

École Polytechnique
CSC_52064 : Compilation
examen 2026

Jean-Christophe Filiâtre

16 mars 2026 — 14h00–17h00

The test lasts 3 hours. Handwritten or printed course notes are the only documents allowed. Most questions are independent, in the sense that it is not necessary to have answered the previous questions in order to deal with a question. On the other hand, questions can call on definitions or results introduced in previous questions. Unless explicitly stated otherwise, all answers must be justified.

Feel free to answer in French or English.

Figures 1–4 are grouped together at the end of the subject on page 8.

Suggestion: detach the last two sheets.

In this subject, we consider a small imperative language called `GOO`, manipulating integers and structures allocated on the heap, whose abstract syntax is given in Figure 1. A program (p) is a list of type declarations of structures (D), mutually recursive, and a list of function definitions (F), also mutually recursive. A structure S is declared with the list of its fields, each field y being accompanied by its type τ . A type is either `int`, for an integer, or `*S` for a pointer to a structure S . Here is an example of a program that defines a type `L` of a singly linked list and a function `len` to calculate the length of a list:

```
struct L { head int, next *L }
struct R { r int }
func (l *L) len(out *R) {
  if l = nil then out.r = 0 else { l.next.len(out) out.r = out.r - -1 }
}
```

As we can see, the definition of a function and its call distinguish the first parameter, called the *receiver*, whose type must be a pointer. We note that the receiver *can be the null pointer*. The semantics is strict, with call by value. Functions do not return anything, but they do have effects, either through the `print` instruction or by modifying the content of a structure, as in the example above.

Question 1 Give the code for a function `func (l *L) nth(i int, out *R)` that puts the i -th element of the list `l` into `out.r`, with elements counted from 0. We assume $0 \leq i < n$ where n is the length of the list `l`.

Question 2 Give the code for a function `fib_below` that constructs and stores in `out.l` the list of Fibonacci numbers strictly less than `max`, in ascending order.

```
struct P { l *L }
func (out *P) fib_below(max int) { ... }
```

(Reminder: The Fibonacci sequence (F_n) is defined by $F_0 = 0$, $F_1 = 1$, and $F_{n+2} = F_n + F_{n+1}$ for $n \geq 0$.)

Semantics. We equip GOO with a big-step operational semantics. A **value** v is either an integer n , or the value `nil`, or an address ℓ . A **memory** M is a function that associates an address ℓ with the content of a structure, the latter itself being a function from fields to values. Thus, $M(\ell)(y)$ is the value of the field y of the structure located in memory at address ℓ , provided that $\ell \in \text{dom}(M)$ and $y \in \text{dom}(M(\ell))$. An **environment** E is a function from variables to values.

For expressions, we introduce a relation $M, E, e \rightarrow M', v$ meaning “in memory M and environment E , the expression e evaluates successfully to the value v , for a final memory M' .” (The presence of the `new` construction in expressions induces a possible modification of memory during the evaluation of an expression.) Figure 2 defines this relation through a set of inference rules. In these rules, the notation $M \oplus \{x \mapsto v\}$ denotes the function M' defined by:

$$\begin{cases} M'(x) = v, \\ M'(y) = M(y) \text{ for } y \in \text{dom}(M) \text{ and } y \neq x. \end{cases}$$

In particular, we have $\text{dom}(M') = \text{dom}(M) \cup \{x\}$.

For instructions, we introduce a relation $M, E, s \xrightarrow{t} M'$ meaning “in memory M and environment E , the instruction s evaluates successfully, for a final memory M' and a trace t .” A **trace** is the sequence of integers displayed by `print`, represented by a word over the alphabet \mathbb{Z} . We denote ε the empty word and $t_1 t_2$ the concatenation of the words t_1 and t_2 . For $n \in \mathbb{Z}$, the trace n is the word of length 1 reduced to n . Figure 3 defines the relation \xrightarrow{t} through a set of inference rules.

Question 3 Give the derivation of the semantics of the instruction `{print(34) print(60-5)}` in an empty memory and an empty environment.

Determinism of the Semantics. Similar to what we did in class, we can ask ourselves about the determinism of the semantics of GOO. Since this is not entirely trivial, let’s take a few moments to think about it.

Question 4 Show that the semantics of expressions is **not** deterministic, in the sense that if $M, E, e \rightarrow M_1, v_1$ and $M, E, e \rightarrow M_2, v_2$, then we do not necessarily have $v_1 = v_2$ or even $M_1 = M_2$.

Question 5 Provide convincing arguments to show that, nevertheless, the trace semantics of GOO is deterministic, in the sense that if $M, E, s \xrightarrow{t_1} M_1$ and $M, E, s \xrightarrow{t_2} M_2$, then $t_1 = t_2$. We do not ask for a formal proof down to the last detail, but we do ask to introduce at least the key ingredients (definitions, lemmas).

Compilation to Goo. We want to convince ourselves that our language GOO is not too simple by studying the possibility of executing arbitrarily complex programs in it. To do this, we consider the small language WHILE whose abstract syntax is as follows:

| | | | |
|-----------|------------------------------------|--|-------------------|
| $e ::= n$ | <i>constant</i> $n \in \mathbb{Z}$ | $s ::= \{s \dots s\}$ | <i>block</i> |
| x | <i>variable</i> | $x = e$ | <i>assignment</i> |
| $e - e$ | <i>subtraction</i> | <code>print</code> (x) | <i>print</i> |
| | | <code>while</code> $x < n$ <code>do</code> s | <i>loop</i> |

A WHILE program is reduced to an instruction s . All variables are global (not explicitly declared) and only contain integers. The semantics of this language should be clear. To compile a WHILE program to GOO, it is sufficient to define a type **State** for the set of its variables, for example like this for a program manipulating three variables `a`, `b`, `c`:

```
struct State { a int, b int, c int }
```

Then, define a function `run` on this state, like this:

```
func (s *State) run() { ... }
```

And finally, execute the instruction `new(State).run()`.

Question 6 Provide a compilation scheme for the body of the function `run`.

Question 7 Equip the language WHILE with a big-step semantics analogous to that of GOO, with:

- a relation $E, e \rightarrow n$ for the evaluation of an expression e ;
- a relation $E, s \xrightarrow{t} E'$ for the evaluation of an instruction s .

State the theorem of correctness of the compilation from WHILE to GOO. (No proof is required.)

Syntactic Analysis. We want to perform the syntactic analysis of the GOO language using the Menhir tool. For expressions, we write the following grammar:

```
expr :  
| CST           { ... }  
| NIL           { ... }  
| IDENT        { ... }  
| expr MINUS expr { ... }  
| expr DOT IDENT { ... }  
| NEW LEFTPAR IDENT RIGHTPAR { ... }  
| LEFTPAR expr RIGHTPAR      { ... }
```

(The semantic actions are not of interest here and are omitted, as are the declarations of terminal symbols.)

Question 8 The Menhir tool reports several conflicts. Identify and explain them. Propose a solution to resolve these conflicts by adding priority and/or associativity indications.

Static Typing of Goo. The GOO language is equipped with a static typing system whose rules are given in Figure 4. The judgment $\Gamma \vdash e : \tau$ means that the expression e is well-typed with type τ in the context Γ , and the judgment $\Gamma \vdash s$ means that the instruction s is well-typed in the context Γ . As usual, the context Γ is a function from variables to types. In the typing rules, the notation $S\{y \tau\}$ means that the structure S has a field y of type τ .

Question 9 Are these typing rules syntax-directed? Discuss the implementation of these rules in a typing algorithm (for now, limiting to expressions and instructions, *i.e.*, the rules given in Figure 4).

Question 10 Propose a typing rule to verify the conformity of a function definition.

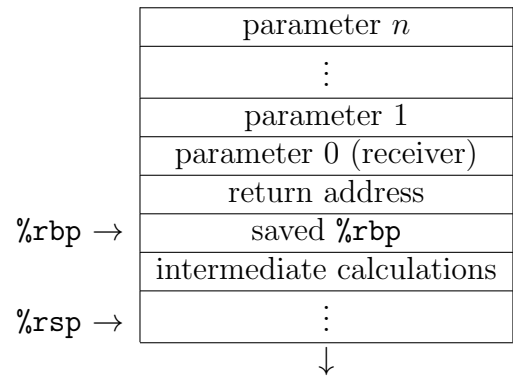
Question 11 Propose an algorithm to type a program. Recall that structures and functions are mutually recursive.

Question 12 In the GOO language, type safety is quite relative, as evaluation can always fail dynamically on the dereferencing of a null pointer. That being said, we can hope for a weakened version of type safety, as follows:

- if $\Gamma \vdash e : \text{int}$ and if $M, E, e \rightarrow M', v$ then the value v is an integer;
- if $\Gamma \vdash e : *S$ and if $M, E, e \rightarrow M', v$ then the value v is either `nil` or an address.

Give necessary conditions on the memory M , the environment E , and the context Γ so that we can hope to prove these two results.

Compilation of Goo to x86-64. We propose to compile our small language to x86-64 assembly. (A cheat sheet is provided in the appendix.) We assume that the integers in our language are limited to 64-bit signed integers. A value is either an integer or a pointer to a structure allocated on the heap (in which the field values are juxtaposed). The value `nil` is represented by the integer 0. We adopt a simple compilation scheme where all parameters are passed on the stack, including the receiver, and where we do not attempt register allocation, using the stack to store intermediate calculations. The stack frame then takes the form shown on the right.



In this context, we denote $C(e)$ as the compilation of an expression e , in the form of a piece of assembly code that puts the value of e in the register `%rax`, and $C(s)$ as the compilation of an instruction s .

Question 13 Give the definition of C in the following cases:

1. an expression x ;
2. an expression $e.y$;
3. an instruction `if $e < 0$ then s_1 else s_2 .`

Question 14 Propose an assembly code for the function `addfib` below, optimizing the tail call.

```

struct Out { v int }
func (res *Out) addfib(n int) {
    if n - 2 < 0 { res.v = res.v - (0 - n) }
    else { res.addfib(n - 2) res.addfib(n - 1) }
}

```

Adding an Object Layer. In this last part, we add an object layer to the GOO language. To do this, we add the possibility to declare **interfaces** alongside structure declarations and function definitions to our language. The abstract syntax in Figure 1 is modified as follows:

$$\begin{array}{ll}
 A ::= \text{ intf } I \{ f(x \tau, \dots, x \tau) & \text{interface} \\
 & \vdots \\
 & f(x \tau, \dots, x \tau) \} \\
 \tau ::= \text{ int } \mid *S \mid I & \text{type} \\
 p ::= A \dots A D \dots D F \dots F & \text{program}
 \end{array}$$

Here is an example of an interface `Rope` declaring two functions `len` and `nth` (reusing the structure `R` given at the very beginning of the subject):

```

intf Rope {
  len(out *R)
  nth(i int, out *R)
}

```

We say that a structure `S` **satisfies** an interface `I` as soon as all the functions declared in `I` are implemented for the receiver type `*S`. For instance, the type `L` of integer lists given at the beginning of the subject satisfies the interface `Rope` because we have defined the functions `len` and `nth` on the receiver `*L` with the expected profiles. It is not necessary to explicitly declare that `L` satisfies the interface `Rope`; the compiler is capable of verifying this on its own if needed. Here is another example of a structure satisfying the interface `Rope`:

```

struct Leaf { n int }
func (e *Leaf) len(out *R) { out.r = 1 }
func (e *Leaf) nth(i int, out *R) { out.r = e.n }

```

As indicated above (new definition of τ), an interface is also a type. Thus, we can define a third structure, `App`, whose fields have the type `Rope`:

```

struct App { left Rope, right Rope }

```

In particular, we can assign values of type `*L` or `*Leaf` to the fields `left` and `right` of an `App` structure because these two structures satisfy the interface `Rope`. In other words, we have the following typing rule:

$$\frac{\Gamma \vdash e : *S \quad S \text{ satisfies } I}{\Gamma \vdash e : I}$$

Finally, on the `App` structure itself, we can also implement the functions expected by the `Rope` interface, like this:

```

func (e *App) len(out *R) { e.left.len(out)  out.addlen(out.r, e.right) }
func (out *R) addlen(n int, r Rope) { r.len(out)  out.r = out.r - (0 - n) }
func (e *App) nth(i int, out *R) {
  e.left.len(out)
  if i-out.r < 0 then { e.left.nth(i, out) } else { e.right.nth(i-out.r, out) }
}

```

In particular, we can therefore build binary trees with internal nodes of type `App` and leaves of type `L` or `Leaf`, including mixing the two in the same tree.

Question 15 For the last function above (the `nth` function on the `App` type), give the static type of each sub-expression. Explain why we can see GOO as an object language, even though there are neither classes nor inheritance.

Interface Value. To compile this extension of the GOO language to x86-64, we extend our compilation scheme with a new type of value, called an **interface value**. This is a pointer to a two-word block allocated on the heap. The first word contains the “name” of a structure S (an identifier). The second word contains either `nil` (*i.e.*, 0) or a pointer to a structure (of type S) allocated on the heap. The idea is, as in any object language, to use the name of the structure to find, during execution, the code of the function to call in a table built during compilation.

Question 16 Propose a solution to represent structure names on the one hand and function tables on the other. You can assume that we have the entire source program at the time of compilation. Illustrate with the program above, *i.e.*, with the structures `L`, `Leaf`, and `App` and the interface `Rope`.

Question 17 Explain at which points in the program interface values are constructed by the compiler, and how.

Question 18 Give the compilation scheme for a call $e.f(e_1, \dots, e_n)$. Clearly distinguish two cases, depending on whether the static type of e is $*S$ or I .

Question 19 Still in the context of the declarations above, what does the following program inspire you?

```
func (out *R) q0()           { out.q1(new(App), nil)           }
func (out *R) q1(a *App, l *L) { a.left = l  a.right = l  a.len(out) }
```

Should it be rejected during typing, and if so, why? Otherwise, can we hope to execute it successfully, and if so, how?

(The GOO language is directly inspired by the Go language.)

| | |
|--|--|
| $e ::= n$ nil x $e - e$ $e.y$ $\text{new}(S)$ | <i>constant</i> $n \in \mathbb{Z}$ <i>null pointer</i> <i>variable</i> <i>subtraction</i> <i>field access</i> <i>allocation</i> |
| $s ::= e.y = e$ $e.f(e, \dots, e)$ $\text{print}(e)$ $\text{if } e = \text{nil} \text{ then } s \text{ else } s$ $\text{if } e < 0 \text{ then } s \text{ else } s$ $\{s \dots s\}$ | <i>assignment</i> <i>function call</i> <i>print</i> <i>conditional</i> <i>conditional</i> <i>block</i> |
| $\tau ::= \text{int} \mid *S$ $D ::= \text{struct } S \{y \tau, \dots, y \tau\}$ $F ::= \text{func } (x *S) f(x \tau, \dots, x \tau) s$ $p ::= D \dots D F \dots F$ | <i>type</i> <i>structure declaration</i> <i>function definition</i> <i>program</i> |

Figure 1: Abstract syntax of GOO.

| | |
|---|---|
| $v ::= n$ nil ℓ | <i>integer value</i> $n \in \mathbb{Z}$ <i>null pointer</i> <i>memory address</i> |
| $d(\text{int}) \stackrel{\text{def}}{=} 0$ $d(*S) \stackrel{\text{def}}{=} \text{nil}$ | |
| $\frac{}{M, E, n \rightarrow M, n} \quad \frac{}{M, E, \text{nil} \rightarrow M, \text{nil}} \quad \frac{x \in \text{dom}(E)}{M, E, x \rightarrow M, E(x)}$ $\frac{M, E, e_1 \rightarrow M_1, n_1 \quad M_1, E, e_2 \rightarrow M_2, n_2}{M, E, e_1 - e_2 \rightarrow M_2, n_1 - n_2} \quad \frac{M, E, e \rightarrow M_1, \ell \quad y \in \text{dom}(M_1(\ell)) \quad M_1(\ell)(y) = v}{M, E, e.y \rightarrow M_1, v}$ $\frac{\text{struct } S \{y_1 \tau_1, \dots, y_n \tau_n\} \quad \ell \notin \text{dom}(M) \quad M_1 = M \oplus \{\ell \mapsto \{y_1 \mapsto d(\tau_1), \dots, y_n \mapsto d(\tau_n)\}\}}{M, E, \text{new}(S) \rightarrow M_1, \ell}$ | |

Figure 2: Operational semantics of GOO (expressions).

$$\begin{array}{c}
\frac{M, E, e \rightarrow M_1, n}{M, E, \text{print}(e) \xrightarrow{n} M_1} \quad \frac{M, E, e_1 \rightarrow M_1, \ell \quad y \in \text{dom}(M_1(\ell)) \quad M_1, E, e_2 \rightarrow M_2, v_2}{M, E, e_1.y = e_2 \xrightarrow{\varepsilon} M_2 \oplus \{\ell \mapsto M_1(\ell) \oplus \{y \mapsto v_2\}\}} \\
\\
\frac{M, E, e \rightarrow M_1, v \quad \forall i. M_i, E, e_i \rightarrow M_{i+1}, v_i \quad \text{func } (x *S) f(x_1 \tau_1, \dots, x_n \tau_n) s}{M_{n+1}, \{x \mapsto v, x_1 \mapsto v_1, x_2 \mapsto v_2, \dots, x_n \mapsto v_n\}, s \xrightarrow{t} M'} \\
\frac{M, E, e.f(e_1, \dots, e_n) \xrightarrow{t} M'}{} \\
\\
\frac{M, E, e \rightarrow M_1, \text{nil} \quad M_1, E, s_1 \xrightarrow{t} M_2}{M, E, \text{if } e = \text{nil} \text{ then } s_1 \text{ else } s_2 \xrightarrow{t} M_2} \quad \frac{M, E, e \rightarrow M_1, \ell \quad M_1, E, s_2 \xrightarrow{t} M_2}{M, E, \text{if } e = \text{nil} \text{ then } s_1 \text{ else } s_2 \xrightarrow{t} M_2} \\
\frac{M, E, e \rightarrow M_1, n \quad n < 0 \quad M_1, E, s_1 \xrightarrow{t} M_2}{M, E, \text{if } e < 0 \text{ then } s_1 \text{ else } s_2 \xrightarrow{t} M_2} \quad \frac{M, E, e \rightarrow M_1, n \quad n \geq 0 \quad M_1, E, s_2 \xrightarrow{t} M_2}{M, E, \text{if } e < 0 \text{ then } s_1 \text{ else } s_2 \xrightarrow{t} M_2} \\
\\
\frac{}{M, E, \{ \} \xrightarrow{\varepsilon} M} \quad \frac{M, E, s_1 \xrightarrow{t_1} M_1 \quad M_1, E, \{s_2 \dots\} \xrightarrow{t_2} M_2}{M, E, \{s_1 s_2 \dots\} \xrightarrow{t_1 t_2} M_2}
\end{array}$$

Figure 3: Operational semantics of GOO (instructions).

$$\begin{array}{c}
\frac{}{\Gamma \vdash n : \text{int}} \quad \frac{}{\Gamma \vdash \text{nil} : *S} \quad \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \\
\\
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 - e_2 : \text{int}} \quad \frac{\Gamma \vdash e : *S \quad S\{y \tau\}}{\Gamma \vdash e.y : \tau} \quad \frac{S \text{ exists}}{\Gamma \vdash \text{new}(S) : *S} \\
\\
\frac{\Gamma \vdash e_1 : *S \quad S\{y \tau\} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1.y = e_2} \quad \frac{\Gamma \vdash e : \text{int}}{\Gamma \vdash \text{print}(e)} \\
\\
\frac{\text{func } (x *S) f(x_1 \tau_1, \dots, x_n \tau_n) \quad \Gamma \vdash e : *S \quad \forall i. \Gamma \vdash e_i : \tau_i}{\Gamma \vdash e.f(e_1, \dots, e_n)} \\
\\
\frac{\Gamma \vdash e : *S \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \text{if } e = \text{nil} \text{ then } s_1 \text{ else } s_2} \quad \frac{\Gamma \vdash e : \text{int} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \text{if } e < 0 \text{ then } s_1 \text{ else } s_2} \quad \frac{\forall i. \Gamma \vdash s_i}{\Gamma \vdash \{s_1 s_2 \dots s_n\}}
\end{array}$$

Figure 4: Static typing of GOO.

Appendix: x86-64 cheat sheet

A fragment of the x86-64 instruction set is given here. You are free to use any other part of the x86-64 assembler. In the following, r_i designates a register, n an integer constant and L a label.

| | |
|---|---|
| <code>mov r_2, r_1</code> | copies register r_2 into register r_1 |
| <code>mov $\\$n, r_1$</code> | loads constant n into register r_1 |
| <code>mov $\\$L, r_1$</code> | loads the address of label L into register r_1 |
| <code>sub r_2, r_1</code> | computes $r_1 - r_2$ and stores it into r_1 |
| <code>neg r_1</code> | computes $-r_1$ and stores it into r_1 |
| <code>mov $n(r_2), r_1$</code> | loads r_1 with the value contained in memory at address $r_2 + n$ |
| <code>mov $r_1, n(r_2)$</code> | writes in memory at address $r_2 + n$ the value of r_1 |
| <code>push r_1</code> | pushes the value of r_1 on the stack |
| <code>pop r_1</code> | pops a value from the stack and stores it into register r_1 |
| <code>test r_2, r_1</code> | sets the flags according to the value of r_1 AND r_2 |
| <code>jz L</code> | jumps to address L if flags signal a zero value |
| <code>jmp L</code> | jumps to address L |
| <code>call L</code> | pushes the return address to the stack and jumps to address L |
| <code>ret</code> | pops an address from the stack and jumps there |

Calling conventions:

- up to six arguments are passed via registers `%rdi, %rsi, %rdx, %rcx, %r8, %r9`;
- other arguments are passed on the stack, if any;
- the returned value is put in `%rax`;
- registers `%rsp, %rbp, %rbx, %r12, %r13, %r14` and `%r15` are *callee-saved*: they won't be clobbered by a `call`;
- the other registers are *caller-saved*: they may be clobbered by a `call`;
- `%rsp` is the stack pointer, `%rbp` the *frame pointer*.