Langages, interprétation, compilation – Examen
Lundi 13 décembre 2021 – durée 2h – tous documents autorisés. Les exercices sont indépendants.
Les temps de résolution sont des suggestions, et indiquent approximativement le barème.

1. Mise en jambes : rétro-ingénierie (10 minutes)

Observons comment une même expression peut être interprétée différemment en Caml (à gauche) et en Python (à droite).

```
# 1 = 0 = false ;;
- : bool = true
```

```
>>> 1 == 0 == False
False
```

De légères modifications de l'expression peuvent également changer l'interprétation.

```
# 0 = 0 = false ;;
- : bool = false

# false = 0 = 1 ;;
          ^
Error: This expression has type int but
an expression was expected of type bool
```

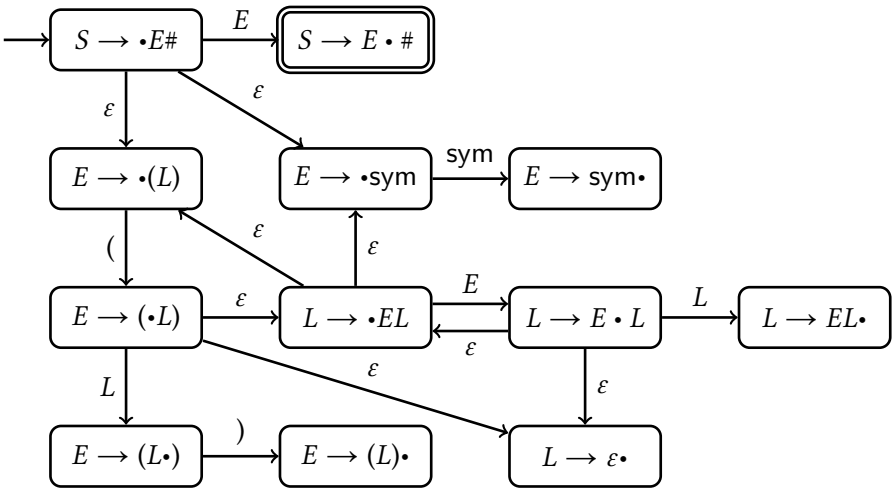
```
>>> 0 == 0 == False
True

>>> False == 0 == 1
False
```

Questions

1. Que pouvez-vous en déduire sur l'arbre de syntaxe abstraite produit par les analyseurs syntaxiques respectifs de Caml et de Python à partir d'une expression de la forme `e1 = e2 = e3` (caml) ou `e1 == e2 == e3` (python) ?
2. Dans chacun de ces cas, quelles sont les combinaisons de types acceptables pour les trois expressions `e1`, `e2` et `e3` ?

2. Analyse ascendante (30 minutes)

Voici l'automate LR(0) non déterministe d'une certaine grammaire G.



Questions

1. Quelle est cette grammaire G ?
2. Déterminiser l'automate.
3. Décrivez les conflits LR(0) de cette grammaire et des entrées menant à ces conflits.
4. Cette grammaire est-elle SLR(1) ?

3. Compilation des valeurs optionnelles (30 minutes)

Pour représenter en mémoire des valeurs optionnelles telles que celles du type `'a option` de Caml, on propose de procéder ainsi :

— La valeur `None` est représentée comme l'entier `0`.

— La valeur `Some v` est représentée par un pointeur vers un bloc alloué dans le tas, formé d'une part d'un entête contenant le nombre d'éléments dans le bloc (en l'occurrence, 1), et d'autre part d'un champ par élément du bloc (en l'occurrence, un unique champ pour l'unique élément `v`).

Note : la valeur `0` n'est jamais reconnue comme un pointeur valide.

Questions

1. Décrire l'état des registres et du tas après l'exécution du code suivant, et préciser la valeur contenue dans le registre `$v0`.

```
    li   $a0, 8
    li   $v0, 9
    syscall
    li   $t0, 1
    sw   $t0, 0($v0)
    li   $t0, 2
    sw   $t0, 4($v0)
    move $t0, $v0
    li   $v0, 9
    syscall
    sw   $t0, 4($v0)
    li   $t0, 1
    sw   $t0, 0($v0)
```

2. Supposons que le registre `$a0` contienne la valeur `Some (Some (Some 3))`. Donner une configuration possible du tas, en précisant les adresses des blocs représentés et leur contenu, sachant que la première adresse du tas est `0x10040000`.
3. Supposons que le registre `$a0` contienne une valeur de la forme `Some (Some` n`)`. Écrire un fragment de code MIPS qui place la valeur n dans le registre `$v0`.
4. Donner le code MIPS pour une fonction f qui prend en entrée une valeur de type `int option` et est telle que :

$$\begin{aligned} f(\texttt{None}) &= -1 \\ f(\texttt{Some } n) &= n+1 \end{aligned}$$

   Le paramètre est passé par le registre `$a0` et le résultat est passé par le registre `$v0`.
5. Supposons que les registres `$a0` et `$a1` contiennent chacun une valeur de type `int option`. Écrire un fragment de code MIPS qui écrit `1` dans le registre `$v0` si les deux valeurs sont égales, et `0` sinon.

4. Exceptions : typage et sémantique (50 minutes)

On considère dans cet exercice le langage FUN, avec ses types simples et sa sémantique à grands pas en appel par valeur, dont les règles sont rappelées en annexe. On va étendre ce langage avec des mécanismes d'exceptions, comme `try/with` en caml ou `try/catch` en java.

On considère donc le fragment suivant :

$$
\begin{array}{rcl}
e & ::= & n \\
& | & e \texttt{ + } e \\
& | & x \\
& | & \texttt{let } x \texttt{ = } e \texttt{ in } e \\
& | & \texttt{fun } x \texttt{ -> } e \\
& | & e\ e \\
& | & \texttt{raise } e \\
& | & \texttt{try } e \texttt{ with } x \texttt{ -> } e
\end{array}
$$

où une expression `raise` e déclenche une exception désignée par l'expression e, et où une expression `try` e_1 `with` x `->` e_2 essaie d'évaluer e_1 et bascule sur l'évaluation de e_2 si e_1 déclenche une exception.

La sémantique à grands pas associe à une expression e un *résultat* r, qui est soit une valeur v au sens habituel (un entier, une fonction, ou une exception), soit l'indication `exn` v qu'une exception v a été déclenchée.

$$
\begin{array}{rcl}
v & ::= & n \\
& | & \texttt{fun } x \texttt{ -> } e \\
& | & E \\
r & ::= & v \\
& | & \texttt{exn } v
\end{array}
$$

Les exceptions E ont le type `exn`, qui est un type de base au même titre que `int`.

$$
\begin{array}{rcl}
\tau & ::= & \texttt{int} \\
& | & \texttt{exn} \\
& | & \tau \rightarrow \tau
\end{array}
$$

On a donc la règle de typage suivante pour `raise`.

$$\frac{\Gamma \vdash e : \texttt{exn}}{\Gamma \vdash \texttt{raise } e : \tau}$$

On donne les règles de sémantique suivantes pour les nouvelles constructions.

$$\frac{e \Longrightarrow v}{\texttt{raise } e \Longrightarrow \texttt{exn } v} \qquad \frac{e_1 \Longrightarrow v}{\texttt{try } e_1 \texttt{ with } x \texttt{ -> } e_2 \Longrightarrow v} \qquad \frac{e_1 \Longrightarrow \texttt{exn } v_1 \qquad e_2[x := v_1] \Longrightarrow v}{\texttt{try } e_1 \texttt{ with } x \texttt{ -> } e_2 \Longrightarrow v}$$

On ajoute à ces cas de base des règles pour propager les exceptions. Voici les deux règles relatives aux applications :

$$\frac{e_1 \Longrightarrow \texttt{exn } v_1}{e_1\ e_2 \Longrightarrow \texttt{exn } v_1} \qquad \frac{e_1 \Longrightarrow v_1 \qquad e_2 \Longrightarrow \texttt{exn } v_2}{e_1\ e_2 \Longrightarrow \texttt{exn } v_2}$$

Questions

1. Donner les règles de propagation qui manquent.
2. En supposant que `E` est une exception, donner un arbre de dérivation pour l'évaluation de

```
let f = fun x -> raise E in try f 1 with x -> 0
```

3. Donner une règle de typage pour la construction `try/with`.

On va étudier la sémantique des exceptions à travers leur traduction dans un mécanisme plus élémentaire de filtrage. On étend pour cela FUN avec deux constructeurs unaires `V` et `E`, et une construction `match/with` fonctionnant comme en caml, pour distinguer ces deux constructeurs. L'ensemble des valeurs est étendu avec :

$$\begin{array}{rcl} v & ::= & \ldots \\ & | & \texttt{V}(v) \\ & | & \texttt{E}(v) \end{array}$$

et on a une nouvelle forme $\texttt{res}_\tau$ pour les types de ces constructeurs.

Voici les règles de typage associées à ces nouvelles formes.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{V}(e) : \texttt{res}_\tau} \qquad \frac{\Gamma \vdash e : \texttt{exn}}{\Gamma \vdash \texttt{E}(e) : \texttt{res}_\tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{res}_\sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau \qquad \Gamma, x : \texttt{exn} \vdash e_3 : \tau}{\Gamma \vdash \texttt{match } e_1 \texttt{ with V}(x) \texttt{ -> } e_2 \texttt{ | E}(e_3) \texttt{ -> } e_3 : \tau}$$

Voici les règles de sémantique à grands pas.

$$\frac{e \Longrightarrow v}{\texttt{V}(e) \Longrightarrow \texttt{V}(v)} \qquad \frac{e \Longrightarrow v}{\texttt{E}(e) \Longrightarrow \texttt{E}(v)}$$

$$\frac{e_1 \Longrightarrow \texttt{V}(v_1) \qquad e_2[x := v_1] \Longrightarrow v_2}{\texttt{match } e_1 \texttt{ with V}(x) \texttt{ -> } e_2 \texttt{ | E}(x) \texttt{ -> } e_3 \Longrightarrow v_2}$$

$$\frac{e_1 \Longrightarrow \texttt{E}(v_1) \qquad e_3[x := v_1] \Longrightarrow v_3}{\texttt{match } e_1 \texttt{ with V}(x) \texttt{ -> } e_2 \texttt{ | E}(x) \texttt{ -> } e_3 \Longrightarrow v_3}$$

On définit une fonction f transformant expression FUN avec exceptions en une expression FUN avec filtrage. Après transformation, les expressions produiront soit un résultat $\texttt{V}(v)$ pour une expression d'origine s'évaluant sans exception et produisant la valeur v, soit un résultat $\texttt{E}(v)$ pour une expression d'origine déclenchant une exception v.

Voici quelques cas de cette traduction.

$$\begin{array}{rcl} f(x) & = & V(x) \\ f(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) & = & \texttt{match } f(e) \texttt{ with V}(x) \texttt{ -> } f(e_2) \texttt{ | E}(v) \texttt{ -> E}(v) \\ f(\texttt{raise } e) & = & \texttt{match } f(e) \texttt{ with V}(v) \texttt{ -> E}(v) \texttt{ | E}(v) \texttt{ -> E}(v) \\ f(\texttt{try } e_1 \texttt{ with } x \texttt{ -> } e_2) & = & \texttt{match } f(e_1) \texttt{ with V}(v) \texttt{ -> V}(v) \texttt{ | E}(x) \texttt{ -> } f(e_2) \end{array}$$

Questions

1. Donner une équation pour $f(e_1\ e_2)$.
2. Traduire l'expression `try f a with exn -> 0`.
3. Démontrer que si une expression de la forme $\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2$ est bien typée, alors l'expression $\texttt{match } f(e) \texttt{ with V}(x) \texttt{ -> } f(e_2) \texttt{ | E}(v) \texttt{ -> E}(v)$ est elle-même bien typée. Quel est son type ?
4. On veut démontrer que la transformation préserve la sémantique, et plus précisément que pour tout e,
   - si $e \Longrightarrow v$ alors $f(e) \Longrightarrow f(v)$,
   - si $e \Longrightarrow \texttt{exn } v$ et $f(v) = V(v')$ alors $f(e) \Longrightarrow E(v')$.

   On le démontre par récurrence sur e. Rédigez les cas relatifs à la constante n, à la levée $\texttt{raise } e$ d'une exception, et à l'application $e_1\ e_2$. *Pour ce dernier cas vous avez besoin d'un lemme : énoncez-le sans démonstration.*
5. En vous inspirant de cet exercice, décrivez une manière dont on pourrait écrire en caml un interprète pour le langage FUN étendu avec les exceptions (mais sans utiliser les exceptions de caml). *Il n'est pas nécessaire de donner du code. Vous pouvez en revanche préciser les types de données utilisés et votre stratégie générale.*

Annexe 1 : aide-mémoire MIPS

Voici quelques instructions MIPS susceptibles de vous être utiles (vous avez le droit d'en utiliser d'autres) :

`li` r`,` n	charge l'entier n dans le registre r
`move` r_1`,` r_2	copie le registre r_2 dans le registre r_1
`add` r_1`,` r_2`,` r_3	calcule la somme de r_2 et r_3 et la place dans r_1
`lw` r_1`,` n`(`r_2`)`	charge dans r_1 la valeur contenue à l'adresse $r_2 + n$
`sw` r_1`,` n`(`r_2`)`	écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans r_1
`b` L	saute à l'adresse désignée par l'étiquette L
`beqz` r`,` L	saute à l'adresse désignée par l'étiquette L si le registre r contient `0`
`syscall`	effectue un appel système, dont la nature est donnée par le registre `$v0` ; par exemple, si `$v0` contient `9`, alors l'appel déclenché est `sbrk`, qui alloue sur le tas un nombre d'octets donné par `$a0`, et qui place dans `$v0` l'adresse de début du bloc ainsi alloué

Annexe 2 : règles de typage et de sémantique pour FUN

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \longrightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \longrightarrow \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

$$\frac{}{n \Longrightarrow n} \qquad \frac{}{\texttt{fun } x \texttt{ -> } e \Longrightarrow \texttt{fun } x \texttt{ -> } e}$$

$$\frac{e_1 \Longrightarrow n_1 \qquad e_2 \Longrightarrow n_2}{e_1 \texttt{ + } e_2 \Longrightarrow n_1 + n_2} \qquad \frac{e_1 \Longrightarrow v_1 \qquad e_2[x := v_1] \Longrightarrow v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Longrightarrow v}$$

$$\frac{e_1 \Longrightarrow \texttt{fun } x \texttt{ -> } e \qquad e_2 \Longrightarrow v_2 \qquad e[x := v_2] \Longrightarrow v}{e_1\ e_2 \Longrightarrow v}$$