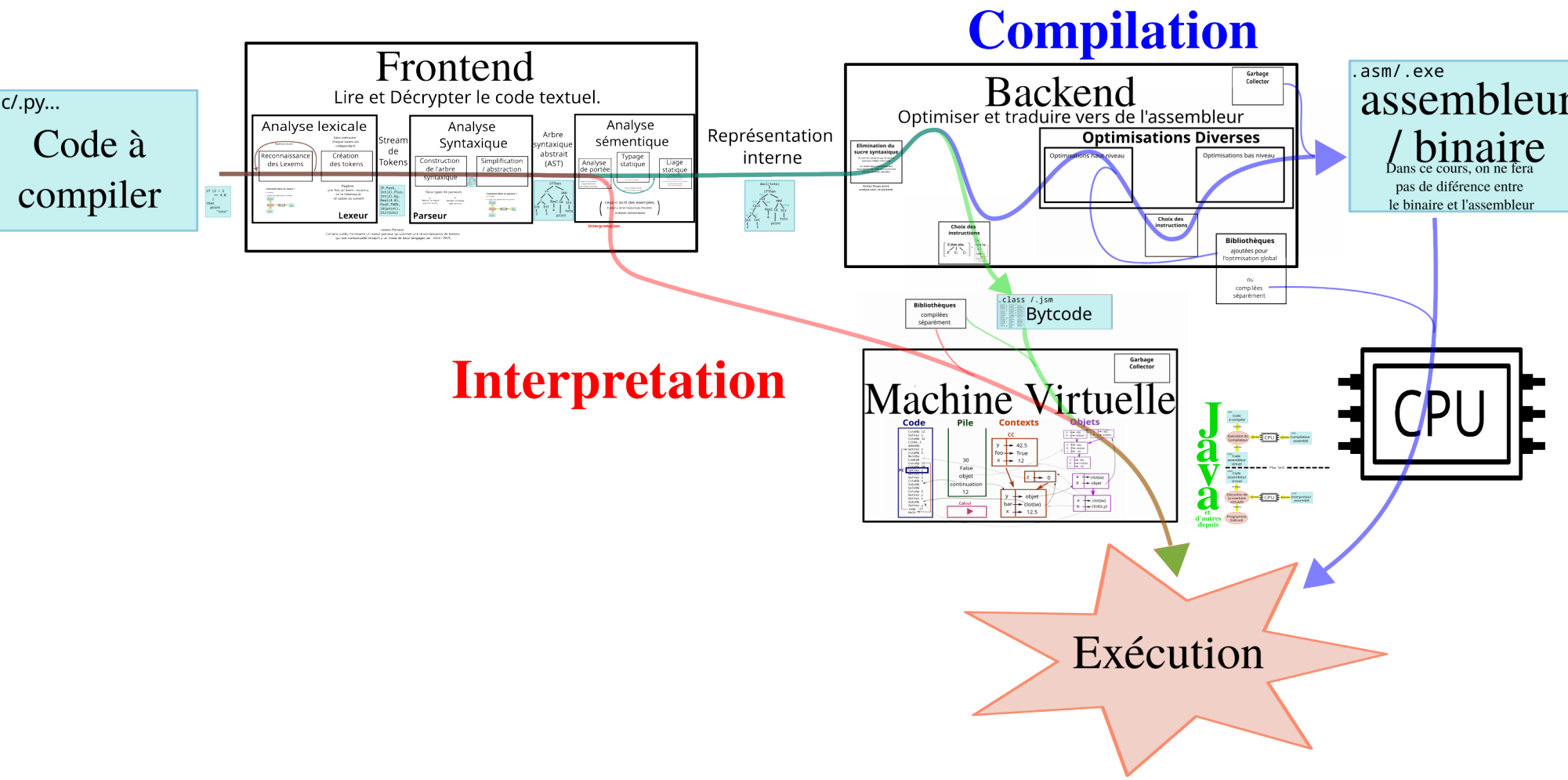
Compilation, 6th course : Semantic Analysis, Backend and VM

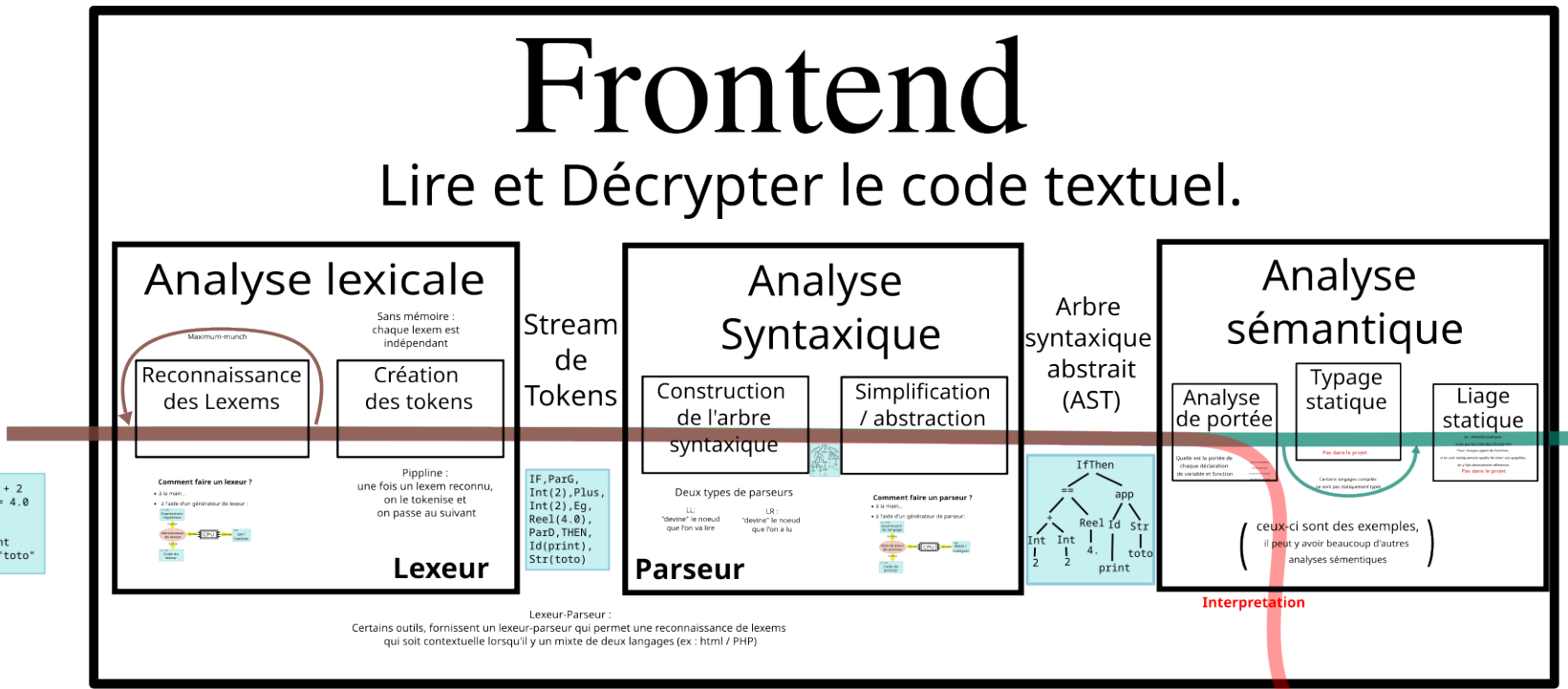
Flavien BREUVART

February 23, 2026

Semantic analyses

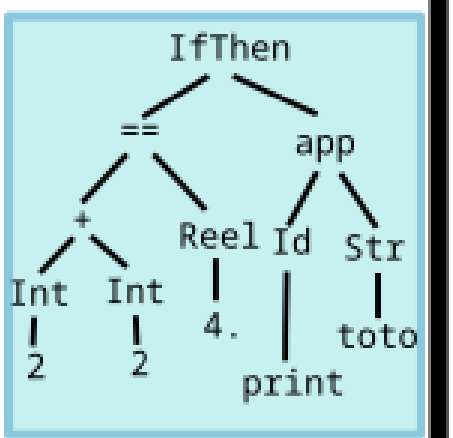


Semantic analyses



Semantic analyses

Arbre syntaxique abstrait (AST)



Analyse sémantique

Analyse de portée

Quelle est la portée de chaque déclaration de variable et fonction

Typage statique

Pas dans le projet

Certains langages compilés ne sont pas statiquement typés

Liage statique

Pour chaque appel de fonction, si on sait statiquement quelle fonction est appelée, on y'a directement référence.

Pas dans le projet

(ceux-ci sont des exemples, il peut y avoir beaucoup d'autres analyses sémentiques)

Interpretation

R

Principles of Semantic analyses

Semantic analysis is multiple

This is a sequence of more or less independent analyses.

Input and output are both ASTs

The phase analyses, rearranges, modifies and enriches the AST.

Modification

- Removal of syntactical sugar,
- Scope analysis,
- ...

Enrichment

- Static Type inference,
- Static functional linking,
- ...

Diversity of the analyses

The order and choices of analyses is different in each compiler,
some analysis require or would break others making this choice subtle.

Removing syntactical sugar

Syntactical sugar

Operators or code structures that are redundant.

A solution for a common dilemma

The programmer needs
a rich language, easy to read, write and remember

The compiler needs
a minimalist language, for concise and efficient analyses

Example : java "λ-expressions" are anonymous classes

if `expr` is of type `B`, then **(A a $\Rightarrow$ expr)** means :

```
new Function<A,B>(){
  public B apply (A a) {return expr;}
}
```

(Remark: in java, type inference has to be performed before in order to find B)

Syntactical sugar in JS

Some operators (++...)

Syntactical sugar in JS

Some operators (++...)

Syntactical sugar in JS

Some operators (++...)

$++x \quad \simeq$ $+e \quad \simeq$

Syntactical sugar in JS

Some operators (++...)

$$\texttt{++}x \quad \simeq \quad x = x\texttt{+1} \qquad\qquad \texttt{+}e \quad \simeq \quad \texttt{Number}(e)$$

Syntactical sugar in JS

Some operators (++...)

$$\texttt{++}x \quad \simeq \quad x = x\texttt{+1} \qquad\qquad \texttt{+}e \quad \simeq \quad \texttt{Number}(e)$$

Some commands (for, foreach, ...)

$$\texttt{var } x = e; \quad \simeq$$

$$\texttt{function } f(x)\,\{p\} \quad \simeq$$

Syntactical sugar in JS

Some operators (++...)

$$\texttt{++}x \quad \simeq \quad x = x\texttt{+1} \qquad\qquad \texttt{+}e \quad \simeq \quad \texttt{Number}(e)$$

Some commands (for, foreach, ...)

$$\texttt{var } x = e; \quad \simeq \quad \{\texttt{var } x;\ x = e;\}$$

$$\texttt{function } f(x)\,\{p\} \quad \simeq \quad \{\texttt{var } f;\ f = \texttt{function } (x)\,\{p\};\}$$

Syntactical sugar in JS

Some operators (++...)

$$\texttt{++}x \quad \simeq \quad x = x\texttt{+1} \qquad\qquad \texttt{+}e \quad \simeq \quad \texttt{Number}(e)$$

Some commands (for, foreach, ...)

$$\texttt{var } x = e; \quad \simeq \quad \{\texttt{var } x;\ x = e;\}$$

$$\texttt{function } f(x)\,\{p\} \quad \simeq \quad \{\texttt{var } f;\ f = \texttt{function } (x)\,\{p\};\}$$

$$\texttt{for}(e_1; e_2; e_3)\ \ \texttt{c}$$

Syntactical sugar in JS

Some operators (++...)

$$\texttt{++}x \quad \simeq \quad x = x\texttt{+1} \qquad\qquad \texttt{+}e \quad \simeq \quad \texttt{Number}(e)$$

Some commands (for, foreach, ...)

$$\begin{aligned}
\texttt{var } x = e; \quad &\simeq \quad \{\texttt{var } x;\ x = e;\} \\
\texttt{function } f(x)\,\{p\} \quad &\simeq \quad \{\texttt{var } f;\ f = \texttt{function } (x)\,\{p\};\} \\
\texttt{for}(e_1; e_2; e_3)\ \texttt{c} \quad &\simeq \quad \{e_1;\ \texttt{while}(e_2)\ \{\,\texttt{c}\ e_3;\,\}\}
\end{aligned}$$

Syntactical sugar in JS

Some operators (++...)

$$\texttt{++}x \quad \simeq \quad x = x\texttt{+1} \qquad\qquad \texttt{+}e \quad \simeq \quad \texttt{Number}(e)$$

Some commands (for, foreach, ...)

$$\begin{aligned}
\texttt{var } x = e; \quad &\simeq \quad \{\texttt{var } x;\ x = e;\} \\
\texttt{function } f(x)\,\{p\} \quad &\simeq \quad \{\texttt{var } f;\ f = \texttt{function } (x)\,\{p\};\} \\
\texttt{for}(e_1; e_2; e_3)\ \texttt{c} \quad &\simeq \quad \{e_1;\ \texttt{while}(e_2)\ \{\,\texttt{c}\ e_3;\,\}\}
\end{aligned}$$

```
for(let x of t) c
```

Syntactical sugar in JS

Some operators (++...)

$$\texttt{++}x \quad \simeq \quad x = x\texttt{+1} \qquad\qquad \texttt{+}e \quad \simeq \quad \texttt{Number}(e)$$

Some commands (for, foreach, ...)

$$\texttt{var } x = e; \quad \simeq \quad \{\texttt{var } x;\ x = e;\}$$

$$\texttt{function } f(x)\{p\} \quad \simeq \quad \{\texttt{var } f;\ f = \texttt{function } (x)\{p\};\}$$

$$\texttt{for}(e_1; e_2; e_3)\ \texttt{c} \quad \simeq \quad \{e_1;\ \texttt{while}(e_2)\ \{\ \texttt{c}\ e_3;\ \}\}$$

$$\texttt{for(let x of t) c} \simeq \left\{\begin{array}{l}\texttt{let \_iter = t[Symbol.iterator]();}\\ \texttt{let \_x = \_iter.next();}\\ \texttt{while(!\_x.done)}\\ \quad \{\texttt{x=\_x.value; c \_x=\_iter.next();}\}\end{array}\right\}$$

Syntactical sugar in JS

Some operators (++...)

$$\texttt{++}x \quad \simeq \quad x = x\texttt{+1} \qquad\qquad \texttt{+}e \quad \simeq \quad \texttt{Number}(e)$$

Some commands (for, foreach, ...)

$$\texttt{var } x = e; \quad \simeq \quad \{\texttt{var } x;\ x = e;\}$$

$$\texttt{function } f(x)\{p\} \quad \simeq \quad \{\texttt{var } f;\ f = \texttt{function } (x)\{p\};\}$$

$$\texttt{for}(e_1; e_2; e_3)\ \texttt{c} \quad \simeq \quad \{e_1;\ \texttt{while}(e_2)\ \{\texttt{c}\ e_3;\}\}$$

$$\texttt{for(let x of t) c} \simeq \left\{\begin{array}{l}\texttt{let \_iter = t[Symbol.iterator]();}\\ \texttt{let \_x = \_iter.next();}\\ \texttt{while(!\_x.done)}\\ \quad \texttt{\{ x=\_x.value; c \_x=\_iter.next();\}}\end{array}\right\}$$

(real code a bit more complex)

(obsolete) Classes used to be syntactical sugar

Not anymore, but legacy classes were identified with their constructors.

Scope Analysis and hoisting

Binding variable to their binder

An identifier (variable or function) have to be defined somewhere, but can be redefined...
We want to raise error is used without definition and guaranty that the correct definition is used !

Ideally : the binder is the first found ancestor ($\simeq \lambda$-calculus)

Rising up in the tree until the declaration of the identifier is found.

Hoisting : In some language (like JS) identifiers can be declared later

One have to hoist the declarations up to there point of application.

Scope analysis in JS : the `let x` declaration

The scope analysis verify that in the block where `let x` appear

- no other `x` is defined (with `let` or `var`)
- `x` is not used before being defined.

What is a block

It is a sequence of commands, generally delimited by '{' and '}'.

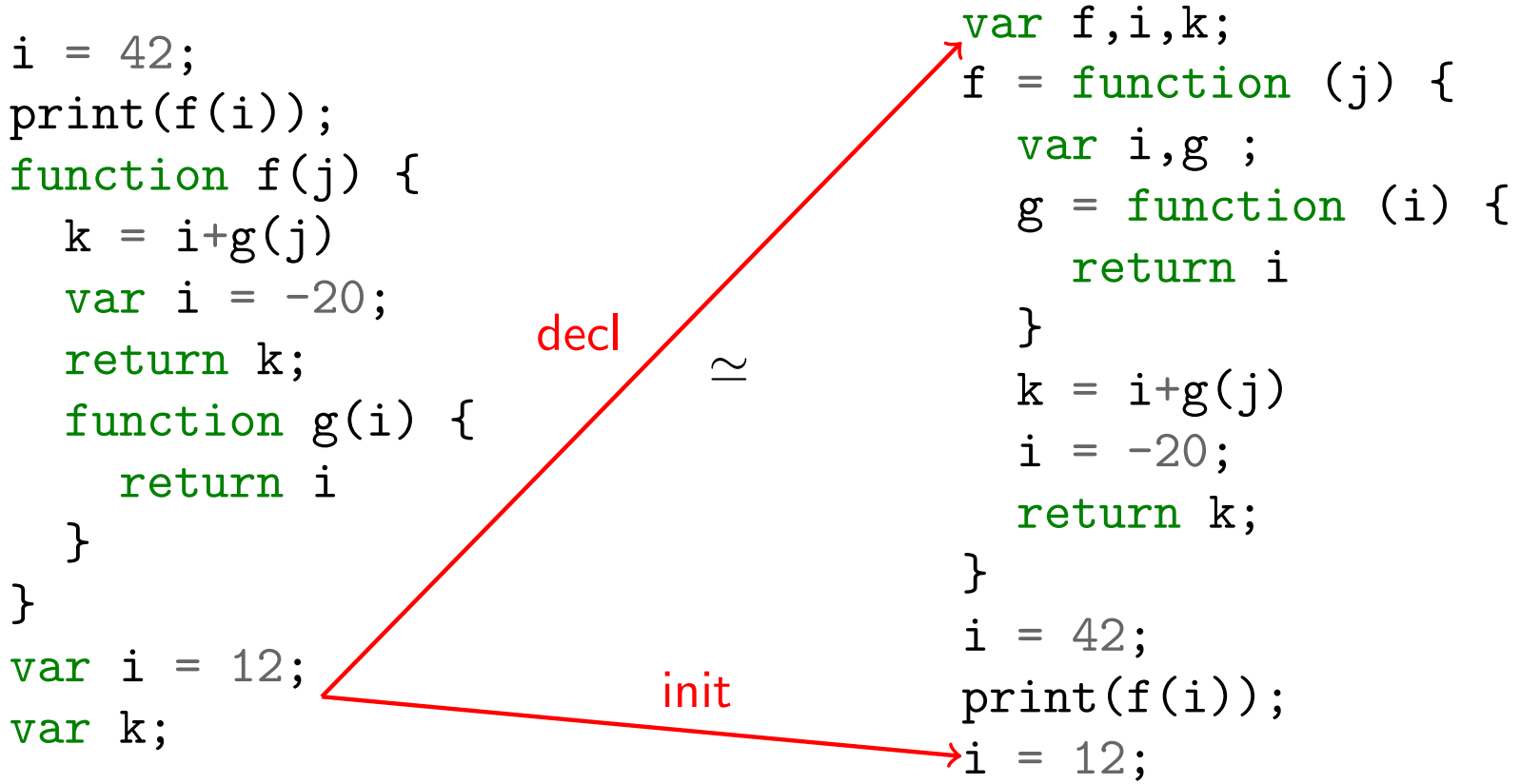
Examples of hoisting in JS : Variables can be declared later on !

```
i = 42;
print(f(i));
function f(j) {
  k = i+g(j)
  var i = -20;
  return k;
  function g(i) {
    return i
  }
}
var i = 12;
var k;
```

$\simeq$

```
var f,i,k;
f = function (j) {
  var i,g ;
  g = function (i) {
    return i
  }
  k = i+g(j)
  i = -20;
  return k;
}
i = 42;
print(f(i));
i = 12;
```

Examples of hoisting in JS : Variables can be declared later on !



var declaration :

The declaration is hoisted, but not the initialisation.

Examples of hoisting in JS : Variables can be declared later on !

```
i = 42;
print(f(i));
function f(j) {
  k = i+g(j)
  var i = -20;
  return k;
  function g(i) {
    return i
  }
}
var i = 12;
var k;
```

decl

init

$\simeq$

```
var f,i,k;
f = function (j) {
  var i,g ;
  g = function (i) {
    return i
  }
  k = i+g(j)
  i = -20;
  return k;
}
i = 42;
print(f(i));
i = 12;
```

`function` declaration :

Both the declaration and the initialisation are hoisted.

Examples of hoisting in JS : Variables can be declared later on !

```
i = 42;
print(f(i));
function f(j) {
  k = i+g(j)
  var i = -20;
  return k;
  function g(i) {
    return i
  }
}
var i = 12;
var k;
```

decl

$\simeq$

```
var f,i,k;
f = function (j) {
  var i,g ;
  g = function (i) {
    return i
  }
  k = i+g(j)
  i = -20;
  return k;
}
i = 42;
print(f(i));
i = 12;
```

Warning : hoisting do not cross functions

variables and functions are hoisted to the first function def.

Interlude : declaration of JS variable

Declared variables

Declaration is hoisted to the heading of the current function.

(deprecated) Non declared variables

Considered as declared at the beginning of the program.

(The project's virtual machine does that for you)

And functions ?

Same as variables, but
their "value" is also hoisted.

And classes ?

Those are not hoisted

Static Typing

Getting types at compile time

Different algorithms :

- Limited inference (C, Java old school) :
  the user explicits every type (variables and functions).
- Strong inference (OCaml, Scala) :
  the compiler "guesses" the types

Not in our project

Because JS's types are dynamic :
types are decided during the evaluation
Slower, unsafe, but "simpler".

Typescript = JS + static types

JITs can fasten dynamic types, but can't secure them,
thus the creation of Typescript

Static Linking of Function

Definition : Static Linking

On function application :

- statically determine which function is used,
- add, in the application node of the AST, a pointer toward this function

(the AST is a tree no more)

Attention : not always available (ex : object's methods, functional argument)

Allow many optimisations

- immediate jumps on the correct line,
- easier context switch,
- no closure creation,
- potential inclining (replacing function by corresponding code),

Project : No static linking in JS.

Static linking in Java (not JS !)

Static methods ⇒ Static linking

Do not depends on the instance, only the type signature.

Non static method ⇒ Dynamic linking

If the types is inherited, another function could be called.

Java has mixed linking by overloading methods

Signature do not determine the called function but can restrict them, creating surprising behaviours.

Example of linking in Java

```
class A {
  public String f (A obj) {return("A/A");}
}
class B extends A {
  public String f (B obj) {return("B/B");}
  public String f (A obj) {return("B/A");}
}
....
    A aa = new A ();  A ab = new B ();  B b  = new B ();
....
    System.out.println (aa.f (ab ));  // "A/A"
    System.out.println (aa.f (b  ));  // "A/A"
    System.out.println (ab.f (ab ));  // "B/A"
    System.out.println (ab.f (b  ));  // "B/A"
    System.out.println ( b.f (ab ));  // "B/A"
    System.out.println ( b.f (b  ));  // "B/B"
```

Example of linking in Java

```
class A {
  public String f (A obj) {return("A/A");}
}
class B extends A {
  public String f (B obj) {return("B/B");}
  public String f (A obj) {return("B/A");}
}
....
    A aa = new A ();  A ab = new B ();  B b  = new B ();
....
    System.out.println (aa^A.f (ab^A));  // "A/A"
    System.out.println (aa^A.f (b^B));  // "A/A"
    System.out.println (ab^A.f (ab^A));  // "B/A"
    System.out.println (ab^A.f (b^B));  // "B/A"
    System.out.println ( b^B.f (ab^A));  // "B/A"
    System.out.println ( b^B.f (b^B));  // "B/B"
```

Compile time : Type inference

Example of linking in Java

```
class A {
  public String f^A (A obj) {return("A/A");}
}
class B extends A {
  public String f^B (B obj) {return("B/B");}
  public String f^A (A obj) {return("B/A");}
}
....
    A aa = new A ();  A ab = new B ();  B b  = new B ();
....
    System.out.println (aa^A.f^A (ab^A ));  // "A/A"
    System.out.println (aa^A.f^A (b ^B ));  // "A/A"
    System.out.println (ab^A.f^A (ab^A ));  // "B/A"
    System.out.println (ab^A.f^A (b ^B ));  // "B/A"
    System.out.println ( b^B.f^B (ab^A ));  // "B/A"
    System.out.println ( b^B.f^B (b ^B ));  // "B/B"
```

Compile time : Static linking

Example of linking in Java

```
class A {
  public String f^A (A obj) {return("A/A");}
}
class B extends A {
  public String f^B (B obj) {return("B/B");}
  public String f^A (A obj) {return("B/A");}
}
....
    A aa = new A ();  A ab = new B ();  B b  = new B ();
....
    System.out.println (aa^A.f^A (ab^A_B));  // "A/A"
    System.out.println (aa^A.f^A (b^B));  // "A/A"
    System.out.println (ab^A_B.f^A (ab^A_B));  // "B/A"
    System.out.println (ab^A_B.f^A (b^B));  // "B/A"
    System.out.println ( b^B.f^B (ab^A_B));  // "B/A"
    System.out.println ( b^B.f^B (b^B));  // "B/B"
```

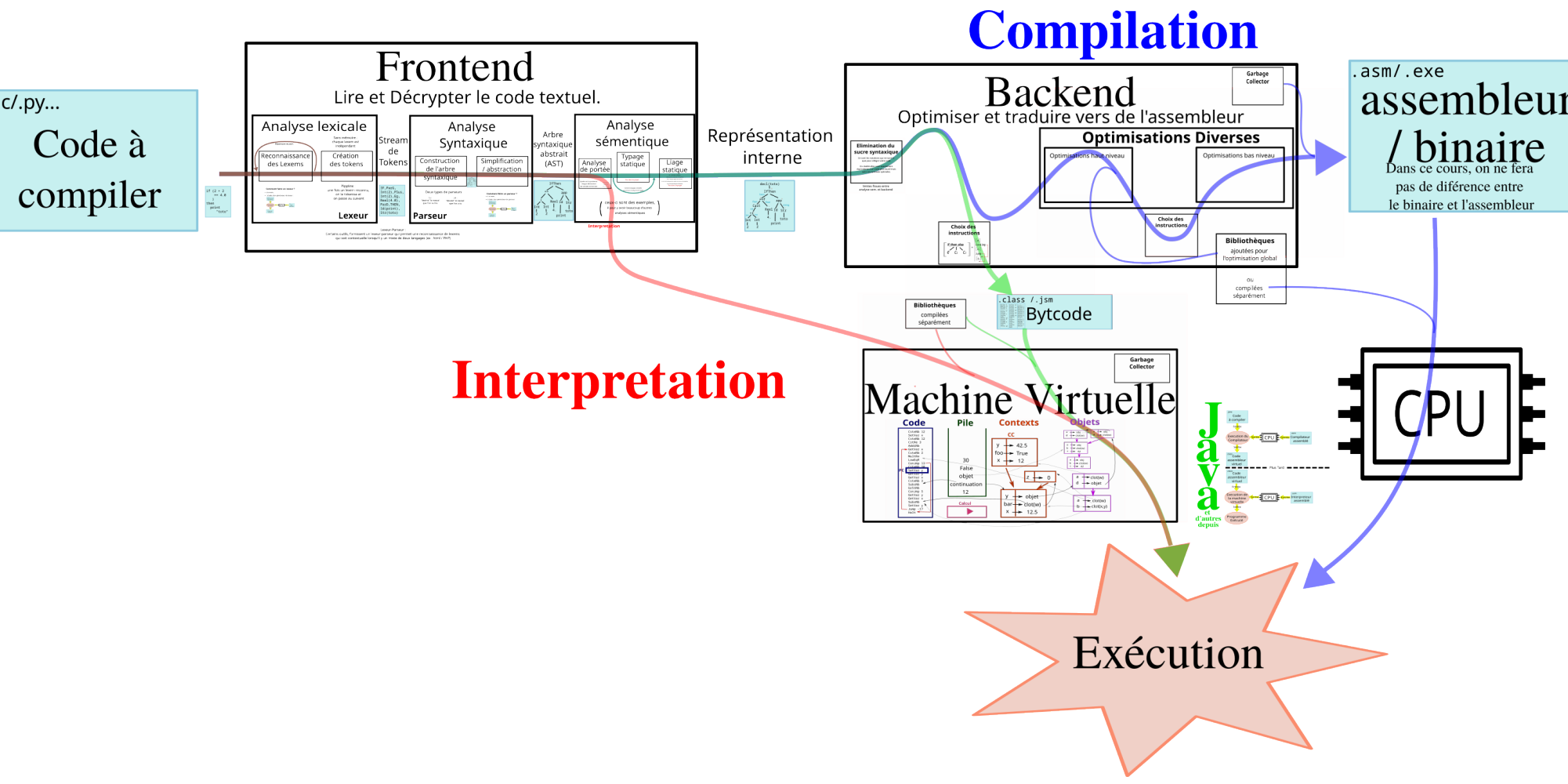
Execution time : Class inspection

Example of linking in Java

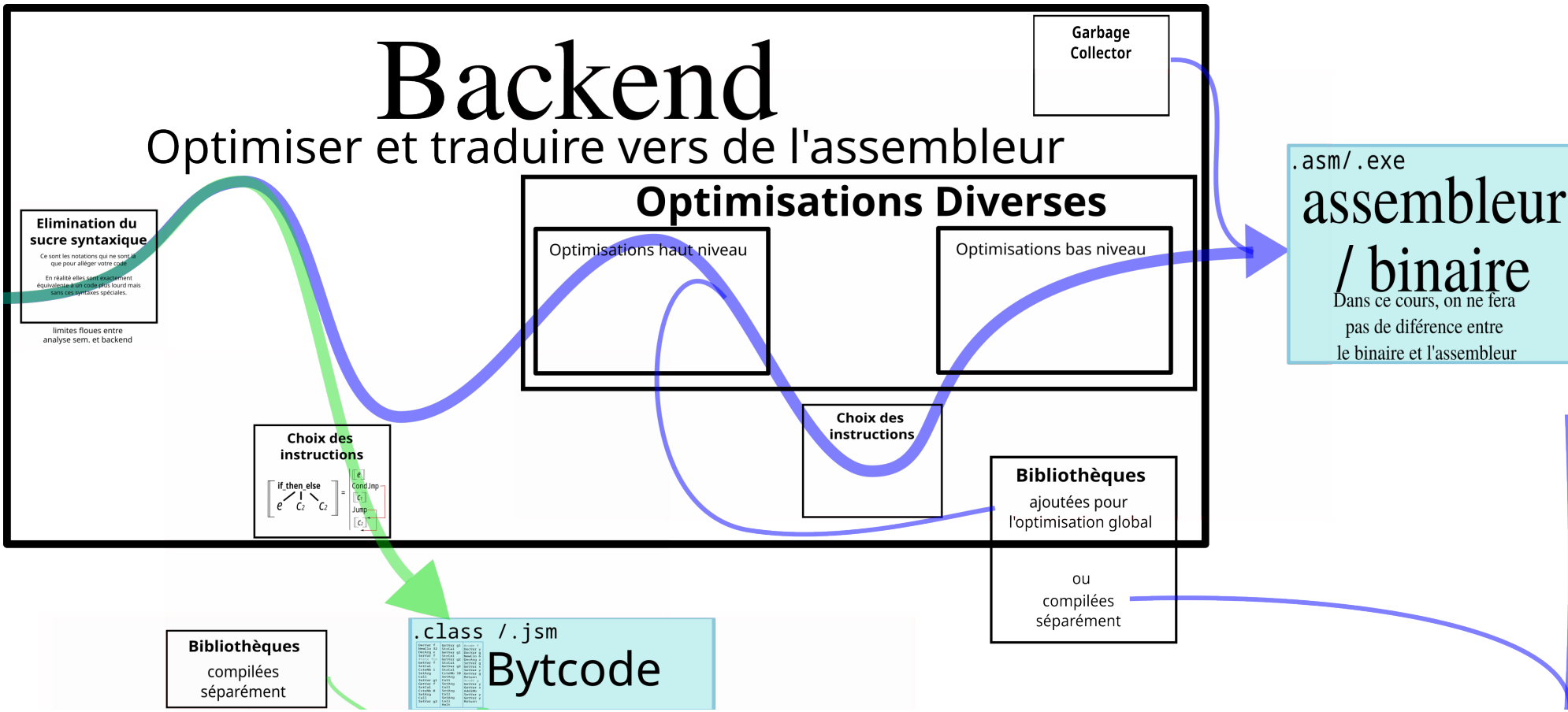
```
class A {
  public String f^A_A (A obj) {return("A/A");}
}
class B extends A {
  public String f^B_B (B obj) {return("B/B");}
  public String f^A_B (A obj) {return("B/A");}
}
....
    A aa = new A ();  A ab = new B ();  B b  = new B ();
....
    System.out.println (aa^A_A.f^A_A (ab^A_B));  // "A/A"
    System.out.println (aa^A_A.f^A_A (b^B));  // "A/A"
    System.out.println (ab^A_B.f^A_B (ab^A_B));  // "B/A"
    System.out.println (ab^A_B.f^A_B (b^B));  // "B/A"
    System.out.println ( b^B_B.f^B_B (ab^A_B));  // "B/A"
    System.out.println ( b^B_B.f^B_B (b^B));  // "B/B"
```

Execution time : Dynamic linking

Backend (briefly)



Backend (briefly)



In the project : Only the green arrow (downward)

Assembly/binary ?

Binary .exe

Machine language

1 memory block = 1 instruction (or meta-instr.)

0100111111111111

Assembly .asm, .S

Assembled
+ macros
+ labels
+ meta-instr.

loop : JSR loop

Compiler target : object code

In the project, we target assembly language (more readable)

For simplification we will call everything assembly...

(This is even more complex for GPU or embedded systems)

Assembly/binary ?

Binary .exe

Machine language

1 memory block
= 1 instruction
(or meta-instr.)

0100111111111111

Object .o,.obj

Binary
+ external link

0100111111111111

Assembled

literal translation of object code

JSR 0x-1

Assembly .asm, .S

Assembled
+ macros
+ labels
+ meta-instr.

loop : JSR loop

Compiler target : object code

In the project, we target assembly language (more readable)

For simplification we will call everything assembly...

(This is even more complex for GPU or embedded systems)

Assembly/binary ?

Binary .exe

Machine language

1 memory block = 1 instruction (or meta-instr.)

0100111111111111

micro-operations

Bin. instructions are often splet into more atomic ones inside the CPU

Object .o,.obj

Binary + external link

0100111111111111

Assembled

literal translation of object code

JSR 0x-1

Assembly .asm, .S

Assembled + macros + labels + meta-instr.

loop : JSR loop

Compiler target : object code

In the project, we target assembly language (more readable)

For simplification we will call everything assembly...

(This is even more complex for GPU or embedded systems)

Library linking

Issue : merging different code files

Library, project, C function, system call...

Three solutions :

Global compil.

- in the backend
- optimised

Separated compil.

- object→binary
- faster compil.

System calls

- during execution
- ...

In practice: a mix of those three.

Optimisations

Many passes of optimisation

Tens of different optimisations methods.
Some are performed several times.

gcc : around 100 different passes !

High-level

On the AST
global analysis,
functions/classes

gcc : 40+ passes

Call-graph

on the flow-chart
loops,
variable domains

gcc : 20+ passes

Low-level

on assembly
instruction change.
instr. order

gcc : 20+ passes

If you are interested you can look as the superb "cours confiné" of ENS Lyon :
`https://www.youtube.com/playlist?list=PLtjm-n_Ts-J-6EU1WfVIWLhl1BUUR-Sqm`

Optimisations in the course

In the project

Some example of small optimisations are to be implemented.

Lectures

We will give some slightly more complex examples here and there to elevate spirit.

Additional (bonus) lecture

I have one lecture (on blackboard only) that changes every year at the end of the course. It can be on some optimisation.

You are welcome if you have requests on this additional lecture.

Garbage Collector (GC)

En français : “ramasse miettes” ou “Glaneur de Cellules”

Objective

Automatically freeing unused memory

Independent thread

Mini-program executed on a different thread.

Method : freeing orphan nodes

i.e., non-accessible objects following pointers from current state (stack+cont.+register)

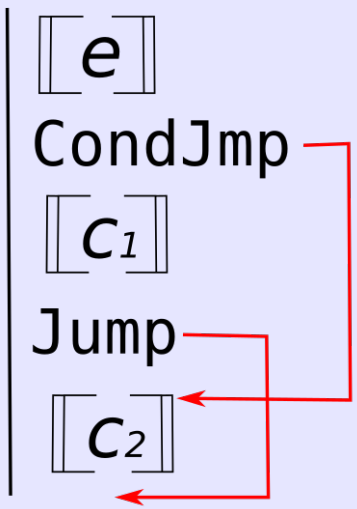
(Other unorthodox methods exist)

We will come back to it later on

Instruction choice : Translating into target language

Translates a node of the AST into an assembly structure

$$[\![\textbf{if (} e \textbf{) } c_1 \textbf{ else } c_2]\!] = \begin{array}{|l} [\![e]\!] \\ \texttt{CondJmp} \\ [\![c_1]\!] \\ \texttt{Jump} \\ [\![c_2]\!] \end{array}$$

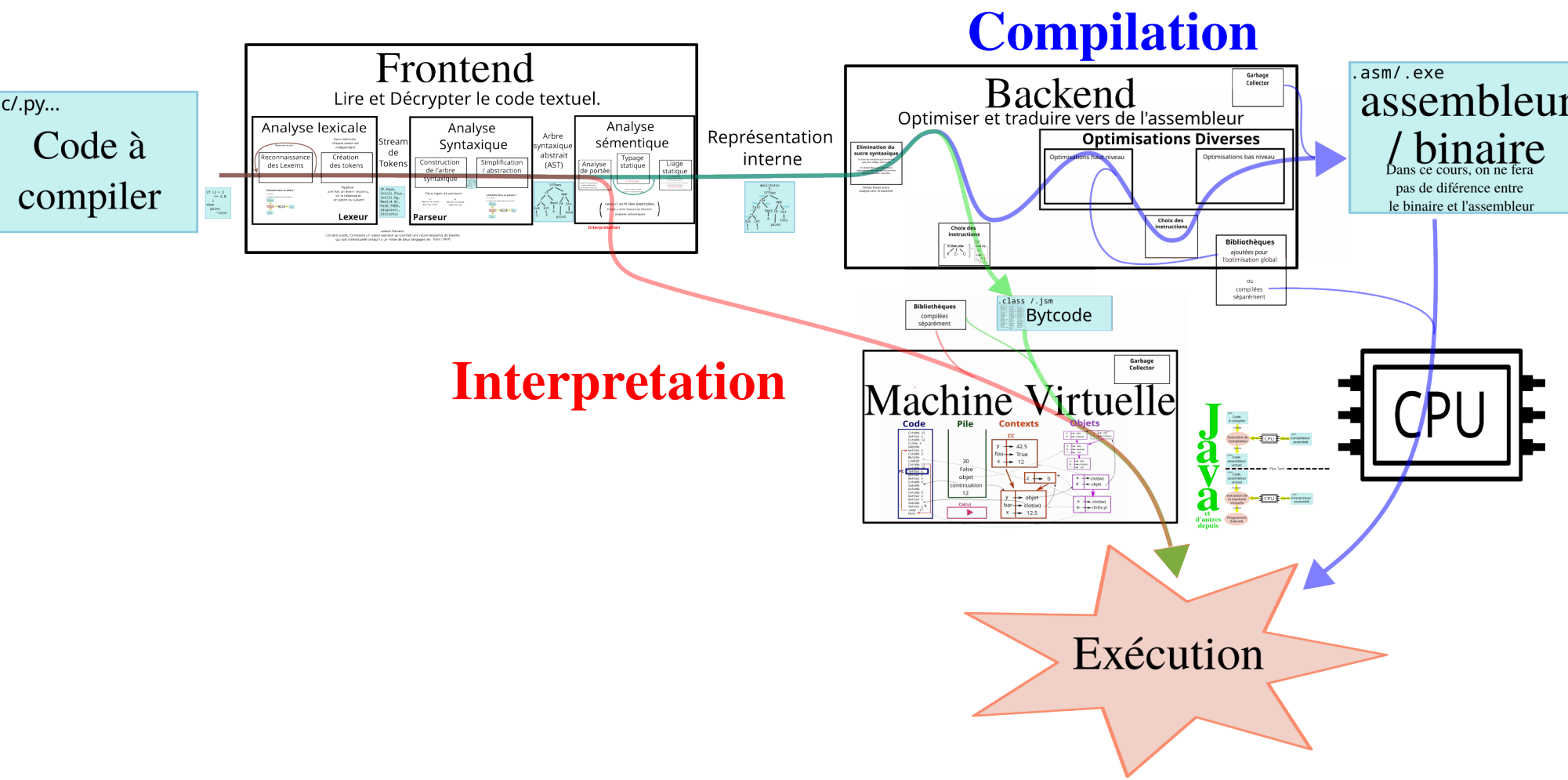


The choice can be subtle if with many available instructions

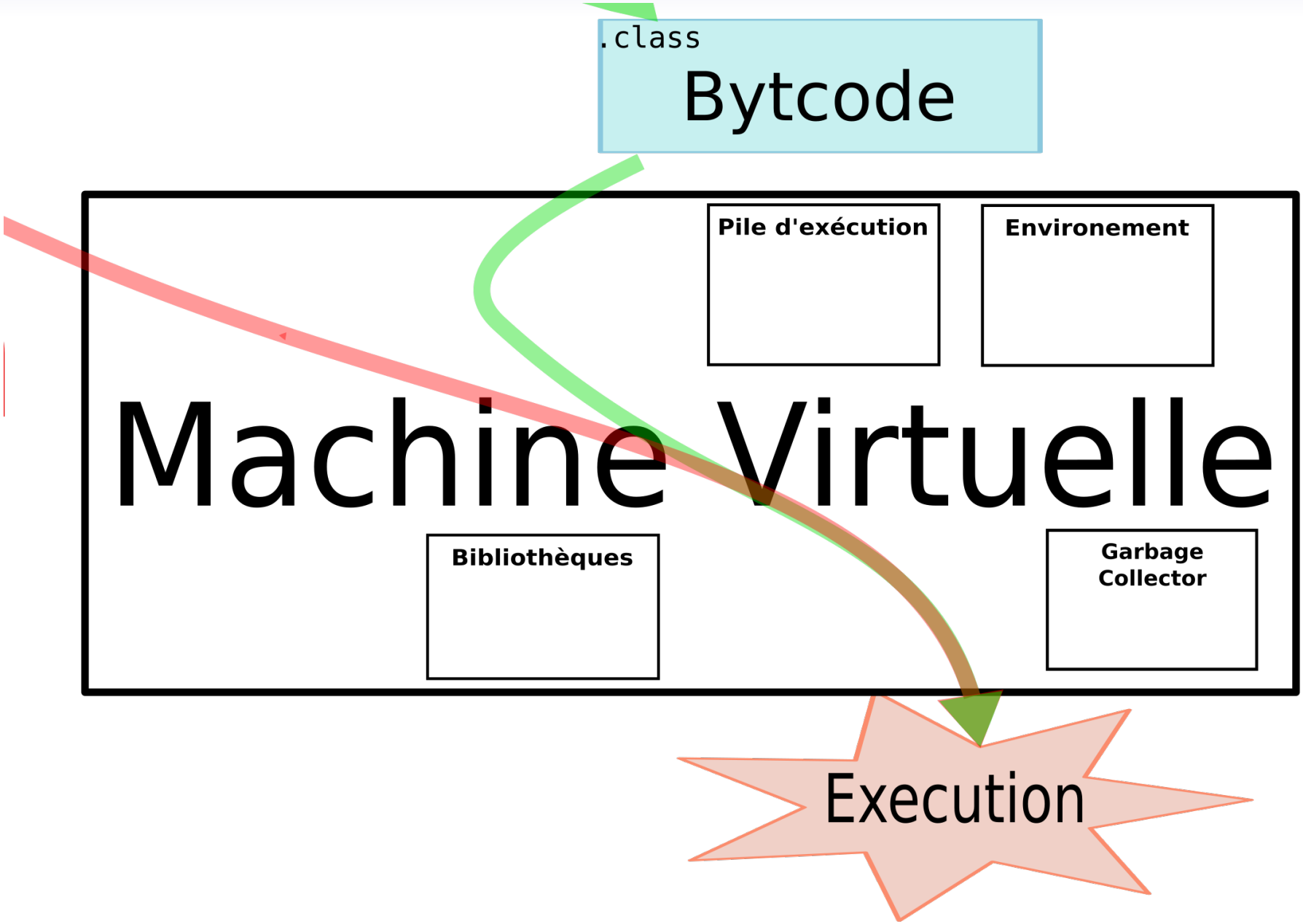
You can look at alternative instruction for our mini-JS-machine.

This is the part we will study during this semester.

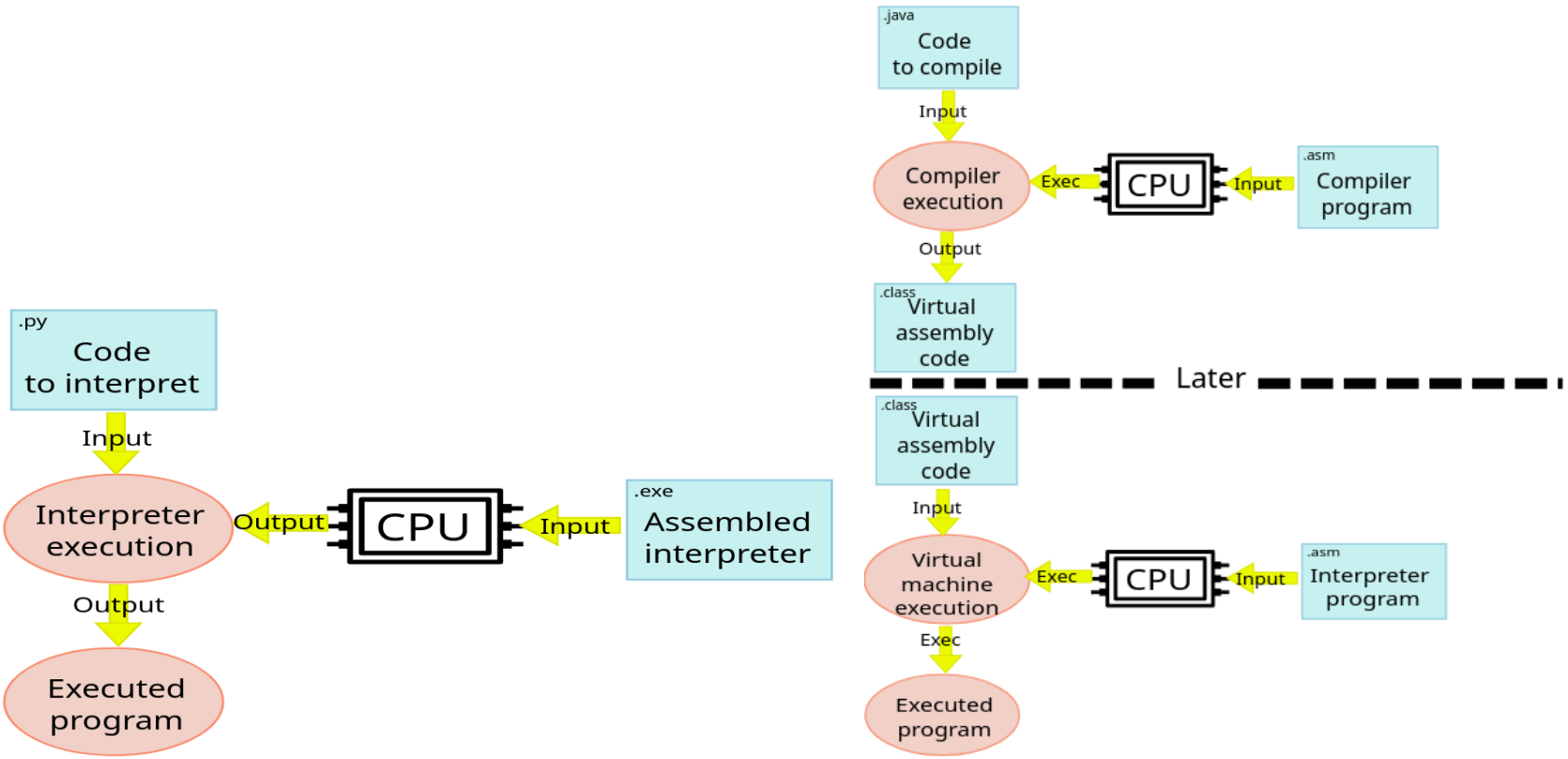
Virtual Machine



Virtual Machine



Goal of the VM



What is a "machine" in this course ?

Executes immediately a (parsed) program

Can also execute assembly or bytecode (parser trivial).

Some example of machines

Processors	on	machine bytecode
Java virtual machine	on	Java bytecode
JS virtual machine	on	parsed JS
Mini-JS Machine	on	homemade assembly
~~Linux virtual machine~~	on	???

machine : physical/virtual/abstract

Physical	Virtual	Abstract
Hardwear → CPU/GPU	Softwear →Java/JS/Pytho, VM	Not implemented →Turing/Krivine M.
Efficient	**Adapted**	**Theoretic**
but complex input: binary	and universal input:bin./assem./AST	semantic input: AST
Circuit structures	**Data structures**	**High level struct.**
registers, heap, addressing,caches..	stacks, contexts, objects, GC	AST infinite objects...
steps	**steps**	**steps**
constant time	near constant time	arbitrary

Abstraction Layers (informal)

...	...
mathematics	algo
formal logic	pseudo-code
language semantics	program
virtual/abstract machine	program/assembly/binairy
system	assembly
processor model	binary
processor	binary + caches + ...
sequential (clocked) circuits	electricity...
...	...

In this course : Mini-JS machine

Toy virtual machine used for the course

Input : mini-JS Assembly

Not very efficient,

pedagogical choices
(progressive difficulty)

I conceived and wrote it, bugs are probable

Mini-JS Assembly code

Assembly adapted to JS language :

- values are those of JS types,
- default behaviours compatible with JS,
- well chosen instructions.