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 14. 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 $[\![op]\!]$ 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.)

Correction : Ce programme calcule les nombres de Fibonacci et affiche la liste suivante :

```
[3, 2, 1, 1, 0]
```

Question 2 Écrire une fonction `rev` qui, lorsqu'elle est appliquée à une liste $[v_0, v_1, \ldots, v_{n-1}]$, renvoie la liste $[v_{n-1}, \ldots, v_1, v_0]$, 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.

Correction : On introduit une fonction auxiliaire, avec un indice `i` et un accumulateur `acc`. On traite le cas $n = 0$ de façon particulière, car on ne sait pas construire de liste vide avec les opérateurs qui nous sont donnés.

```
def revaux(s, i, acc):
    return acc if i==len(s) else revaux(s, i+1, [s[i]] + acc)
def rev(s):
    return s if len(s)==0 else revaux(s, 1, [s[0]])
```

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])`

Correction :

- `myst(0, [1]+[0])`
  Notons `bmyst` le corps de la fonction `myst`.

```
          1-->1     0-->0
        --------- ---------    -------------   -----------
        [1]-->[1] [0]-->[0]    V,n==0-->True   V,s-->[1,0]
        -------------------  -------------------------------
 0-->0   [1]+[0]-->[1,0]      V={n:0,s:[1,0]},bmyst-->[1,0]
 ------------------------------------------------------------
              myst(0,[1]+[0])  --> [1,0]
```

- `aux([0])`
  On a `[0]` $\twoheadrightarrow [0]$. Soit $V := \{s \mapsto [0]\}$. On a $V, s\texttt{[0]} \twoheadrightarrow 0$. Puis, on cherche à évaluer $s\texttt{[1]}$ dans $V' = \{s \mapsto [0]; a \mapsto 0\}$. Mais $[\![get]\!]$ n'est définie que pour un accès dans les bornes, ce qui n'est pas le cas ici. Donc $s\texttt{[1]}$ n'a pas de valeur dans V' et donc toute l'expression n'a pas de valeur.
- `myst(-1, [1]+[0])` n'a pas de valeur car l'évaluation ne termine pas, ce que le sémantique à grands pas échoue à capturer. Plus précisément, montrons que si $e_1 \twoheadrightarrow n$ et $e_2 \twoheadrightarrow v$, et si $\texttt{myst}(e_1, e_2) \twoheadrightarrow v'$, alors $n \geq 0$, par induction sur la dérivation de `myst`.
  — si $n = 0$, c'est immédiat.
  — sinon, la dérivation a la forme suivante

```
                                                  ...
                                          ----------------------------
                              V,n=0->False    V,myst(n-1, aux(s))-->v'
                              ----------------------------------------
        e1-->n   e2-->v         V={n->n,s->v},bmyst --> v'
        ------------------------------------------------
```

`myst(e1, e2) --> v'`

et par hypothèse de récurrence appliquée à la sous-dérivation de $\texttt{myst}(n-1, \texttt{aux}(s))$, on a $n - 1 \geq 0$ et donc $n > 0$.

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)`.

Correction : On peut se donner les types suivants pour la syntaxe abstraite

```
type expr =
  | Enone
  | Ebool of bool
  | Eint of int
  | Evar of string
  | Eapp of string * expr list
  | Eite of expr * expr * expr
type stmt = string * expr
type def = string * string list * stmt list * expr
type program = def list * expr
```

et le type suivant pour les valeurs calculées par l'interprète :

```
type value =
  | Vnone
  | Vbool of bool
  | Vint of int
  | Vlist of value array
```

Les fonctions peuvent être stockées dans une table de hachage globale :

```
let funs : (string, string list * stmt list * expr) H.t = ...
```

L'environnement peut être représentée en revanche par un dictionnaire immuable :

```
module M = Map.Make(String)
type env = value M.t
```

Pour l'interprète, on peut se donner par exemple les trois fonctions suivantes :

```
val num: value -> int
val expr: env -> expr -> value
val program: program -> unit (* évalue et imprime la valeur *)
```

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

$$\frac{}{\Delta, \Gamma \vdash \texttt{None} : \texttt{none}} \qquad \frac{}{\Delta, \Gamma \vdash b : \texttt{bool}} \qquad \frac{}{\Delta, \Gamma \vdash n : \texttt{int}} \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Delta, \Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \tau_1 \quad \Delta, \Gamma \vdash e_2 : \tau_2 \quad \Delta, \Gamma \vdash e_3 : \tau_3}{\Delta, \Gamma \vdash e_1 \texttt{ if } e_2 \texttt{ else } e_3 : \tau_1 \cup \tau_3}$$

$$\frac{\forall\ 0 \leq i < n,\ \Delta, \Gamma \vdash e_i : \tau_i}{\Delta, \Gamma \vdash f(e_0, \ldots, e_{n-1}) : \Delta(f)(\tau_0, \ldots, \tau_{n-1})} \quad \text{avec } \Delta(f)(\tau_0, \ldots, \tau_{n-1}) \stackrel{\text{def}}{=} \bigcup_{\substack{f : \tau'_0 \times \cdots \tau'_{n-1} \to \tau \in \Delta \\ \forall 0 \leq i < n,\ \tau_i \cap \tau'_i \neq \emptyset}} \tau$$

Environnement Δ_{op} :

opérateur	type
add	$\{\texttt{list}\} \times \{\texttt{list}\} \to \{\texttt{list}\}$
add	$\{\texttt{bool}, \texttt{int}\} \times \{\texttt{bool}, \texttt{int}\} \to \{\texttt{int}\}$
sub	$\{\texttt{bool}, \texttt{int}\} \times \{\texttt{bool}, \texttt{int}\} \to \{\texttt{int}\}$
len	$\{\texttt{list}\} \to \{\texttt{int}\}$
mk	$\{\texttt{none}, \texttt{bool}, \texttt{int}, \texttt{list}\} \to \{\texttt{list}\}$
get	$\{\texttt{list}\} \times \{\texttt{bool}, \texttt{int}\} \to \{\texttt{none}, \texttt{bool}, \texttt{int}, \texttt{list}\}$
eq	$\{\texttt{none}, \texttt{bool}, \texttt{int}, \texttt{list}\} \times \{\texttt{none}, \texttt{bool}, \texttt{int}, \texttt{list}\} \to \{\texttt{bool}\}$

FIGURE 1 – Typage statique d'une expression.

`none`, `bool`, `int` et `list`, pour représenter respectivement une valeur `None`, une valeur booléenne, une valeur entière ou une liste. Un type τ est alors un ensemble de sortes, c'est-à-dire

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

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

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

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

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

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

Question 5 Dans un environnement Γ vide et un environnement Δ qui ne contient que les opérations primitives, donner

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

Correction :

1. `1+None`
2. `True if True else 1 if True else [1]`

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

$$\Delta \stackrel{\text{def}}{=} \Delta_{op} \cup \{(f, \{\texttt{int}\} \times \{\texttt{list}\} \to \{\texttt{list}\}), (f, \{\texttt{int}\} \times \{\texttt{none}\} \to \{\texttt{none}\})\}$$
$$\Gamma \stackrel{\text{def}}{=} \{x \mapsto \{\texttt{list}\}\}$$

Pour chaque expression suivante, donner sa dérivation de typage dans Δ, Γ.

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

Correction :

1.
$$\frac{1 : \{\texttt{int}\} \quad x : \{\texttt{list}\}}{f(1, x) : \{\texttt{list}\}}$$

2.
$$\frac{1 : \{\texttt{int}\} \quad \frac{}{x\texttt{[0]} : \{\texttt{none},\texttt{bool},\texttt{int},\texttt{list}\}}}{f(1, x\texttt{[0]}) : \{\texttt{none}, \texttt{list}\}}$$

3.
$$\frac{\texttt{None} : \{\texttt{none}\} \quad \texttt{None} : \{\texttt{none}\}}{f(\texttt{None}, \texttt{None}) : \emptyset}$$

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

Correction : Montrons par récurrence sur e que e admet un type dans Δ, Γ si et seulement si toutes les variables de e sont définies dans Γ.

- si e est une constante, c'est immédiat ;
- si e est une variable, la condition est justement $x \in \text{dom}(\Gamma)$;
- si $e = e_1$ `if` e_2 `else` e_3, alors par HR les trois sous-expressions ont un type si et seulement si leurs variables sont définies dans Γ, et donc e admet un type si et seulement si toutes ses variables sont définies dans Γ ;
- si $e = f(e_1, \ldots, e_n)$, alors par HR toutes les sous-expressions e_i ont un type si et seulement si leurs variables sont définies dans Γ, et donc e admet un type si et seulement si toutes ses variables sont définies dans Γ (car l'union est toujours définie, quand bien même f n'est pas dans Δ ou n'y est pas avec un type compatible).

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

Correction : Supposons $\Delta, \Gamma \vdash e : \tau$ et $\Delta, \Gamma \vdash e : \tau'$, et montrons $\tau = \tau'$ par récurrence structurelle sur la dérivation de typage.

- si e est une constante, c'est immédiat ;
- si e est une variable, on a $\tau = \tau' = \Gamma(x)$;
- si $e = f(e_1, \ldots, e_n)$, alors par HR les trois types pour les trois sous-expressions sont les mêmes, d'où le résultat ;
- si $e = f(e_1, \ldots, e_n)$, alors par HR toutes les sous-expressions e_i ont les mêmes types dans les deux dérivations, et l'union calculée donne alors le même type pour e.

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

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

Même question avec les deux environnements suivants :

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

Correction :

1. Dans le premier cas, il n'y a pas de différence. En effet, si les types des deux arguments de f intersecte $\{\texttt{bool}\}$ et, soit $\{\texttt{bool}\}$, soit $\{\texttt{int}\}$, alors on aura $\{\texttt{int}\}$ pour l'application, et sinon $\emptyset$, dans les deux cas.
2. Dans le second cas, en revanche, il y a une différence, car on pourrait alors obtenir des types moins précis. Exemple : $f(\texttt{False}, \texttt{False})$ se voit attribué maintenant le type `list`.

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

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

(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 Δ contenant un ou plusieurs types pour f.

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

$$\Delta_{op} \vdash \texttt{aux} : \{\texttt{list}\} \to \{\texttt{list}\}.$$

Donner la dérivation de typage complète.

Correction :

```
  en posant G := {s:list} et any={none,bool,int,list}

                                              a:any  b:any
                                             --------------
  s:list  0:int      s:list  1:int           a+b:{int,list}  s:list
  --------------   -------------------   --------------------------
  G|-s[0] : any    G+{a:any}|-s[1]:any   G+{a,b:any}|-[a+b]+s:{list}
-------------------------------------------------------------------
                      |- aux : {list} -> {list}
```

Question 11 Proposer au moins deux types σ différents tels que, pour chacun, on ait

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

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 σ.)

Correction :

$$\begin{aligned} \texttt{myst} &: \{\texttt{int}\} \times \{\texttt{list}\} \to \{\texttt{list}\} \\ \texttt{myst} &: \{\texttt{bool}, \texttt{int}\} \times \{\texttt{list}\} \to \{\texttt{list}\} \end{aligned}$$

Question 12 Proposer un type pour la fonction

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

qui soit le plus informatif possible.

Correction :

$$\texttt{loop} : \{\texttt{none}, \texttt{bool}, \texttt{int}, \texttt{list}\} \to \emptyset$$

Question 13 On souhaiterait montrer la sûreté de notre typage statique dans le sens suivant : si $V, e \twoheadrightarrow v$ 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

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

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

Correction : Une première condition évidente est une cohérence entre Γ et V :

$$\text{pour tout } x \in \mathrm{dom}(V), \text{ on a } T(V(x)) \in \Gamma(x)$$

Une autre condition est la cohérence entre Δ et les opérations :

$$\begin{array}{l} \text{si } (op, \tau_0 \times \cdots \times \tau_{n-1} \to \tau) \in \Delta, \\ \text{alors pour tous } v_0, \ldots, v_{n-1} \text{ tels que } T(v_i) \in \tau_i, \\ \text{on a } T([\![op]\!](v_0, \ldots, v_{n-1})) \in \tau \end{array}$$

On peut vérifier que c'est bien le cas avec le contenu des figures 3 et 1. Enfin, il faut que toute fonction soit bien typée au regard de Δ, c'est-à-dire que, pour tout $(f, \sigma) \in \Delta$, on a $\Delta \vdash f : \sigma$.

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 Δ, pas de trouver un type pour f. Proposer un algorithme pour inférer un ensemble de types pour une fonction dont on a la définition.

Correction : Même si elle n'est pas très efficace, une solution consiste à donner successivement aux paramètres toutes les combinaisons possibles de types singletons, puis de typer la fonction dans cet environnement. Lorsqu'on obtient un type $\emptyset$ pendant le typage, on échoue, c'est-à-dire qu'on élimine ce cas-là. On donne alors à la fonction l'ensemble des types qu'on a obtenus.

Lorsque deux types ne diffèrent que par le type du résultat, ou que par le type d'un seul paramètre, on peut les réunir en un seul type avec une union (et alors recommencer). Cela améliore la lisibilité des types mais surtout l'efficacité du typage, qui aura moins de travail pour typer une application.

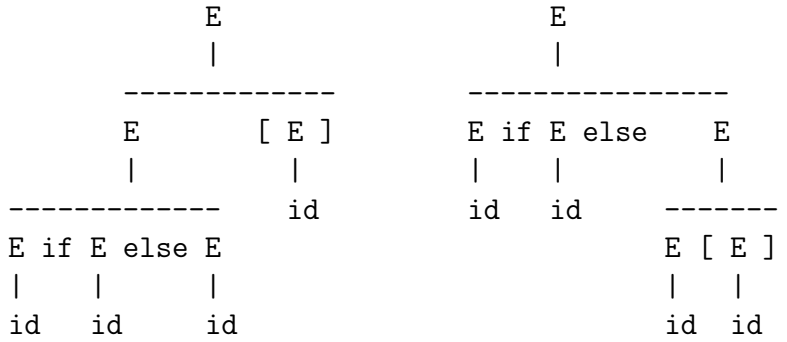
Il y a cependant une difficulté pour typer une fonction récursive f. On commence par lui donner un type $all \times \cdots \times all \to \emptyset$ pour faire une première itération (pendant laquelle il ne faut pas échouer sur une expression de type $\emptyset$ si cela est dû à un appel récursif, directement ou indirectement). Une fois un ensemble de types trouvé pour f, on recommence avec ces types-là. Le processus converge.

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

$$\begin{array}{rcl} E & ::= & \texttt{id} \\ & | & E\ \texttt{if}\ E\ \texttt{else}\ E \\ & | & E\ \texttt{[}\ E\ \texttt{]} \end{array}$$

Question 15 Montrer que cette grammaire est ambiguë.

Correction : On a deux arbres de dérivations pour l'expression `id if id else id[id]`, à savoir



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.

Correction : Il s'agit de deux conflits `shift/reduce` :

- le conflit `E if E else E . [E]` (réduction du `if-else` ou lecture du `[`)
- le conflit `E if E else E . if E else E` (réduction du `if-else` ou lecture du `if`)

On peut déclarer auprès de Menhir les priorités suivantes :

```
%nonassoc ELSE
%nonassoc IF
%nonassoc LSQB
```

c'est-à-dire la priorité la plus forte pour la lecture de `[`, puis pour la lecture de `IF` et enfin pour la réduction de `IF-ELSE` (la priorité étant celle du lexème le plus à droite de la règle à réduire, c'est-à-dire `ELSE`).

Question 17 On propose une autre grammaire pour ce langage :

$$\begin{array}{rcl} E & ::= & A \\ & | & A \texttt{ if } E \texttt{ else } E \\ A & ::= & \texttt{ident} \\ & | & A \texttt{ [} E \texttt{]} \end{array}$$

Montrer que cette grammaire est SLR(1).

Correction : On a $\text{NULL}(A) = \text{NULL}(E) = \texttt{false}$. On a $\text{FIRST}(A) = \text{FIRST}(E) = \{\texttt{ident}\}$. On a $\text{FOLLOW}(A) = \{\texttt{\#}, \texttt{[}, \texttt{]}, \texttt{else}, \texttt{if}\}$ et $\text{FOLLOW}(E) = \{\texttt{\#}, \texttt{]}, \texttt{else}\}$.
L'automate SLR(1) a dix états :

```
state 0:
  S ->  . E #
  E ->  . A
  E ->  . A if E else E
  A ->  . ident
  A ->  . A [ E ]
state 1:
  A -> ident .
state 2:
  S -> E . #
state 3: // pas de problème ici, car if et [ ne sont pas dans Follow(E)
  E -> A .
  E -> A . if E else E
  A -> A . [ E ]
state 4:
  E ->  . A
  E ->  . A if E else E
  A ->  . ident
  A ->  . A [ E ]
  A -> A [ . E ]
state 4:
  A -> A [ E . ]
state 6:
  A -> A [ E ] .
state 7:
  A ->  . ident
  A ->  . A [ E ]
  E ->  . A
  E ->  . A if E else E
  E -> A if . E else E
state 8:
  E -> A if E . else E
state 9:
  A ->  . ident
  A ->  . A [ E ]
  E ->  . A
```

```
  E ->  . A if E else E
  E -> A if E else . E
state 10:
  E -> A if E else E .
```

et aucun ne contient de conflit.

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	...	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 ?

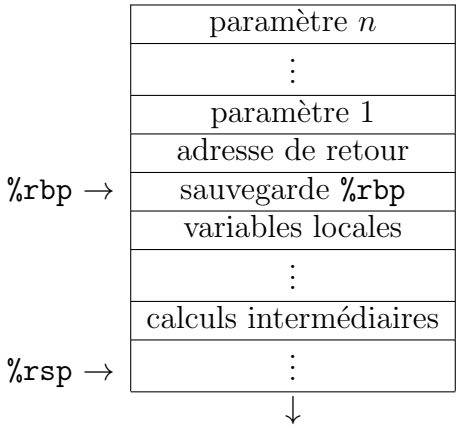
Correction : Ce choix est dicté par le fait qu'on ne connaisse pas toujours statiquement le type d'une valeur. En particulier, certains fonctions sont « polymorphes » au sens où elles doivent s'accommoder dynamiquement de valeurs de types différents. D'où le premier intérêt d'une représentation uniforme où toute valeur a la même taille (une adresse = un mot mémoire).

Par ailleurs, les fonctions doivent aussi pouvoir tester dynamiquement la nature de leurs paramètres. Ainsi, la fonction `+` a un comportement différents sur les booléens/entiers et sur les listes. L'étiquette que constitue le premier champ permet cela. Elle permet également à la fonction `print` d'afficher correctement la valeur (sans confondre les booléens et les entiers, en particulier).

On pourrait en effet représenter `None` par le pointeur nul. Cependant, on ne perd rien à le faire comme suggéré plus haut, car cette valeur peut être allouée statiquement, dans le segment de données (de même que les deux booléens). On y gagne même ainsi, comme on le verra plus bas avec la compilation de l'expression `if-else`.

En revanche, on ne pourrait pas représenter un entier directement par sa valeur, au risque de le confondre avec un pointeur, par exemple vers une liste.

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.

Correction :

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

C([e]) =>
  C(e)
  push %rdi
  mov  $24, %rdi
  call malloc
  mov  $3, (%rax)
  mov  $1, 8(%rax)
  pop  %rsi
  mov  %rsi, 16(%rax)

C(e1 if e2 else e3) =>
  C(e2)
  movq  8(%rax), %rcx
  testq %rcx, %rcx
  jnz   1f   # couvre tous les cas None, False, 0 et liste de longueur 0 !
  C(e3)
  jmp   2f
1: C(e1)
2:
```

On voit bien l'intérêt de la représentation sur ce dernier cas.

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.

Correction : L'information apportée par le typage statique nous permet d'éviter, dans certains cas, le recours à des tests dynamiques sur les types des valeurs. Ainsi, si dans la compilation d'une expression `e1+e2` on sait que `e1` et `e2` ont le type $\{\texttt{bool}, \texttt{int}\}$, alors on peut directement appeler la fonction qui fait l'addition de deux entiers, au lieu de commencer par tester si on doit faire une addition ou une concaténation ou échouer.

Exemple : la compilation de la fonction

```
def opt(x, y):
    return 1+x+y
```

qu'on aura typée avec l'unique type $\{\texttt{bool}, \texttt{int}\} \times \{\texttt{bool}, \texttt{int}\} \to \{\texttt{int}\}$.

On peut aller plus loin encore et ne pas allouer de valeurs intermédiaires sur le tas mais les stocker directement dans des registres ou sur la pile (*unboxing*). Dans la fonction ci-dessus, il n'est pas nécessaire d'allouer un bloc pour le résultat de `1+x`, mais seulement pour le résultat final. C'est possible car la sous-expression `1+x` a été typée de type $\{\texttt{bool}, \texttt{int}\}$.

$$
\begin{array}{rcll}
e & ::= & c & \textit{constante} \\
& | & x & \textit{variable} \\
& | & op(e, \ldots, e) & \textit{opération} \\
& | & f(e, \ldots, e) & \textit{application} \\
& | & e \texttt{ if } e \texttt{ else } e & \textit{conditionnelle}
\end{array}
\qquad
\begin{array}{rcl}
c & ::= & \texttt{None} \\
& | & b \quad b \in \{\texttt{True}, \texttt{False}\} \\
& | & n \quad n \in \mathbb{Z} \\
d & ::= & f(x, \ldots, x) \stackrel{\text{def}}{=} x = e; \ldots; x = e; \texttt{return } e \\
p & ::= & d \ldots d \texttt{ print}(e)
\end{array}
$$

FIGURE 2 – Syntaxe abstraite.

$$
\begin{array}{rrcl}
\text{valeur} & v & ::= & c \\
& & | & [v, \ldots, v] \\
\text{environnement} & V & ::= & x \mapsto v
\end{array}
\qquad
\begin{array}{rcl}
num(\texttt{False}) & \stackrel{\text{def}}{=} & 0 \\
num(\texttt{True}) & \stackrel{\text{def}}{=} & 1 \\
num(n) & \stackrel{\text{def}}{=} & n
\end{array}
$$

$$
\frac{}{V, c \twoheadrightarrow c}
\qquad
\frac{x \in \mathrm{dom}(V)}{V, x \twoheadrightarrow V(x)}
\qquad
\frac{V, e_2 \twoheadrightarrow v_2 \quad v_2 \notin \{\texttt{None}, \texttt{False}, 0, []\} \quad V, e_1 \twoheadrightarrow v_1}{V, e_1 \texttt{ if } e_2 \texttt{ else } e_3 \twoheadrightarrow v_1}
$$

$$
\frac{V, e_2 \twoheadrightarrow v_2 \quad v_2 \in \{\texttt{None}, \texttt{False}, 0, []\} \quad V, e_3 \twoheadrightarrow v_3}{V, e_1 \texttt{ if } e_2 \texttt{ else } e_3 \twoheadrightarrow v_3}
$$

$$
\frac{V, e_i \twoheadrightarrow v_i \quad [\![op]\!](v_0, \ldots, v_{n-1}) = v}{V, op(e_0, \ldots, e_{n-1}) \twoheadrightarrow v}
$$

$$
\frac{\begin{array}{c}
\forall\ 0 \leq i < n,\ V, e_i \twoheadrightarrow v_i \\
f(x_0, \ldots, x_{n-1}) \stackrel{\text{def}}{=} x_n = e_n; \ldots; x_{m-1} = e_{m-1}; \texttt{return } e \\
V_n \stackrel{\text{def}}{=} \{x_0 \mapsto v_0; \ldots; x_{n-1} \mapsto v_{n-1}\} \\
\forall\ n \leq i < m,\ V_i, e_i \twoheadrightarrow v_i \quad V_{i+1} \stackrel{\text{def}}{=} V_i[x_i \mapsto v_i] \\
V_m, e \twoheadrightarrow v
\end{array}}{V, f(e_0, \ldots, e_{n-1}) \twoheadrightarrow v}
$$

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

`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
`add` r_2`,` r_1	calcule la somme de r_1 et r_2 dans r_1 (on a de même `sub` et `imul`)
`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

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`.