École Polytechnique

CSC_52064 : Compilation

examen 2026

Jean-Christophe Filliâtre

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 16.

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 $0 \leq \texttt{i} < n$ where n is the length of the list `l`.

Correction :

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

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 $n \geq 0$.)

Correction :

```
struct P { l *L }
func (out *P) fib_below(max int) {  out.fib_below_aux(0, 1, max) }

// adds F(n)... to the list, with a=F(n) and b=F(n+1)
func (out *P) fib_below_aux(a int, b int, max int) {
  if a - max < 0 then {
    out.fib_below_aux(b, a - (0 - b), max)  // adds F(n+1)... to the list
    out.cons(a, new(L))                     // then a at the front
  } else { out.l = nil }
}

func (out *P) cons(v int, l *L) {
  l.head = v
  l.next = out.l
  out.l = l
}
```

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 $\ell \in \mathrm{dom}(M)$ and $y \in \mathrm{dom}(M(\ell))$. An **environment** E is a function from variables to values.

For expressions, we introduce a relation $M, E, e \twoheadrightarrow M', v$ 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:

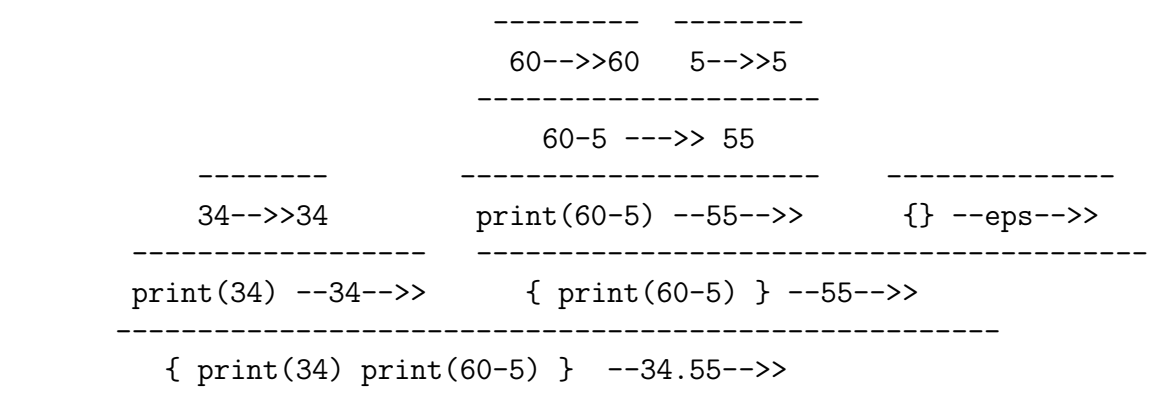
$$\begin{cases} M'(x) &= v, \\ M'(y) &= M(y) \text{ for } y \in \mathrm{dom}(M) \text{ and } y \neq x. \end{cases}$$

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

For instructions, we introduce a relation $M, E, s \xrightarrow{t}\!\!\!\!\!\rightarrow M'$ 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 $\xrightarrow{t}\!\!\!\!\!\rightarrow$ 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.

Correction :



(The memories and environments have been systematically omitted, as they are empty.)

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 $M, E, e \twoheadrightarrow M_1, v_1$ and $M, E, e \twoheadrightarrow M_2, v_2$, then we do not necessarily have $v_1 = v_2$ or even $M_1 = M_2$.

Correction : It is sufficient to consider an expression that involves an allocation, as simple as `new(S)` for example, because we then obtain two fresh addresses that are not necessarily equal, and thus two possibly different memories M_1 and M_2.

Question 5 Provide convincing arguments to show that, nevertheless, the trace semantics of Goo **is** deterministic, in the sense that if $M, E, s \xrightarrow{t_1}\!\!\!\!\twoheadrightarrow M_1$ and $M, E, s \xrightarrow{t_2}\!\!\!\!\twoheadrightarrow M_2$, 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).

Correction : The general idea is that, even though the semantics is not deterministic due to the address returned by `new`, the memories and environments of the same computation remain isomorphic.

We can formalize this as follows. If θ is a mapping from addresses to addresses, we denote $v_1 \overset{\theta}{\approx} v_2$ the relation defined by:

$$n \overset{\theta}{\approx} n \qquad \texttt{nil} \overset{\theta}{\approx} \texttt{nil} \qquad \ell \overset{\theta}{\approx} \theta(\ell)$$

That is, θ maps the value v_1 to the value v_2.

Then, we define an isomorphism for environments:

$$E_1 \overset{\theta}{\approx} E_2 \overset{\text{def}}{=} \text{dom}(E_1) = \text{dom}(E_2) \ \wedge \ \forall x.\, E_1(x) \overset{\theta}{\approx} E_2(x)$$

Another for memories:

$$\begin{aligned} M_1 \overset{\theta}{\approx} M_2 \ \overset{\text{def}}{=} \ & (\forall \ell.\, \ell \in \text{dom}(M_1) \Leftrightarrow \theta(\ell) \in \text{dom}(M_2)) \ \wedge \\ & \forall \ell.\, \ell \in \text{dom}(M_1) \Rightarrow \text{dom}(M_1(\ell)) = \text{dom}(M_2(\theta(\ell))) \ \wedge \ \forall y.\, M_1(\ell)(y) \overset{\theta}{\approx} M_2(\theta(\ell)) \end{aligned}$$

And finally:

$$M_1, E_1 \overset{\theta}{\approx} M_2, E_2 \overset{\text{def}}{=} M_1 \overset{\theta}{\approx} M_2 \ \wedge \ E_1 \overset{\theta}{\approx} E_2$$

We can then state (and prove, even if we do not do it here) the following properties. For expressions:

> If $M_1, E_1 \overset{\theta}{\approx} M_2, E_2$, if $M_1, E_1, e \twoheadrightarrow M_1', v_1$ and if $M_2, E_2, e \twoheadrightarrow M_2', v_2$, then there exists θ' such that $M_1' \overset{\theta'}{\approx} M_2'$ and $v_1 \overset{\theta'}{\approx} v_2$.

For instructions:

> If $M_1, E_1 \overset{\theta}{\approx} M_2, E_2$, if $M_1, E_1, s \xrightarrow{t_1}\!\!\!\!\!\rightarrow M_1'$ and if $M_2, E_2, s \xrightarrow{t_2}\!\!\!\!\!\rightarrow M_2'$, then $t_1 = t_2$ and there exists θ' such that $M_1' \overset{\theta'}{\approx} M_2'$.

We deduce the desired result (since trivially $M, E \overset{\theta}{\approx} M, E$ by the identity).

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	*constant* $n \in \mathbb{Z}$	s	::=	`{` $s \ldots s$ `}` *block*
		$\mid$ x	*variable*			$\mid$ x `=` e *assignment*
		$\mid$ e `-` e	*subtraction*			$\mid$ `print(`x`)` *print*
						$\mid$ `while` x `<` n `do` s *loop*

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`.

Correction : The compilation of WHILE expressions is immediate:

```
C(n)     := n
C(x)     := s.x
C(e1-e2) := C(e1)-C(e2)
```

For instructions, we number the different loops of the WHILE program, then translate the expressions as follows:

```
C({s1 s2 ... sn})   := { C(s1) C(s2) ... C(sn) }
C(x = e)            := s.x = C(e)
C(print(x))         := print(s.x)
C(while x < n do s) := s.while_i()
  with
  func (s *State) while_i() { if s.x - n < 0 { C(s) s.while_i() } else {} }
```

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

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

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

Correction : For expressions:

```
                    x in dom(E)          E,e1 -->> n1  E,e2 -->> n2
-------------     -----------------     -------------------------
 E,n -->> n         E,x -->> E(x)             E,e1-e2 -->> n1-n2
```

For instructions:

```
      E,e -->> n                      E(x) = n
-----------------------     --------------------------
 E,x=e --eps-->> E+{x->n}     E,print(x) --n-->> E

      E(x) >= n
------------------------------
 E,while x<n do s --eps-->> E

    E(x) < n   E,{s while x<n do s} --t-->> E'
 ---------------------------------------------
             E,while x<n do s --t-->> E'

                          E,s1 --t1-->> E1   E1,{s2...} --t2-->> E2
 -----------------     ---------------------------------------------
 E,{} --eps-->> E                E,{s1 s2...} --t1t2-->> E2
```

The correctness theorem can be written, for example, as:

$$\text{if } E_0, s \xrightarrow{t}\!\!\!\!\rightarrow E' \text{ then } \emptyset, \emptyset, \texttt{new(State).run()} \xrightarrow{t}\!\!\!\!\rightarrow M'$$

where E_0 is an environment that maps all global variables of s to 0.

It is not necessary to mention M' in this theorem, as the equality of traces is sufficient to capture all possible behaviors.

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.

Correction : The Menhir tool reports two shift/reduce conflicts, corresponding respectively to:

- `expr MINUS expr MINUS expr`
- `expr MINUS expr DOT IDENT`

In both cases, when reading the token following the second `expr`, we can either reduce `expr MINUS expr` or shift.

A natural solution consists in declaring:

```
%left MINUS
%nonassoc DOT
```

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).

Correction : Yes, the rules are indeed syntax-directed.

The only difficulty concerning their implementation in a typing algorithm is the typing rule for `nil`, which assumes guessing the structure S. At least two solutions exist:

- perform type inference, giving `nil` a type variable that will be determined later, through unification, when the type of an expression is confronted with an expected type;
- only accept `nil` in positions where the context immediately allows determining its type, *i.e.*, an assignment (we have the type of the field) or a parameter of a call (we have the expected type for the parameter).

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

Correction : The function definition

$$\texttt{func (}x\ \texttt{*}S\texttt{)}\ f\texttt{(}x_1\ \tau_1, \ldots, x_n\ \tau_n\texttt{)}\ s$$

is well-typed if

- the structure S exists;
- the names $x, x_1, \ldots, x_n$ are pairwise distinct;
- the types τ_i are well-formed;
- we have $x : *S, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash s$

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

Correction : It is sufficient to proceed in three steps:

1. declare all structures (but ignoring their fields for now);
2. declare all structure fields and the profiles of all functions, each time verifying that the mentioned structures do indeed exist;
3. type all function bodies (see the previous question).

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 $\Gamma \vdash e : \texttt{int}$ and if $M, E, e \twoheadrightarrow M', v$ then the value v is an integer;
- if $\Gamma \vdash e : *S$ and if $M, E, e \twoheadrightarrow M', v$ 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.

Correction : We need to express the consistency between the memory M, the environment E, and the typing context Γ, in the sense that:

- the value of a variable of type `int` is an integer;
- the value of a variable of type $*S$ is either `nil`, or an address at which is a structure that is itself well-formed with respect to typing.

We can do this formally as follows. We start by defining $M \vdash v : \tau$, which means that the value v is well-typed with type τ, as follows:

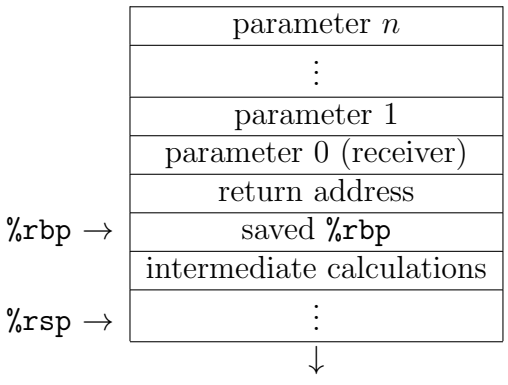
$$\frac{}{M \vdash n : \texttt{int}} \qquad \frac{}{M \vdash \texttt{nil} : *S}$$

$$\frac{\ell \in \mathrm{dom}(M) \quad \mathrm{dom}(M(\ell)) = \mathrm{dom}(S) \quad \forall y : \tau \in S.\, M \vdash M(\ell)(y) : \tau}{M \vdash \ell : *S}$$

Then, we define the consistency relation with these two conditions:

- $\mathrm{dom}(E) = \mathrm{dom}(\Gamma)$;
- for all $x \in \mathrm{dom}(E)$, we have $M \vdash E(x) : \Gamma(x)$.

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.

Correction :

```
C(x) =>
  mov  ofs(%rbp), %rax

C(e.y) =>
  C(e)
  mov  ofs(%rax), %rax

C(if e < 0 then s1 else s2) =>
  C(e)
  cmpq $0, %rax
  jge  L1  # if e-0 >= 0
  C(s1)
  jmp  L2
L1:C(s2)
L2:
```

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) }
}
```

Correction :

```
# res = %rbp + 16
#   n = %rbp + 24
addfib: pushq %rbp
        movq  %rsp, %rbp
        movq  24(%rbp), %rdi  # n
        movq  16(%rbp), %rsi  # res
        cmpq  $2, %rdi
        jl    1f
        addq  $-2, %rdi
```

```
    pushq %rdi
    pushq %rsi
    call  addfib
    addq  $16, %rsp       # pop parameters
    addq  $-1, 24(%rbp)   # n <- n-1
    jmp   addfib          # tail call
1:  addq  %rdi, (%rsi)
    popq  %rbp
    ret
```

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:

$$
\begin{array}{rcll}
A & ::= & \texttt{intf}\ I\ \texttt{\{}\ f\texttt{(}x\ \tau, \ldots, x\ \tau\texttt{)} & \textit{interface} \\
 & & \quad\vdots & \\
 & & f\texttt{(}x\ \tau, \ldots, x\ \tau\texttt{)}\ \texttt{\}} & \\
\tau & ::= & \texttt{int} \mid \texttt{*}S \mid I & \textit{type} \\
p & ::= & A \ldots A\ D \ldots D\ F \ldots F & \textit{program}
\end{array}
$$

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:

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

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.

Correction :

```
func (e *App) nth(i int, out *R) {
  e    .left.len(out)
  ----           ---
  *App           *R
  ----------
     Rope

  if i-out.r < 0 then { e   .left.nth(i   , out) }
     -------            ----          ---  ---
       int              *App          int   *R
                        ---------
                          Rope

                  else { e    .right.nth(i-out.r, out) }
                         ----            -------  ---
                         *App              int     *R
                         -----------
                           Rope
}
```

As we can see, for example, above with `e.left.nth`, the static type of the receiver, here `Rope`, does not allow selecting the appropriate `nth` function. We must therefore resort to a dynamic method call.

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`.

Correction : It is sufficient to number the structures with consecutive integers $0, 1, \ldots$, arbitrarily. The first word of an interface value is then this integer.

The function tables are then allocated in the data segment, following the order of this numbering of structures. Thus, with `L`, `Leaf`, and `App` numbered respectively 0, 1, and 2, we will have:

```
.data
table_len:
 .quad len_L
 .quad len_Leaf
 .quad len_App
table_nth:
 .quad nth_L
 .quad nth_Leaf
 .quad nth_App
```

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

Correction : Invariant: an expression evaluates to an interface value if and only if its static type is an interface.

Therefore, we construct an interface value exactly at the points where a value of type $*S$ is converted (by the typing rule above) to an interface type. This occurs in two places in the code:

- an assignment;
- a parameter passing in a call.

An elegant way to implement this is to insert an explicit coercion in the AST during typing. The static type of the argument of this coercion is exactly the type that needs to be stored in the first component of the interface value.

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

Correction : We distinguish two cases, depending on the type of e:

- for e of type $*S$: we directly call the function f for the receiver $*S$

```
C(en) pushq %rax
...
C(e1) pushq %rax
C(e)  pushq %rax
call  f
addq  $8(n+1), %rsp
```

- for e of type I: we know by invariant that the value of e is an interface value

```
C(en) pushq %rax
...
C(e1) pushq %rax
C(e)  pushq 8(%rax)     # we push the receiver, not the interface value
movq  (%rax), %rcx            # structure number
movq  table_f(,%rcx,8), %rcx  # code retrieved from the table of f
call  *%rcx                   # dynamic call
addq  $8(n+1), %rsp
```

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?

Correction : The program above is well-typed, in particular because:

- the value `nil` is accepted as the second parameter of `q1`, because $\vdash$ `nil` : `*L`;
- the two assignments of `a.left` and `a.right` are well-typed, because `l` is of type `*L` and `L` satisfies `Rope`.

It executes successfully because, during the two assignments, we construct an interface value with the structure `L` and the value `nil`. Then, during the two calls to `len` hidden in the call `a.len(out)`, the dynamic call will then select the implementation of `len` on the type `*L`, which is indeed defined on the receiver `nil` (see the very first page of the subject).

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

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

Figure 1: Abstract syntax of Goo.

$$
\begin{array}{rcll}
v & ::= & n & \textit{integer value } n \in \mathbb{Z} \\
& | & \texttt{nil} & \textit{null pointer} \\
& | & \ell & \textit{memory address}
\end{array}
$$

$$
\begin{array}{rcl}
d(\texttt{int}) & \stackrel{\text{def}}{=} & 0 \\
d(\texttt{*}S) & \stackrel{\text{def}}{=} & \texttt{nil}
\end{array}
$$

$$
\frac{}{M, E, n \twoheadrightarrow M, n} \qquad \frac{}{M, E, \texttt{nil} \twoheadrightarrow M, \texttt{nil}} \qquad \frac{x \in \mathrm{dom}(E)}{M, E, x \twoheadrightarrow M, E(x)}
$$

$$
\frac{M, E, e_1 \twoheadrightarrow M_1, n_1 \quad M_1, E, e_2 \twoheadrightarrow M_2, n_2}{M, E, e_1 \texttt{ - } e_2 \twoheadrightarrow M_2, n_1 - n_2} \qquad \frac{M, E, e \twoheadrightarrow M_1, \ell \quad y \in \mathrm{dom}(M_1(\ell)) \quad M_1(\ell)(y) = v}{M, E, e\texttt{.}y \twoheadrightarrow M_1, v}
$$

$$
\frac{\texttt{struct } S \texttt{ \{} y_1\ \tau_1, \ldots, y_n\ \tau_n \texttt{ \}} \quad \ell \notin \mathrm{dom}(M) \quad M_1 = M \oplus \{\ell \mapsto \{y_1 \mapsto d(\tau_1), \ldots, y_n \mapsto d(\tau_n)\}\}}{M, E, \texttt{new(}S\texttt{)} \twoheadrightarrow M_1, \ell}
$$

Figure 2: Operational semantics of Goo (expressions).

$$\frac{M, E, e \twoheadrightarrow M_1, n}{M, E, \texttt{print(}e\texttt{)} \xrightarrow{n}\!\!\!\!\twoheadrightarrow M_1} \qquad \frac{M, E, e_1 \twoheadrightarrow M_1, \ell \quad y \in \mathrm{dom}(M_1(\ell)) \quad M_1, E, e_2 \twoheadrightarrow M_2, v_2}{M, E, e_1\texttt{.}y\ \texttt{=}\ e_2 \xrightarrow{\varepsilon}\!\!\!\!\twoheadrightarrow M_2 \oplus \{\ell \mapsto M_1(\ell) \oplus \{y \mapsto v_2\}\}}$$

$$\frac{\begin{array}{c} M, E, e \twoheadrightarrow M_1, v \quad \forall i.\, M_i, E, e_i \twoheadrightarrow M_{i+1}, v_i \\ \texttt{func (}x\ \texttt{*}S\texttt{) }f\texttt{(}x_1\ \tau_1, \ldots, x_n\ \tau_n\texttt{) }s \\ M_{n+1}, \{x \mapsto v, x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_n \mapsto v_n\}, s \xrightarrow{t}\!\!\!\!\twoheadrightarrow M' \end{array}}{M, E, e\texttt{.}f\texttt{(}e_1, \ldots, e_n\texttt{)} \xrightarrow{t}\!\!\!\!\twoheadrightarrow M'}$$

$$\frac{M, E, e \twoheadrightarrow M_1, \texttt{nil} \quad M_1, E, s_1 \xrightarrow{t}\!\!\!\!\twoheadrightarrow M_2}{M, E, \texttt{if } e\ \texttt{=}\ \texttt{nil then } s_1 \texttt{ else } s_2 \xrightarrow{t}\!\!\!\!\twoheadrightarrow M_2} \qquad \frac{M, E, e \twoheadrightarrow M_1, \ell \quad M_1, E, s_2 \xrightarrow{t}\!\!\!\!\twoheadrightarrow M_2}{M, E, \texttt{if } e\ \texttt{=}\ \texttt{nil then } s_1 \texttt{ else } s_2 \xrightarrow{t}\!\!\!\!\twoheadrightarrow M_2}$$

$$\frac{M, E, e \twoheadrightarrow M_1, n \quad n < 0 \quad M_1, E, s_1 \xrightarrow{t}\!\!\!\!\twoheadrightarrow M_2}{M, E, \texttt{if } e\ \texttt{<}\ \texttt{0 then } s_1 \texttt{ else } s_2 \xrightarrow{t}\!\!\!\!\twoheadrightarrow M_2} \qquad \frac{M, E, e \twoheadrightarrow M_1, n \quad n \geq 0 \quad M_1, E, s_2 \xrightarrow{t}\!\!\!\!\twoheadrightarrow M_2}{M, E, \texttt{if } e\ \texttt{<}\ \texttt{0 then } s_1 \texttt{ else } s_2 \xrightarrow{t}\!\!\!\!\twoheadrightarrow M_2}$$

$$\frac{}{M, E, \texttt{\{ \}} \xrightarrow{\varepsilon}\!\!\!\!\twoheadrightarrow M} \qquad \frac{M, E, s_1 \xrightarrow{t_1}\!\!\!\!\twoheadrightarrow M_1 \quad M_1, E, \texttt{\{}\ s_2 \ldots\ \texttt{\}} \xrightarrow{t_2}\!\!\!\!\twoheadrightarrow M_2}{M, E, \texttt{\{}\ s_1\ s_2 \ldots\ \texttt{\}} \xrightarrow{t_1 t_2}\!\!\!\!\twoheadrightarrow M_2}$$

Figure 3: Operational semantics of Goo (instructions).

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash \texttt{nil} : \texttt{*}S} \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\ \texttt{-}\ e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e : \texttt{*}S \quad S\{y\ \tau\}}{\Gamma \vdash e\texttt{.}y : \tau} \qquad \frac{S \text{ exists}}{\Gamma \vdash \texttt{new(}S\texttt{)} : \texttt{*}S}$$

$$\frac{\Gamma \vdash e_1 : \texttt{*}S \quad S\{y\ \tau\} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\texttt{.}y\ \texttt{=}\ e_2} \qquad \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{print(}e\texttt{)}}$$

$$\frac{\texttt{func (}x\ \texttt{*}S\texttt{) }f\texttt{(}x_1\ \tau_1, \ldots, x_n\ \tau_n\texttt{)} \quad \Gamma \vdash e : \texttt{*}S \quad \forall i.\, \Gamma \vdash e_i : \tau_i}{\Gamma \vdash e\texttt{.}f\texttt{(}e_1, \ldots, e_n\texttt{)}}$$

$$\frac{\Gamma \vdash e : \texttt{*}S \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if } e\ \texttt{=}\ \texttt{nil then } s_1 \texttt{ else } s_2} \qquad \frac{\Gamma \vdash e : \texttt{int} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if } e\ \texttt{<}\ \texttt{0 then } s_1 \texttt{ else } s_2} \qquad \frac{\forall i.\, \Gamma \vdash s_i}{\Gamma \vdash \texttt{\{}\ s_1\ s_2\ \ldots\ s_n\ \texttt{\}}}$$

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.

`mov` r_2`,` r_1	copies register r_2 into register r_1
`mov` `$`n`,` r_1	loads constant n into register r_1
`mov` `$`L`,` r_1	loads the address of label L into register r_1
`sub` r_2`,` r_1	computes $r_1 - r_2$ and stores it into r_1
`neg` r_1	computes $-r_1$ and stores it into r_1
`mov` n`(`r_2`),` r_1	loads r_1 with the value contained in memory at address $r_2 + n$
`mov` r_1`,` n`(`r_2`)`	writes in memory at address $r_2 + n$ the value of r_1
`push` r_1	pushes the value of r_1 on the stack
`pop` r_1	pops a value from the stack and stores it into register r_1
`test` r_2`,` r_1	sets the flags according to the value of r_1 AND r_2
`jz` L	jumps to address L if flags signal a zero value
`jmp` L	jumps to address L
`call` L	pushes the return address to the stack and jumps to address L
`ret`	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`, `%14` 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*.