Langages, interprétation, compilation – Examen
Lundi 12 décembre 2022 – durée 2h – tous documents autorisés. Les exercices sont indépendants. Les temps de résolution sont des suggestions, et indiquent approximativement le barème.

1 Petits exercices

1.1 Termes et récurrence (15 minutes)

On représente des listes d'entiers par les termes définis par la signature

$$\Sigma = \{\mathsf{Nil} : \mathsf{liste}, \mathsf{Cel} : \mathbb{N} \times \mathsf{liste} \to \mathsf{liste}\}$$

On rappelle les équations définissant la longueur d'une liste et la concaténation de deux listes.

$$\begin{aligned} \mathsf{longueur}(\mathsf{Nil}) &= 0 \\ \mathsf{longueur}(\mathsf{Cel}(x, l)) &= 1 + \mathsf{longueur}(l) \end{aligned}$$

$$\begin{aligned} \mathsf{concat}(\mathsf{Nil}, l_2) &= l_2 \\ \mathsf{concat}(\mathsf{Cel}(x, l_1), l_2) &= \mathsf{Cel}(x, \mathsf{concat}(l_1, l_2)) \end{aligned}$$

Questions.

1. Démontrer par récurrence que, pour toute liste l, on a $\mathsf{concat}(l, \mathsf{Nil}) = l$.
2. On souhaite démontrer que, pour toute liste l, on a $\mathsf{longueur}(\mathsf{concat}(l, l)) = 2 \times \mathsf{longueur}(l)$. Que se passe-t-il si l'on tente de prouver cet énoncé directement par récurrence sur la structure de l ? Pouvez-vous proposer une autre stratégie ?

1.2 Grammaires et expressions régulières (20 minutes)

On considère les deux grammaires suivantes générant des mots sur l'aphabet $\Sigma = \{a, b\}$.

$$\begin{array}{llll} (G_1) & E & \to & abbE \\ & & | & bE \\ & & | & \varepsilon \end{array} \qquad \begin{array}{llll} (G_2) & E & \to & aEbEbE \\ & & | & bE \\ & & | & \varepsilon \end{array}$$

Questions.

1. Par quelle(s) grammaire(s) les mots suivants peuvent-ils être générés ?
   (a) $babb$ (b) $ababbb$ (c) bab (d) $abbb$
2. Décrire l'ensemble des mots générés par G_1. Est-il reconnaissable par une expression régulière ou un automate ? (donner une justification adaptée)
3. Décrire l'ensemble des mots générés par G_2. Est-il reconnaissable par une expression régulière ou un automate ? (donner une justification adaptée)

2 Problème : pointeurs de fonctions

On considère un petit langage dans lequel on peut manipuler des pointeurs de fonctions : la spécificité du langage est que l'on peut passer en paramètre à une fonction f un pointeur vers une autre fonction g qui sera appelée par f. Ainsi dans ce langage, une fonction f n'est plus un identifiant d'une nature particulière, mais une simple variable dont la valeur est l'adresse de la fonction correspondante. Une définition

```
int f(int x, int y) := x + y
```

définit donc une variable f de type (int, int) → int susceptible d'être appliquée à une paire de paramètres.

2.1 Analyse syntaxique (25 minutes)

Voici un fragment de définition en Menhir de la grammaire de notre langage (on ne montre pas la définition des symboles non terminaux instr, formals, types et parameters).

```
%token ID CST ADD LP RP SET INT ARROW
%right ADD
%start <unit>prog
%%

prog:
| decls; instr; EOF {}

decls:
| (* empty *) {}
| fun_decl; decls {}
| var_decl; decls {}

fun_decl:
| typed_id; LP; formals; RP; SET; expr {}
```

```
var_decl:
| typed_id; SET; expr {}

typed_id:
| typ; ID {}

typ:
| INT {}
| LP; types; RP; ARROW; typ {}

expr:
| ID {}
| CST {}
| expr; ADD; expr {}
| expr; LP; parameters; RP {}
```

Questions.

1. Donner les étapes de l'analyse ascendante du fragment suivant en précisant pour chaque étape l'état de la pile, le fragment de l'entrée encore à lire, et l'action effectuée.

   ```
   int x := 1 + y + z
   ```

2. Analysez l'annexe présentant un extrait du fichier .conflicts produit par Menhir pour cette grammaire.
   Combien de conflits sont-ils mentionnés dans l'extrait du fichier .conflicts ? Pour chacun de ces conflits, donner
   - une entrée en syntaxe concrète aboutissant au conflit,
   - des arbres de dérivation justifiant les différentes possibilités,
   - des priorités sur les opérateurs ou une modification de la grammaire permettant d'éliminer le conflit.

2.2 Types (40 minutes)

Pour notre langage avec pointeurs de fonctions, on introduit deux formes de types : le type des entiers et les types des fonctions. On note $(T_1, \ldots, T_n) \to T$ le type des fonctions prenant n paramètres de types T_1 à T_n et produisant un résultat de type T.

Questions.

1. Donner des types possibles pour les fonctions `f` et `g` déclarées ainsi :

   ```
   ... f(... x, ... y) := x + y
   ... g(... x, ... y) := f(x(y), 1)
   ```

On décrit formellement le typage d'une expression par un jugement de typage $\Gamma \vdash e : T$ signifiant que l'expression e est de type T dans l'environnement Γ, l'environnement Γ étant une fonction qui à chaque variable associe un type. On peut justifier un jugement de typage à l'aide des règles de typage suivantes :

$$\frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \ + \ e_2 : \mathsf{int}}$$

Questions.

2. Proposer une règle de typage pour une expression d'application de fonction de la forme $e_0(e_1, \ldots, e_n)$.
3. On se place dans un environnement Γ associant à `x` le type `int` et à `f` le type $(\mathsf{int}, \mathsf{int}) \to ((\mathsf{int}) \to \mathsf{int})$. Les expressions suivantes sont-elles bien typées ? Donner une dérivation de typage ou expliquer le problème.
   (a) `1 + f(x, 2)`
   (b) `1 + (f(x, 2))(3)`

Pour manipuler les expressions et les types de ce langage en Caml, on se donne les deux définitions suivantes :

```
type expression =
  | Id of string
  | Cst of int
  | Add of expression * expression
  | Call of expression * expression list
```

```
type typ =
  | Int
  | Function of typ * typ list
```

Questions.

4. On souhaite produire une fonction `type_expression: expression -> env -> typ` calculant le type de l'expression donnée en paramètre, ou levant une exception dans le cas où l'expression est incohérente. Écrire les cas correspondant aux constructeurs `Add` et `Call`. On ne se préoccupera pas de la manière dont le type `env` est défini.

La transformation d'*extension inline* modifie le code source du programme en remplaçant un appel de fonction par le code de la fonction. Plus précisément, si la fonction f a une définition de la forme $f(T_1\ x_1, \ldots, T_n\ x_n) = e$, alors on remplacera un appel $f(e_1, \ldots, e_n)$ par l'expression e^σ, dans laquelle σ est la substitution remplaçant la variable x_i par l'expression e_i, pour tout i entre 1 et n. Formellement, la définition de l'application à une expression d'une substitution σ remplaçant x_i par e_i est :

$$\begin{aligned}
(x_i)^\sigma &= e_i \\
x^\sigma &= x \qquad\qquad x \notin \{x_1, \ldots, x_n\} \\
n^\sigma &= n \\
(e_1 + e_2)^\sigma &= e_1^\sigma + e_2^\sigma \\
(e_0(e_1, \ldots, e_n))^\sigma &= e_0^\sigma(e_1^\sigma, \ldots, e_n^\sigma)
\end{aligned}$$

On souhaite démontrer que cette optimisation respecte le bon typage d'un programme.

Questions.

5. Supposons que f est définie par $f(T_1\ x_1, \ldots, T_n\ x_n) = e$ et que σ est la substitution remplaçant x_i par e_i. Démontrer que si $\Gamma \vdash f(e_1, \ldots, e_n) : T$, alors $\Gamma \vdash e^\sigma : T$.

2.3 Programmation assembleur (25 minutes)

On se donne la convention suivante pour la compilation des expressions et des fonctions.

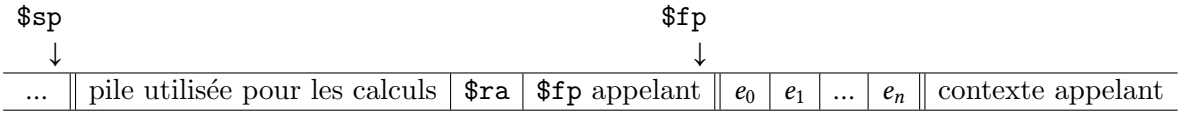
— La valeur d'une fonction est un pointeur vers son code.

— Lors d'un appel de fonction $e_0(e_1, \ldots, e_n)$, l'appelant place au sommet de la pile les valeurs e_n à e_1, dans cet ordre (c'est-à-dire avec la valeur de e_1 au sommet), puis passe la main à la fonction dont la valeur est donnée par e_0. Après l'appel, l'appelant retire de la pile les valeurs e_1 à e_n.

— Une fonction renvoie son résultat via le registre `$v0`.

— L'appelé doit sauvegarder les registres `$fp` et `$ra` avant de commencer son calcul, puis restaurer ces mêmes registres avant de rendre la main à l'appelant.

— Le tableau d'activation d'un appel de fonction a la forme suivante :



Questions.

1. On se donne les trois définitions de fonctions suivantes, où l'expression `e` n'est pas détaillée.

```
int f(int x, int y) = e
int g(int z) = 3 + f(z, 2) + 3
int h(int t) = t + 1
```

   On exécute ensuite une instruction `print(g(h(5)));`. Dessiner la pile et préciser le contenu de chaque case au moment où l'expression *e* vient d'être évaluée (c'est-à-dire avant la fin de l'appel à `f`).

2. On considère le code MIPS suivant.

```
li $t0, 1
li $t1, 4
add $t0, $t0, $t1
li $t1, 5
mul $t1, $t0, $t1
sw $t1, 0($sp)
addi $sp, $sp, -4
```

   À la fin de l'exécution de ce code, que contiennent les registres `$t0` et `$t1` ? Comment a évolué la pile ?

3. On se place dans le corps d'une fonction `f` à deux paramètres :
   — le premier paramètre est un entier `x`,
   — le deuxième paramètre est une fonction `g` de type `(int, int) -> int`.

   Dessiner le tableau d'activation d'un appel à `f`, en précisant la localisation des pointeurs de référence et des deux paramètres, puis donner un code assembleur réalisant l'appel `g(x, 1)`.
   Indication : si `$r` est un registre contenant l'adresse du code d'une fonction, l'instruction MIPS `jalr $r` déclenche l'appel de cette fonction. Comme avec `jal`, l'adresse de retour est alors sauvegardée dans le registre `$ra`.

Annexe. 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
`jr` r	saute à l'adresse donnée par le registre r
`jalr` r	saute à l'adresse donnée par le registre r, et sauvegarde une adresse de retour dans `$ra`

Annexe. Fichier `.conflicts`

```
** Conflict (shift/reduce) in state 19.
** Token involved: LP
** This state is reached from prog after reading:

typed_id SET expr ADD expr

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the derivations begin to differ.)

prog
decls instr EOF
(?)

** In state 19, looking ahead at LP, reducing production
** expr -> expr ADD expr
** is permitted because of the following sub-derivation:

var_decl decls // lookahead token appears because decls can begin with LP
typed_id SET expr // lookahead token is inherited
             expr ADD expr .

** In state 19, looking ahead at LP, shifting is permitted
** because of the following sub-derivation:

var_decl decls
typed_id SET expr
             expr ADD expr
                      expr . LP parameters RP

** Conflict (shift/reduce) in state 13.
** Token involved: LP
** This state is reached from prog after reading:

typed_id SET expr

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the derivations begin to differ.)

prog
decls instr EOF
(?)

** In state 13, looking ahead at LP, shifting is permitted
** because of the following sub-derivation:

var_decl decls
typed_id SET expr
             expr . LP parameters RP

** In state 13, looking ahead at LP, reducing production
** var_decl -> typed_id SET expr
** is permitted because of the following sub-derivation:

var_decl decls // lookahead token appears because decls can begin with LP
typed_id SET expr .
```