address stored in `a0` : `readse`.
- write in memory N bits from address stored in `a0` : `write`.

▶ **indirect addressing** : the address must be stored in a register `a0` or `a1`.

class	action	encoding
write	$\texttt{Mem}[ctr] \leftarrow r_i(Nbits)$	`110100<ctr><N><ri>`
readse	$r_d \leftarrow \texttt{mem}[ctr](Nbits)$	`10011<ctr><N><rd>`

See the ISA for more info.

Memory accesses : 2/2

Classical operations :

- Emulation of load operation : $r_2 \leftarrow Mem[r_1]$:

```
setctr  a0 r1
readse a0 xxx r2 ; xxx the number of bits to read
; be careful a0 has been incremented
```

- Write in memory : $mem[r1] \leftarrow r2$:

```
setctr  a0 r1
write a0 xxx r2 ; xxx the number of bits to write
; be careful a0 has been incremented
```

- Useful instruction : `lea` “load effective address”, see doc.

Ex : Assembly code - demo

```
;;; simple TARGET18 assembly demo
;../TP2018/target18/asm2017/asm.py demo18.s -b
;../TP2018/target18/asm2017/emu/emu demo18.bin

        leti  r1  5             ; first op : cte
        lea r3  b               ; second, from memory
        setctr  a0 r3
        readse a0 8 r2
        add3 r4 r1 r2           ; add
        print  signed r4        ; ok this is 42
        getctr sp r3
        setctr  a0 r3           ; now store in memory (sp)
        write a0 8 r4
        getctr   a0 r3          ; get the result back.
        sub2i r3 8
        setctr  a0 r3
        readse a0 8 r5

        print  signed r5        ; and print.
loop:
        jump loop               ; infinite loop
b:
        .const 8 #00100101      ; 2's complement of 37!
```

TARGET18 Exercises

see TD sheet.