École Normale Supérieure

Langages de programmation et compilation

examen 2025–2026

Jean-Christophe Filliâtre

23 janvier 2026 — 8h30–11h30

L'épreuve dure 3 heures.
Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.
Les questions sont indépendantes, au sens où il n'est pas nécessaire d'avoir répondu aux questions précédentes pour traiter une question. Mais en revanche les questions peuvent faire appel à des définitions ou à des résultats introduits dans les questions précédentes.
Sauf mention explicite du contraire, les réponses doivent être justifiées.
Les figures 1–4 sont regroupées en fin de sujet, page 15.
Suggestion : détacher les deux dernières feuilles.

Dans ce sujet, on considère un petit langage impératif appelé Goo, manipulant des entiers et des structures allouées sur le tas, dont la syntaxe abstraite est donnée figure 1. Un programme (p) est une liste de déclarations de types de structures (D), mutuellement récursives, et une liste de définitions de fonctions (F), également mutuellement récursives. Une structure S est déclarée avec la liste de ses champs, chaque champ y étant accompagné de son type τ. Un type est soit `int`, pour un entier, soit $*S$ pour un pointeur vers une structure S. Voici un exemple de programme qui définit un type `L` de liste simplement chaînée et une fonction `len` pour calculer la longueur d'une liste :

```
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 }
}
```

Comme on le voit, la définition d'une fonction et son appel distinguent le premier paramètre, appelé *receveur* et dont le type est forcément un pointeur. On note que le receveur *peut être le pointeur nul*. La sémantique est stricte, avec appel par valeur. Les fonctions ne renvoient rien, mais elles ont néanmoins des effets, soit par l'instruction `print`, soit par la modification du contenu d'une structure, comme dans l'exemple ci-dessus.

Question 1. Donner le code d'une fonction `func (l *L) nth(i int, out *R)` qui met dans `out.r` le `i`-ième élément de la liste `l`, les éléments étant comptés à partir de 0. On suppose $0 \leq \texttt{i} < n$ où n est la longueur de la liste `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. Donner de code d'une fonction `fib_below` qui construit et stocke dans `out.l` la liste des nombres de Fibonacci strictement plus petits que `max`, dans l'ordre croissant.

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

(À toutes fins utiles, on rappelle que la suite de Fibonacci (F_n) est définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_n + F_{n+1}$ pour $n \geq 0$.)

Correction :

```
struct P { l *L }
func (out *P) fib_below(max int) {  out.fib_below_aux(0, 1, max) }
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)
    out.cons(a, new(L))
  } else { out.l = nil }
}
func (out *P) cons(v int, l *L) {
  l.head = v
  l.next = out.l
  out.l = l
}
```

Sémantique. On munit Goo d'une sémantique opérationnelle à grand pas. Une **valeur** v est soit un entier n, soit la valeur `nil`, soit une adresse ℓ. Une **mémoire** M est une fonction qui associe à une adresse ℓ le contenu d'une structure, ce dernier étant lui-même une fonction des champs vers les valeurs. Ainsi, $M(\ell)(y)$ est la valeur du champ y de la structure située en mémoire à l'adresse ℓ, sous réserve que $\ell \in \mathrm{dom}(M)$ et $y \in \mathrm{dom}(M(\ell))$. Un **environnement** E est une fonction des variables vers les valeurs.

Pour les expressions, on introduit une relation $M, E, e \twoheadrightarrow M', v$ signifiant « dans la mémoire M et l'environnement E, l'expression e s'évalue avec succès en la valeur v, pour une mémoire finale M' ». (La présence de la construction `new` dans les expressions induit une modification possible de la mémoire pendant l'évaluation d'une expression.) La figure 2 définit cette relation par un ensemble de règles d'inférence. Dans ces règles, la notation $M \oplus \{x \mapsto v\}$ désigne la fonction M' définie par

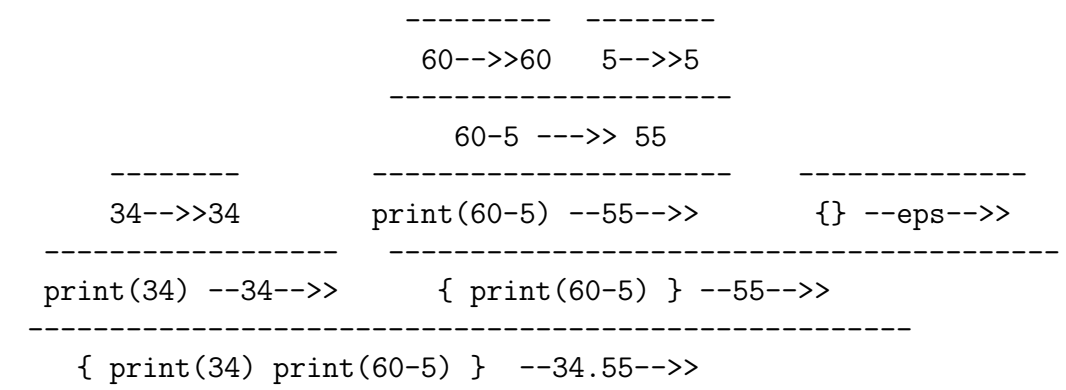
$$\begin{cases} M'(x) &= v, \\ M'(y) &= M(y) \ \text{ pour } y \in \mathrm{dom}(M) \text{ et } y \neq x. \end{cases}$$

En particulier, on a $\mathrm{dom}(M') = \mathrm{dom}(M) \cup \{x\}$.

Pour les instructions, on introduit une relation $M, E, s \xrightarrow{t}\!\!\!\!\twoheadrightarrow M'$ signifiant « dans la mémoire M et l'environnement E, l'instruction s s'évalue avec succès, pour une mémoire finale M' et une trace t ». Une **trace** est la séquence des entiers affichés par `print`, représentée par un mot sur l'alphabet $\mathbb{Z}$. On note ε le mot vide et $t_1 t_2$ la concaténation des mots t_1 et t_2. Pour $n \in \mathbb{Z}$, la trace n est le mot de longueur 1 réduit à n. La figure 3 définit la relation $\xrightarrow{t}\!\!\!\!\twoheadrightarrow$ par un ensemble de règles d'inférence.

Question 3. Donner la dérivation de la sémantique de l'instruction `{ print(34) print(60-5) }` dans une mémoire vide et un environnement vide.

Correction :



(On a omis systématiquement les mémoires et les environnements, car ils sont vides.)

Déterminisme de la sémantique. À l'instar de ce que nous avons fait en cours, on peut se poser la question du déterminisme de la sémantique de Goo. Comme ce n'est pas complètement trivial, prenons quelques instants pour y réfléchir.

Question 4. Montrer que la sémantique des expressions **n'est pas** déterministe, au sens où, si $M, E, e \twoheadrightarrow M_1, v_1$ et $M, E, e \twoheadrightarrow M_2, v_2$, alors on n'a pas nécessairement $v_1 = v_2$ ni même $M_1 = M_2$.

Correction : Il suffit de considérer une expression qui implique une allocation, aussi simple que `new(S)` par exemple, car on obtient alors deux adresses fraîches qui ne sont pas nécessairement égales, et donc deux mémoire M_1 et M_2 possiblement différentes.

Question 5. Donner des arguments convaincants pour montrer que, pour autant, la sémantique de traces de Goo **est** déterministe, au sens où, si $M, E, s \xrightarrow{t_1}\!\!\!\!\twoheadrightarrow M_1$ et $M, E, s \xrightarrow{t_2}\!\!\!\!\twoheadrightarrow M_2$, alors $t_1 = t_2$. On ne demande pas une preuve formelle au dernier degré de détail, mais on demande d'introduire au moins les ingrédients clés (définitions, lemmes).

Correction : L'idée générale est que, quand bien même la sémantique n'est pas déterministe à cause de l'adresse renvoyée `new`, les mémoires et environnements d'un même calcul restent isomorphes.

On peut le formaliser ainsi. Si θ est une application des adresses vers les adresses, on note $v_1 \overset{\theta}{\approx} v_2$ la relation définie par

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

c'est-à-dire que θ envoie la valeur v_1 vers la valeur v_2. Puis on définit un isomorphisme pour les environnements,

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

un autre pour les mémoires,

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

et enfin

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

On peut alors énoncer (et prouver, même si on ne le fait pas ici) les propriétés suivantes. Pour les expressions :

> si $M_1, E_1 \overset{\theta}{\approx} M_2, E_2$, si $M_1, E_1, e \twoheadrightarrow M_1', v_1$ et si $M_2, E_2, e \twoheadrightarrow M_2', v_2$, alors il existe θ' tel que $M_1' \overset{\theta'}{\approx} M_2'$ et $v_1 \overset{\theta'}{\approx} v_2$.

Pour les instructions :

> si $M_1, E_1 \overset{\theta}{\approx} M_2, E_2$, si $M_1, E_1, s \xrightarrow{t_1}\!\!\!\!\twoheadrightarrow M_1'$ et si $M_2, E_2, s \xrightarrow{t_2}\!\!\!\!\twoheadrightarrow M_2'$, alors $t_1 = t_2$ et il existe θ' tel que $M_1' \overset{\theta'}{\approx} M_2'$.

On en déduit le résultat voulu (car trivialement $M, E \overset{\theta}{\approx} M, E$ par l'identité).

Compilation vers Goo. On souhaite se persuader que notre langage Goo n'est pas trop simple, en étudiant la possibilité d'y exécuter des programmes arbitrairement complexes. Pour cela, on considère le petit langage While dont la syntaxe abstraite est la suivante :

e	::=	n	*constante* $n \in \mathbb{Z}$	s ::=	`{` $s \ldots s$ `}` *bloc*
	|	x	*variable*	|	x `=` e *affectation*
	|	e `-` e	*soustraction*	|	`print(`x`)` *affichage*
				|	`while` x `<` n `do` s *boucle*

Un programme While est réduit à une instruction s. Toutes les variables sont globales (non déclarées explicitement) et ne contiennent que des entiers. La sémantique de ce langage devrait être claire. Pour compiler un programme While vers Goo, il suffit de définir un type `State` pour l'ensemble de ses variables, par exemple comme ceci pour un programme manipulant trois variables `a`,`b`,`c`,

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

puis de définir une fonction `run` sur cet état, comme ceci,

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

et enfin d'exécuter l'instruction `new(State).run()`.

Question 6. Donner un schéma de compilation pour le corps de la fonction `run`.

Correction : La compilation des expressions de While est immédiate :

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

Pour les instructions, on numérote les différentes boucles du programme WHILE, puis on traduit les expressions comme ceci :

```
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()
  avec
  func (s *State) while_i() { if s.x - n < 0 { C(s) s.while_i() } else {}
```

Question 7. Munir le langage WHILE d'une sémantique à grands pas analogue à celle de GOO, avec

- une relation $E, e \twoheadrightarrow n$ pour l'évaluation d'une expression e ;
- une relation $E, s \xrightarrow{t}\!\!\!\!\twoheadrightarrow E'$ pour l'évaluation d'une instruction s.

Énoncer le théorème de correction de la compilation de WHILE vers GOO. (On ne demande pas de preuve.)

Correction : Pour les expressions :

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

Pour les 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
```

Le théorème de correction peut s'écrire par exemple comme

$$\text{si } E_0, s \xrightarrow{t}\!\!\!\!\twoheadrightarrow E' \text{ alors } \emptyset, \emptyset, \texttt{new(State).run()} \xrightarrow{t}\!\!\!\!\twoheadrightarrow M'$$

où E_0 est un environnement qui envoie toutes les variables globales de s vers 0. Il n'est pas nécessaire de parler de M' dans ce théorème, car l'égalité des traces suffit à capturer tous les comportements possibles.

Analyse syntaxique. On souhaite réaliser l'analyse syntaxique du langage Goo avec l'outil Menhir. Pour les expressions, on écrit la grammaire suivante :

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

(Les actions sémantiques ne nous intéressent pas ici et sont omises, de même que les déclarations des symboles terminaux.)

Question 8. L'outil Menhir signale plusieurs conflits. Les identifier et les expliquer. Proposer une solution pour lever ces conflits, en ajoutant des indications de priorité et/ou d'associativité.

Correction : L'outil Menhir signale deux conflits lecture/réduction, correspondant respectivement à

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

Dans les deux cas, à la lecture du lexème qui suit le second `expr`, on peut soit réduire `expr MINUS expr`, soit lire.

Une solution naturelle consiste à déclarer :

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

Typage statique de Goo. Le langage Goo est muni d'un typage statique dont les règles sont données figure 4. Le jugement $\Gamma \vdash e : \tau$ signifie que l'expression e est bien typée de type τ dans le contexte Γ, et le jugement $\Gamma \vdash s$ signifie que l'instruction s est bien typée dans le contexte Γ. Comme de coutume, le contexte Γ est une fonction des variables vers les types. Dans les règles de typage, la notation $S\{y\ \tau\}$ signifie que la structure S possède un champ y de type τ.

Question 9. Ces règles de typage sont-elles dirigées par la syntaxe ? Discuter de la mise en œuvre de ces règles dans un algorithme de typage (en se limitant pour l'instant aux expressions et aux instructions, c'est-à-dire aux règles données dans la figure 4).

Correction : Oui, les règles sont bien dirigées par la syntaxe.

La seule difficulté concernant leur mise en œuvre dans un algorithme de typage est la règle de typage de `nil`, qui suppose de deviner la structure S. Deux solutions au moins :

- faire de l'inférence de typage, en donnant à `nil` une variable de type qui sera déterminée plus tard, par unification, lorsque le type d'une expression est confronté à un type attendu ;
- n'accepter `nil` qu'à des positions où le contexte permet immédiatement de déterminer son type, c'est-à-dire une affectation (on a le type du champ) ou un paramètre d'un appel (on a le type attendu pour le paramètre).

Question 10. Proposer une règle de typage pour vérifier la conformité de la définition d'une fonction.

Correction : La définition de fonction

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

est bien typée si
- la structure S existe
- les types τ_i sont bien formés
- les noms $x, x_1, \ldots, x_n$ sont deux à deux distincts
- on a $x : *S, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash s$

Question 11. Proposer un algorithme pour typer un programme. On rappelle que les structures comme les fonctions sont mutuellement récursives.

Correction : Il suffit de procéder en trois temps :

1. déclarer toutes les structures (mais en ignorant leurs champs pour l'instant) ;
2. déclarer tous les champs de structures et les profils de toutes les fonctions, en vérifiant à chaque fois que les structures mentionnées existent bien ;
3. typer tous les corps des fonctions (cf la question précédente).

Question 12. Dans le langage Goo, la sûreté du typage est toute relative, car l'évaluation peut toujours échouer dynamiquement sur le déréférencement d'un pointeur nul. Cela étant, on peut espérer une version affaiblie de la sûreté du typage, comme ceci :
- si $\Gamma \vdash e : \texttt{int}$ et si $M, E, e \twoheadrightarrow M', v$ alors la valeur v est un entier ;
- si $\Gamma \vdash e : *S$ et si $M, E, e \twoheadrightarrow M', v$ alors la valeur v est soit `nil`, soit une adresse.

Donner des conditions nécessaires sur la mémoire M, l'environnement E et le contexte Γ pour que l'on puisse espérer prouver ces deux résultats.

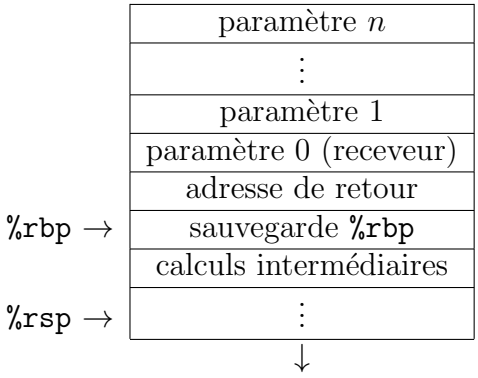
Correction : Il faut exprimer la cohérence entre la mémoire M, l'environnement E et le contexte de typage Γ, au sens où
- la valeur d'une variable de type `int` est un entier ;
- la valeur d'une variable de type $*S$ est soit `nil`, soit une adresse à laquelle se trouve une structure elle-même bien formée au regard du typage.

On peut le faire formellement de la manière suivante. On commence par définir $M \vdash v : \tau$ qui signifie que la valeur v est bien typée de type τ, comme ceci :

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

Puis on définit la relation de cohérence par ces deux conditions :
- $\mathrm{dom}(E) = \mathrm{dom}(\Gamma)$;
- pour tout $x \in \mathrm{dom}(E)$, on a $M \vdash E(x) : \Gamma(x)$.

Compilation de Goo vers x86-64. On se propose de compiler notre petit langage vers l'assembleur x86-64. (Un aide-mémoire est donné en annexe.) On fait l'hypothèse que les entiers de notre langage sont limités à des entiers 64 bits signés. Une valeur est soit un entier, soit un pointeur vers une structure allouée sur le tas (dans laquelle les valeurs des champs sont juxtaposées). La valeur `nil` est représentée par l'entier 0. On adopte un schéma de compilation simple où tous les paramètres sont passés sur la pile, y compris le receveur, et où on ne cherche pas à faire d'allocation de registres, en se servant de la pile pour stocker les calculs intermédiaires. Les tableaux d'activation ont alors la forme ci-contre.



Dans ce contexte, on note $C(e)$ la compilation d'une expression e, sous la forme d'un morceau de code assembleur qui met au final la valeur de e dans le registre `%rax`, et $C(s)$ la compilation d'une instruction s.

Question 13. Donner la définition de C dans les cas suivants :

1. une expression x ;
2. une expression $e.y$;
3. une 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  # si e-0 >= 0
    C(s1)
    jmp  L2
 L1:C(s2)
 L2:
```

Question 14. Proposer un code assembleur pour la fonction `addfib` ci-dessous en optimisant l'appel terminal.

```
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
    1:  movq  24(%rbp), %rdi  # n
        movq  16(%rbp), %rsi  # res
        cmpq  $2, %rdi
        jl    2f
        addq  $-2, %rdi
        pushq %rdi
        pushq %rsi
        call  addfib
        addq  $16, %rsp       # dépile paramètres
        addq  $-1, 24(%rbp)   # n <- n-1
        jmp   1b              # appel terminal
     2: addq  %rdi, (%rsi)
        popq  %rbp
        ret
```

Ajout d'une couche objets. Dans cette dernière partie, on ajoute une couche objets au langage Goo. Pour cela, on ajoute à notre langage la possibilité de déclarer des **interfaces** à côté des déclarations de structures et des définitions de fonctions. La syntaxe abstraite de la figure 1 est modifiée de la façon suivante :

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

Voici par exemple une interface `Rope` déclarant deux fonctions `len` et `nth` (en réutilisant la structure `R` donnée au tout début du sujet) :

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

On dit qu'une structure S **satisfait** une interface I dès lors que toutes les fonctions déclarées dans I sont implémentées pour le type de receveur $*S$. Ainsi, le type `L` des listes d'entiers donné au début du sujet satisfait l'interface `Rope` car on a défini les fonctions `len` et `nth` sur le receveur `*L` avec les profils attendus. Il n'est pas nécessaire de déclarer explicitement que `L` satisfait l'interface `Rope` ; le compilateur est capable de le vérifier tout seul si besoin. Voici un autre exemple de structure satisfaisant l'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 }
```

Comme indiqué plus haut (nouveau τ), une interface est également un type. Ainsi, on peut définir une troisième structure, `App`, dont les champs ont le type `Rope` :

```
struct App { left Rope, right Rope }
```

En particulier, on peut affecter aux champs `left` et `right` d'une structure `App` des valeurs de type `*L` ou encore `*Leaf` car ces deux structures satisfont l'interface `Rope`. Dit autrement, on a la règle de typage suivante :

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

Enfin, sur la structure `App` elle-même, on peut également implémenter les fonctions attendues par l'interface `Rope`, comme ceci :

```
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) }
}
```

En particulier, on peut donc construire des arbres binaires avec des nœuds internes de type `App` et des feuilles de type `L` ou `Leaf`, y compris en mélangeant les deux dans un même arbre.

Question 15. Pour la dernière fonction ci-dessus (la fonction `nth` sur le type `App`), donner le type statique de chaque sous-expression. Expliquer pourquoi on peut parler de langage à objets, quand bien même il n'y a ni classes ni héritage.

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
}
```

Comme on le voit par exemple ci-dessus avec `e.left.nth`, le type statique du receveur, ici `Rope`, ne permet pas de sélectionner la fonction `nth` adéquate. Il faut donc recourir à un appel dynamique de méthode.

Valeur d'interface. Pour compiler cette extension du langage Goo vers x86-64, on étend notre schéma de compilation avec un nouveau type de valeur, appelé **valeur d'interface**. Il s'agit d'un pointeur vers un bloc de deux mots alloué sur le tas. Le premier mot contient le « nom » d'une structure S (un identifiant). Le second mot contient soit `nil` (c'est-à-dire 0), soit un pointeur vers une structure (de type S) allouée sur le tas. L'idée est, comme dans tout langage objet, de se servir du nom de la structure pour retrouver, pendant l'exécution, le code de la fonction à appeler dans une table construite pendant la compilation.

Question 16. Proposer une solution pour représenter les noms de structures d'une part et les tables de fonctions d'autre part. On pourra faire l'hypothèse qu'on dispose de l'intégralité du programme source au moment de la compilation. Illustrer avec le programme ci-dessus, c'est-à-dire avec les structures `L`, `Leaf` et `App` et l'interface `Rope`.

Correction : Il suffit de numéroter les structures par des entiers consécutifs $0, 1, \ldots$, de manière arbitraire. Le premier mot d'une valeur d'interface est alors cet entier.

Les tables de fonctions sont alors allouées dans le segment de données, en suivant l'ordre de cette numérotation des structures. Ainsi, avec `L`, `Leaf` et `App` numérotées respectivement 0, 1 et 2, on aura :

```
.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. Expliquer à quels endroits du programme les valeurs d'interface sont construites par le compilateur, et comment.

Correction : Invariant : une expression s'évalue vers une valeur d'interface si et seulement si son type statique est une interface.

Dès lors, on construit une valeur d'interface exactement aux endroits où une valeur de type $*S$ est convertie (par la règle de typage ci-dessus) vers un type d'interface. Cela se produit à deux endroits dans le code :

- une affectation ;
- un passage de paramètre dans un appel.

Une façon élégante de l'implémenter est d'insérer une coercion explicite dans l'AST pendant le typage. Le type statique de l'argument de cette coercion est exactement le type qu'il faut stocker dans la première composante de la valeur d'interface.

Question 18. Donner le schéma de compilation pour un appel $e.f(e_1, \ldots, e_n)$. On distinguera deux cas, selon que le type statique de e est $*S$ ou I.

Correction :

- pour e de type $*S$: on appelle directement la fonction f pour le receveur $*S$

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

- pour e de type I : on sait par invariant que la valeur de e est une valeur d'interface

```
C(en) pushq %rax
...
C(e1) pushq %rax
C(e)  pushq 8(%rax)     # on empile le receveur, pas la valeur d'interface
movq  (%rax), %rcx            # numéro de la structure
movq  table_f(,%rcx,8), %rcx  # code récupéré dans la table de f
call  *%rcx                   # appel dynamique
addq  $8n+8, %rsp
```

Question 19. Toujours dans le contexte des déclarations ci-dessus, que vous inspire le programme suivant ?

```
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) }
```

Doit-il être rejeté au typage, et si oui pourquoi ? Sinon, peut-on espérer l'exécuter avec succès, et dans ce cas comment ?

Correction : Le programme ci-dessus est bien typé, en particulier car

- la valeur `nil` est acceptée comme second paramètre de `q1`, car $\vdash$ `nil` : $*L$;
- les deux affectations de `a.left` et `a.right` sont bien typées, car `l` est de type `*L` et `L` satisfait `Rope`.

Il s'exécute avec succès car, lors des deux affectations, on construit une valeur d'interface avec la structure `L` et la valeur `nil`. Ensuite, lors des deux appels à `len` cachés dans l'appel `a.len(out)`, l'appel dynamique va alors sélectionner l'implémentation de `len` sur le type `*L`, qui est bien définie sur le receveur `nil` (cf la toute première page du sujet).

(Le langage Goo est directement inspiré du langage Go.)

$$
\begin{array}{rcll}
e & ::= & n & \textit{constante } n \in \mathbb{Z} \\
& | & \texttt{nil} & \textit{pointeur nul} \\
& | & x & \textit{variable} \\
& | & e \texttt{ - } e & \textit{soustraction} \\
& | & e\texttt{.}y & \textit{accès au champ} \\
& | & \texttt{new(}S\texttt{)} & \textit{allocation} \\
\\
s & ::= & e\texttt{.}y \texttt{ = } e & \textit{affectation} \\
& | & e\texttt{.}f\texttt{(}e, \ldots, e\texttt{)} & \textit{appel de fonction} \\
& | & \texttt{print(}e\texttt{)} & \textit{affichage} \\
& | & \texttt{if } e \texttt{ = nil then } s \texttt{ else } s & \textit{conditionnelle} \\
& | & \texttt{if } e \texttt{ < 0 then } s \texttt{ else } s & \textit{conditionnelle} \\
& | & \texttt{\{}\, s \ldots s \,\texttt{\}} & \textit{bloc} \\
\\
\tau & ::= & \texttt{int} \mid \texttt{*}S & \textit{type} \\
D & ::= & \texttt{struct } S \texttt{ \{}\, y\ \tau, \ldots, y\ \tau \,\texttt{\}} & \textit{déclaration de structure} \\
F & ::= & \texttt{func (}x\ \texttt{*}S\texttt{) } f\texttt{(}x\ \tau, \ldots, x\ \tau\texttt{) } s & \textit{définition de fonction} \\
p & ::= & D \ldots D\ F \ldots F & \textit{programme}
\end{array}
$$

FIGURE 1 – Syntaxe abstraite de GOO.

$$
\begin{array}{rcll}
v & ::= & n & \textit{valeur entière } n \in \mathbb{Z} \\
& | & \texttt{nil} & \textit{pointeur nul} \\
& | & \ell & \textit{adresse mémoire}
\end{array}
$$

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

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

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

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

FIGURE 2 – Sémantique opérationnelle de GOO (expressions).

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

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

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

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

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

FIGURE 3 – Sémantique opérationnelle de GOO (instructions).

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

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

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

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

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

FIGURE 4 – Typage de GOO.

Annexe : aide-mémoire x86-64

On donne ici un fragment du jeu d'instructions x86-64. Vous êtes libre d'utiliser tout autre élément de l'assembleur x86-64. Dans ce qui suit, r_i désigne un registre, n une constante entière et L une étiquette.

`mov` r_2`,` r_1	copie le registre r_2 dans le registre r_1
`mov` \$$n$`,` r_1	charge la constante n dans le registre r_1
`mov` L`,` r_1	charge la valeur à l'adresse L dans le registre r_1
`mov` \$$L$`,` r_1	charge l'adresse de l'étiquette L dans le registre r_1
`sub` r_2`,` r_1	calcule $r_1 - r_2$ dans r_1
`mov` n`(`r_2`),` r_1	charge dans r_1 la valeur contenue en mémoire à l'adresse $r_2 + n$
`mov` r_1`,` n`(`r_2`)`	écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans r_1
`push` r_1	empile la valeur contenue dans r_1
`pop` r_1	dépile une valeur dans le registre r_1
`cmp` r_2`,` r_1	positionne les drapeaux en fonction de la valeur de $r_1 - r_2$
`test` r_2`,` r_1	positionne les drapeaux en fonction de la valeur de $r_1 \& r_2$
`je` L	saute à l'adresse désignée par l'étiquette L en cas d'égalité (on a de même `jne`, `jg`, `jge`, `jl` et `jle`)
`jmp` L	saute à l'adresse désignée par l'étiquette L
`call` L	saute à l'adresse désignée par l'étiquette L, après avoir empilé l'adresse de retour
`ret`	dépile une adresse et y effectue un saut

Les registres `rsp`, `rbp`, `rbx`, `r12`, `r13`, `r14` et `r15` sont *callee-saved* ; les autres sont *caller-saved*.

On alloue de la mémoire sur le tas avec un appel à `malloc`, qui attend un nombre d'octets dans `%rdi` et renvoie l'adresse du bloc alloué dans `%rax`.