

École Normale Supérieure  
Langages de programmation et compilation  
examen 2023–2024

Jean-Christophe Filliâtre

26 janvier 2024 — 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 2–3 sont regroupées en fin de sujet, page 7. Suggestion : détacher la dernière feuille.

Dans tout ce sujet, on considère un petit fragment du langage Python dont la syntaxe abstraite est donnée figure 2. (Connaître le langage Python n'est pas nécessaire.) Ce fragment inclut la valeur `None`, des booléens (`False` et `True`), des entiers et des “listes”. Les expressions sont limitées à des constantes ( $c$ ), des variables ( $x$ ), des opérations primitives ( $op$ ), des appels de fonctions ( $f$ ) et une expression conditionnelle. Cette dernière, notée  $e_1$  `if`  $e_2$  `else`  $e_3$ , évalue  $e_2$  puis renvoie la valeur de  $e_1$  si  $e_2$  est vrai et la valeur de  $e_3$  sinon. Un programme est une suite de définitions de fonctions ( $d$ ), suivie de l'évaluation d'une unique expression dont la valeur est affichée avec `print`. Le corps d'une fonction est une séquence d'affectations  $x = e$ , suivie d'une instruction `return`. Chaque fonction peut faire référence aux fonctions précédemment définies ou à la fonction en cours de définition (fonction récursive). Voici un exemple de programme dans ce fragment, écrit dans la syntaxe concrète de Python, qui calcule une liste de cinq entiers et l'affiche :

```
def aux(s):
    a = s[0]
    b = s[1]
    return [a+b] + s
def myst(n, s):
    return s if n==0 else myst(n-1, aux(s))
print(myst(3, [1]+[0]))
```

Une sémantique opérationnelle à grands pas est donnée figure 3. Elle traduit notamment un passage par valeur. Une valeur est notée  $v$ . La valeur des variables est donnée par un environnement  $V$ , c'est-à-dire une fonction qui associe des valeurs à des noms de variables. La sémantique inclut la définition des opérations primitives (dans le tableau en bas de la figure). Pour chaque opération  $op$ , sa sémantique est donnée par une fonction  $\llbracket op \rrbracket$  opérant sur des valeurs. Cette fonction peut être partielle, *i.e.* ne pas être définie sur toutes les valeurs. Ainsi, l'expression `1+None` n'a pas de valeur. L'opération *num* permet d'interpréter un booléen comme un entier dans certaines opérations.

**Question 1** Quelle est la liste calculée et affichée par le programme ci-dessus ? (On ne demande pas de justifier.)

**Question 2** Écrire une fonction `rev` qui, lorsqu'elle est appliquée à une liste  $[v_0, v_1, \dots, v_{n-1}]$ , renvoie la liste  $[v_{n-1}, \dots, v_1, v_0]$ , avec  $n \geq 0$ . Cette fonction doit pouvoir fonctionner avec  $n = 0$ , quand bien même il n'est pas possible de construire une liste vide dans notre fragment. On pourra introduire une fonction auxiliaire.

**Question 3** Pour chacune des expressions suivantes, donner la dérivation de son évaluation dans un environnement vide, lorsqu'elle existe, ou justifier qu'il n'y en a pas. (On se place ici dans le contexte des deux fonctions `aux` et `myst` données en exemple au début du sujet.)

- `myst(0, [1]+[0])`
- `aux([0])`
- `myst(-1, [1]+[0])`

**Question 4** On se propose de programmer en OCaml un interprète de notre langage, en suivant la sémantique à grands pas de la figure 3. Indiquer précisément les types OCaml et les structures de données utilisés. Donner le type de chaque fonction OCaml impliquée dans l'interprète. *On ne demande pas d'écrire le code de ces fonctions.* On rappelle qu'on obtient en OCaml des dictionnaires dont les clés sont des chaînes de caractères avec `module StrMap = Map.Make(String)`.

**Typage statique.** Bien que le langage Python soit un langage typé dynamiquement, on se propose ici de réaliser un peu de typage statique sur notre langage, avec le double objectif de rejeter des programmes incohérents et d'exécuter plus efficacement certains programmes. On se donne quatre sortes `none`, `bool`, `int` et `list`, pour représenter respectivement une valeur `None`, une valeur booléenne, une valeur entière ou une liste. Un type  $\tau$  est alors un ensemble de sortes, c'est-à-dire

$$\tau \subseteq \{\text{none}, \text{bool}, \text{int}, \text{list}\},$$

avec l'interprétation suivante : si une expression de type  $\tau$  s'évalue en une valeur, alors cette valeur sera nécessairement d'une des sortes de  $\tau$ . En particulier, le type peut être l'ensemble vide  $\emptyset$  (l'expression ne peut pas avoir de valeur) ou l'ensemble  $\{\text{none}, \text{bool}, \text{int}, \text{list}\}$  (la valeur peut être quelconque).

Pour typer une expression, on se donne un contexte formé de deux environnements : un environnement  $\Gamma$  donnant le type des variables (une fonction des variables vers les types) et un environnement  $\Delta$  donnant les types des fonctions. (Les opérations primitives sont vues comme des fonctions pour le typage.) Le type d'une fonction, noté  $\sigma$ , est de la forme

$$\tau_0 \times \cdots \times \tau_{n-1} \rightarrow \tau$$

où  $n$  est le nombre de paramètres de la fonction. L'environnement  $\Delta$  est un ensemble de paires  $(f, \sigma)$  où  $f$  est le nom d'une fonction et  $\sigma$  un type de fonction. Pour une même fonction, il peut y avoir plusieurs types différents dans l'environnement  $\Delta$ . La figure 1 donne des règles de typage pour les expressions, ainsi qu'un environnement  $\Delta_{op}$  donnant les types des opérations primitives. On note que l'opération `add` a deux types différents dans  $\Delta_{op}$ , ce qui est cohérent avec ses deux interprétations dans la figure 3.

Pour typer une application  $f(e_0, \dots, e_{n-1})$ , on fait l'union de tous les types que l'on peut obtenir avec les types de  $f$  donnés par  $\Delta$  et compatibles avec les types  $\tau_i$  des paramètres effectifs  $e_i$ . En particulier, le résultat peut être le type  $\emptyset$  si  $\Delta$  ne contient aucun type de fonction compatible.

**Question 5** Dans un environnement  $\Gamma$  vide et un environnement  $\Delta$  qui ne contient que les opérations primitives, donner

1. une expression qui a le type  $\emptyset$ ;
2. une expression qui a le type  $\{\text{bool}, \text{int}, \text{list}\}$ .

$$\begin{array}{c}
\frac{}{\Delta, \Gamma \vdash \text{None} : \text{none}} \quad \frac{}{\Delta, \Gamma \vdash b : \text{bool}} \quad \frac{}{\Delta, \Gamma \vdash n : \text{int}} \quad \frac{x \in \text{dom}(\Gamma)}{\Delta, \Gamma \vdash x : \Gamma(x)} \\
\frac{\Delta, \Gamma \vdash e_1 : \tau_1 \quad \Delta, \Gamma \vdash e_2 : \tau_2 \quad \Delta, \Gamma \vdash e_3 : \tau_3}{\Delta, \Gamma \vdash e_1 \text{ if } e_2 \text{ else } e_3 : \tau_1 \cup \tau_3} \\
\frac{\forall 0 \leq i < n, \Delta, \Gamma \vdash e_i : \tau_i}{\Delta, \Gamma \vdash f(e_0, \dots, e_{n-1}) : \Delta(f)(\tau_0, \dots, \tau_{n-1})} \quad \text{avec } \Delta(f)(\tau_0, \dots, \tau_{n-1}) \stackrel{\text{def}}{=} \bigcup_{\substack{f : \tau'_0 \times \dots \times \tau'_{n-1} \rightarrow \tau \in \Delta \\ \forall 0 \leq i < n, \tau_i \cap \tau'_i \neq \emptyset}} \tau
\end{array}$$

Environnement  $\Delta_{op}$  :

opérateur	type
<i>add</i>	$\{\text{list}\} \times \{\text{list}\} \rightarrow \{\text{list}\}$
<i>add</i>	$\{\text{bool}, \text{int}\} \times \{\text{bool}, \text{int}\} \rightarrow \{\text{int}\}$
<i>sub</i>	$\{\text{bool}, \text{int}\} \times \{\text{bool}, \text{int}\} \rightarrow \{\text{int}\}$
<i>len</i>	$\{\text{list}\} \rightarrow \{\text{int}\}$
<i>mk</i>	$\{\text{none}, \text{bool}, \text{int}, \text{list}\} \rightarrow \{\text{list}\}$
<i>get</i>	$\{\text{list}\} \times \{\text{bool}, \text{int}\} \rightarrow \{\text{none}, \text{bool}, \text{int}, \text{list}\}$
<i>eq</i>	$\{\text{none}, \text{bool}, \text{int}, \text{list}\} \times \{\text{none}, \text{bool}, \text{int}, \text{list}\} \rightarrow \{\text{bool}\}$

FIGURE 1 – Typage statique d'une expression.

**Question 6** Pour cette question, on se donne l'environnement suivant :

$$\begin{aligned}
\Delta &\stackrel{\text{def}}{=} \Delta_{op} \cup \{(f, \{\text{int}\} \times \{\text{list}\} \rightarrow \{\text{list}\}), (f, \{\text{int}\} \times \{\text{none}\} \rightarrow \{\text{none}\})\} \\
\Gamma &\stackrel{\text{def}}{=} \{x \mapsto \{\text{list}\}\}
\end{aligned}$$

Pour chaque expression suivante, donner sa dérivation de typage dans  $\Delta, \Gamma$ .

1.  $f(1, x)$
2.  $f(1, x[0])$
3.  $f(\text{None}, \text{None})$

**Question 7** À quelle(s) condition(s) nécessaires et suffisantes sur l'environnement  $\Delta, \Gamma$  et sur l'expression  $e$  existe-t-il un type  $\tau$  tel que  $\Delta, \Gamma \vdash e : \tau$  ?

**Question 8** Montrer que, pour un environnement  $\Delta, \Gamma$  et une expression  $e$ , il existe au plus un type  $\tau$  tel que  $\Delta, \Gamma \vdash e : \tau$ .

**Question 9** Au regard du typage, y a-t-il une différence entre les deux environnements suivants ?

$$\begin{aligned}
\Delta &= \{(f, \{\text{bool}\} \times \{\text{bool}\} \rightarrow \{\text{int}\}); (f, \{\text{bool}\} \times \{\text{int}\} \rightarrow \{\text{int}\})\} \\
\text{et } \Delta' &= \{(f, \{\text{bool}\} \times \{\text{bool}, \text{int}\} \rightarrow \{\text{int}\})\}
\end{aligned}$$

Même question avec les deux environnements suivants :

$$\begin{aligned}
\Delta &= \{(f, \{\text{bool}\} \times \{\text{bool}\} \rightarrow \{\text{int}\}); (f, \{\text{bool}\} \times \{\text{int}\} \rightarrow \{\text{list}\})\} \\
\text{et } \Delta' &= \{(f, \{\text{bool}\} \times \{\text{bool}, \text{int}\} \rightarrow \{\text{int}, \text{list}\})\}
\end{aligned}$$

**Typage d'une fonction.** Pour typer une définition de fonction, on introduit le jugement  $\Delta \vdash f : \sigma$  qui signifie « dans l'environnement  $\Delta$ , la définition de la fonction  $f$  admet le type de fonction  $\sigma$  ». On propose la règle suivante pour ce jugement :

$$\frac{\begin{array}{c} f(x_0, \dots, x_{n-1}) \stackrel{\text{def}}{=} x_n = e_n; \dots; x_{m-1} = e_{m-1}; \text{return } e \\ \Gamma_n \stackrel{\text{def}}{=} \{x_0 \mapsto \tau_0; \dots; x_{n-1} \mapsto \tau_{n-1}\} \\ \forall n \leq i < m, \Delta, \Gamma_i \vdash e_i : \tau_i \quad \Gamma_{i+1} \stackrel{\text{def}}{=} \Gamma_i[x_i \mapsto \tau_i] \\ \Delta, \Gamma_m \vdash e : \tau \end{array}}{\Delta \vdash f : \tau_0 \times \dots \times \tau_{n-1} \rightarrow \tau} \quad (1)$$

(On notera sa ressemblance avec la règle de sémantique.) Cette règle permet notamment de typer une définition récursive, en montrant  $\Delta \vdash f : \sigma$  pour un environnement  $\Delta$  contenant un ou plusieurs types pour  $f$ .

**Question 10** Pour la fonction `aux` du début du sujet, montrer qu'on a

$$\Delta_{op} \vdash \text{aux} : \{\text{list}\} \rightarrow \{\text{list}\}.$$

Donner la dérivation de typage complète.

**Question 11** Proposer au moins deux types  $\sigma$  différents tels que, pour chacun, on ait

$$\Delta_{op} \cup \{(\text{aux} : \{\text{list}\} \rightarrow \{\text{list}\}); (\text{myst}, \sigma)\} \vdash \text{myst} : \sigma$$

pour la fonction `myst` du début du sujet. (On ne demande pas de justification, c'est-à-dire qu'on ne demande pas les dérivations de typage, mais seulement les deux types  $\sigma$ .)

**Question 12** Proposer un type pour la fonction

```
def loop(x):
    return loop(x)
```

qui soit le plus informatif possible.

**Question 13** On souhaiterait montrer la sûreté de notre typage statique dans le sens suivant : si  $V, e \rightsquigarrow v$  et  $\Delta, \Gamma \vdash e : \tau$ , alors  $T(v) \in \tau$  où  $T$  est la fonction donnant le type d'une valeur sémantique, définie sans surprise comme

$$\begin{aligned} T(\text{None}) &= \text{none} \\ T(b) &= \text{bool} \\ T(n) &= \text{int} \\ T([v_0, \dots, v_{n-1}]) &= \text{list} \end{aligned}$$

Donner des conditions nécessaires sur  $V$  et  $\Delta, \Gamma$  pour que la sûreté du typage soit possible. *Mais on ne demande pas de montrer la sûreté du typage.*

**Question 14** Notre règle de typage d'une fonction n'est pas algorithmique : elle ne permet que de vérifier la définition d'une fonction  $f$  au regard de  $\Delta$ , pas de trouver un type pour  $f$ . Proposer un algorithme pour inférer un ensemble de types pour une fonction dont on a la définition.

**Analyse syntaxique.** On souhaite réaliser l'analyse syntaxique de notre petit langage. Une grammaire pour un sous-ensemble des expressions est la suivante :

$$\begin{aligned}
 E & ::= \text{id} \\
 & \quad | E \text{ if } E \text{ else } E \\
 & \quad | E [ E ]
 \end{aligned}$$

**Question 15** Montrer que cette grammaire est ambiguë.

**Question 16** On écrit cette grammaire dans la syntaxe de l'outil Menhir, de la manière suivante :

```

expr :
| IDENT                { ... }
| expr IF expr ELSE expr { ... }
| expr LBR expr RBR    { ... }

```

(Les actions sémantiques ne nous intéressent pas ici et sont omises, de même que les déclarations des symboles terminaux.) L'outil Menhir signale deux conflits. De quels types de conflits s'agit-il ? Proposer une solution pour lever ces conflits, en ajoutant des indications de priorité et/ou d'associativité.

**Question 17** On propose une autre grammaire pour ce langage :

$$\begin{aligned}
 E & ::= A \\
 & \quad | A \text{ if } E \text{ else } E \\
 A & ::= \text{ident} \\
 & \quad | A [ E ]
 \end{aligned}$$

Montrer que cette grammaire est SLR(1).

**Compilation vers l'assembleur 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.

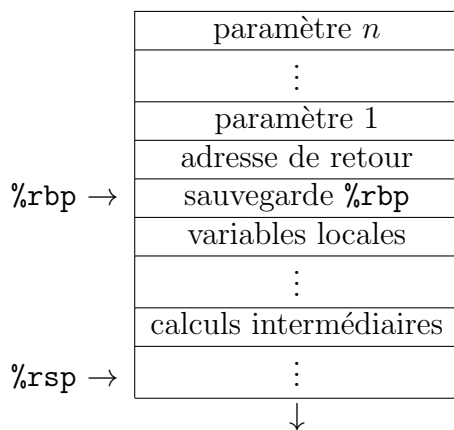
On adopte une représentation où toute valeur est l'adresse d'un bloc mémoire alloué sur le tas, dont la taille est un multiple de 64 bits, avec la forme suivante :

None	0	0				
booléen False	1	0				
booléen True	1	1				
entier $n$	2	$n$				
liste	3	$n$	$v_0$	$v_1$	$\dots$	$v_{n-1}$

Le premier mot est un entier qui indique le type de la valeur.

**Question 18** Discuter ce choix de représentation. En particulier, pourrait-on représenter None par le pointeur nul ? Si oui, y aurait-il un intérêt à le faire ? Et pourrait-on représenter un entier directement par sa valeur ?

**Schéma de compilation.** On adopte un schéma de compilation simple où tous les paramètres sont passés sur la pile 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 suivante :



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  $\%rax$ .

**Question 19** Donner la définition de  $C$  dans les trois cas suivants :

1. l'expression est une variable  $x$  ;
2. l'expression est de la forme  $[e]$  ;
3. l'expression est de la forme  $e_1$  **if**  $e_2$  **else**  $e_3$ .

**Question 20** Expliquer comment le typage statique réalisé dans les questions précédentes peut permettre, localement, de produire un code assembleur plus efficace. Donner des exemples.

$e ::= c$	<i>constante</i>	$c ::= \text{None}$	
$x$	<i>variable</i>	$b$	$b \in \{\text{True}, \text{False}\}$
$op(e, \dots, e)$	<i>opération</i>	$n$	$n \in \mathbb{Z}$
$f(e, \dots, e)$	<i>application</i>		
$e \text{ if } e \text{ else } e$	<i>conditionnelle</i>		
		$d ::= f(x, \dots, x) \stackrel{\text{def}}{=} x = e; \dots; x = e; \text{return } e$	
		$p ::= d \dots d \text{ print}(e)$	

FIGURE 2 – Syntaxe abstraite.

valeur	$v ::= c$	$num(\text{False}) \stackrel{\text{def}}{=} 0$
	$[v, \dots, v]$	$num(\text{True}) \stackrel{\text{def}}{=} 1$
environnement	$V ::= x \mapsto v$	$num(n) \stackrel{\text{def}}{=} n$
$\frac{}{V, c \rightarrow c}$	$\frac{x \in \text{dom}(V)}{V, x \rightarrow V(x)}$	$\frac{V, e_2 \rightarrow v_2 \quad v_2 \notin \{\text{None}, \text{False}, 0, []\} \quad V, e_1 \rightarrow v_1}{V, e_1 \text{ if } e_2 \text{ else } e_3 \rightarrow v_1}$
	$\frac{V, e_2 \rightarrow v_2 \quad v_2 \in \{\text{None}, \text{False}, 0, []\} \quad V, e_3 \rightarrow v_3}{V, e_1 \text{ if } e_2 \text{ else } e_3 \rightarrow v_3}$	
	$\frac{V, e_i \rightarrow v_i \quad \llbracket op \rrbracket(v_0, \dots, v_{n-1}) = v}{V, op(e_0, \dots, e_{n-1}) \rightarrow v}$	
	$\frac{\forall 0 \leq i < n, V, e_i \rightarrow v_i \quad f(x_0, \dots, x_{n-1}) \stackrel{\text{def}}{=} x_n = e_n; \dots; x_{m-1} = e_{m-1}; \text{return } e \quad V_n \stackrel{\text{def}}{=} \{x_0 \mapsto v_0; \dots; x_{n-1} \mapsto v_{n-1}\} \quad \forall n \leq i < m, V_i, e_i \rightarrow v_i \quad V_{i+1} \stackrel{\text{def}}{=} V_i[x_i \mapsto v_i] \quad V_m, e \rightarrow v}{V, f(e_0, \dots, e_{n-1}) \rightarrow v}$	

syntaxe concrète	<i>op</i>	sémantique $\llbracket op \rrbracket$
$e + e$	<i>add</i>	$\llbracket add \rrbracket([v_0, \dots, v_{n-1}], [v'_0, \dots, v'_{m-1}]) \stackrel{\text{def}}{=} [v_0, \dots, v_{n-1}, v'_0, \dots, v'_{m-1}]$ $\llbracket add \rrbracket(v_0, v_1) \stackrel{\text{def}}{=} num(v_0) + num(v_1)$ , sinon
$e - e$	<i>sub</i>	$\llbracket sub \rrbracket(v_0, v_1) \stackrel{\text{def}}{=} num(v_0) - num(v_1)$
$\text{len}(e)$	<i>len</i>	$\llbracket len \rrbracket([v_0, \dots, v_{n-1}]) \stackrel{\text{def}}{=} n$
$[e]$	<i>mk</i>	$\llbracket mk \rrbracket(v) \stackrel{\text{def}}{=} [v]$
$e[e]$	<i>get</i>	$\llbracket get \rrbracket([v_0, \dots, v_{n-1}], i) \stackrel{\text{def}}{=} v_{num(i)}$ si $0 \leq num(i) < n$
$e == e$	<i>eq</i>	$\llbracket eq \rrbracket(v_0, v_1) \stackrel{\text{def}}{=} \text{True}$ si $v_0 = v_1$ ou $num(v_0) = num(v_1)$ $\text{False}$ sinon

FIGURE 3 – Sémantique opérationnelle à grands pas.

## 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.

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

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`.