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

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

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 ?

Correction.

1. Preuve par récurrence structurelle sur l.
   — Cas Nil. Par définition, $\mathsf{concat}(\mathsf{Nil}, \mathsf{Nil}) = \mathsf{Nil}$.
   — Cas $\mathsf{Cel}(x, l)$. On suppose (hypothèse de récurrence) que $\mathsf{concat}(l, \mathsf{Nil}) = l$, et on calcule :

$$\begin{aligned} \mathsf{concat}(\mathsf{Cel}(x, l), \mathsf{Nil}) &= \mathsf{Cel}(x, \mathsf{concat}(l, \mathsf{Nil})) && \text{par définition} \\ &= \mathsf{Cel}(x, l) && \text{par hypothèse de récurrence} \end{aligned}$$

2. Par récurrence structurelle sur l, on aurait à considérer un cas $\mathsf{longueur}(\mathsf{concat}(\mathsf{Cel}(x, l), \mathsf{Cel}(x, l)))$ dans lequel on ne sait pas se ramener à l'hypothèse de récurrence sur $\mathsf{longueur}(\mathsf{concat}(l, l))$. On peut s'en sortir en généralisant l'énoncé pour détacher les deux listes : $\mathsf{longueur}(\mathsf{concat}(l_1, l_2)) = \mathsf{longueur}(l_1) + \mathsf{longueur}(l_2)$.

1.2 Grammaires et expressions régulières (20 minutes)

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

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

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)

Correction.

1.

	Mot	G_1	G_2
(a)	$babb$	✓	✓
(b)	$ababbb$	×	✓
(c)	bab	×	×
(d)	$abbb$	✓	✓

2. Mots dans lesquels chaque a est immédiatement suivi d'au moins deux b. Reconnaissable, par exemple avec l'expression $(abb|b)^*$.
3. Mots dans lesquels chaque a peut être associé à deux b situés sur sa droite, et qui ne sont pas déjà associés à un autre a. Une propriété important de ces mots : ils contiennent au moins deux fois plus de b que de a. Non reconnaissable. Supposons ce langage reconnaissable. Soit N l'entier donné par le lemme de l'étoile. On considère le mot $a^N b^{2N}$, qui appartient bien au langage. Par lemme de l'étoile forte, le facteur a^N peut être décomposé en $a^{n_1}a^{n_2}a^{n_3}$ avec $n_2 \neq 0$ de sorte que pour tout k, $a^{n_1}(a^{n_2})^k a^{n_3}$ soit dans le langage. En prenant $k = 2$, on aurait donc $a^{N+n_2}b^{2N}$ dans le langage. Or ce mot ne contient pas deux fois plus de b que de a : contradiction.

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.

Correction.

1. Note : `x`, `y` et `z` sont directement compris comme des instances de `ID`, et `1` comme une instance de `CST`.

Pile	Entrée	Action
∅	`int x := 1 + y + z`	S `int`
`int`	`x := 1 + y + z`	R `int`
`typ`	`x := 1 + y + z`	S `x`
`typ x`	`:= 1 + y + z`	R `typ x`
`typed_id`	`:= 1 + y + z`	S `:=`
`typed_id :=`	`1 + y + z`	S `1`
`typed_id := 1`	`+ y + z`	R `1`
`typed_id := expr`	`+ y + z`	S `+`
`typed_id := expr +`	`y + z`	S `y`
`typed_id := expr + y`	`+ z`	R `y`
`typed_id := expr + expr`	`+ z`	R `expr + expr`
`typed_id := expr`	`+ z`	S `+`
`typed_id := expr +`	`z`	S `z`
`typed_id := expr + z`	∅	R `z`
`typed_id := expr + expr`	∅	R `expr + expr`
`typed_id := expr`	∅	R `typed_id := expr`
`var_decl`	∅	

2. On a deux conflits progression/réduction.

   — État 19. Conflit rencontré sur l'entrée

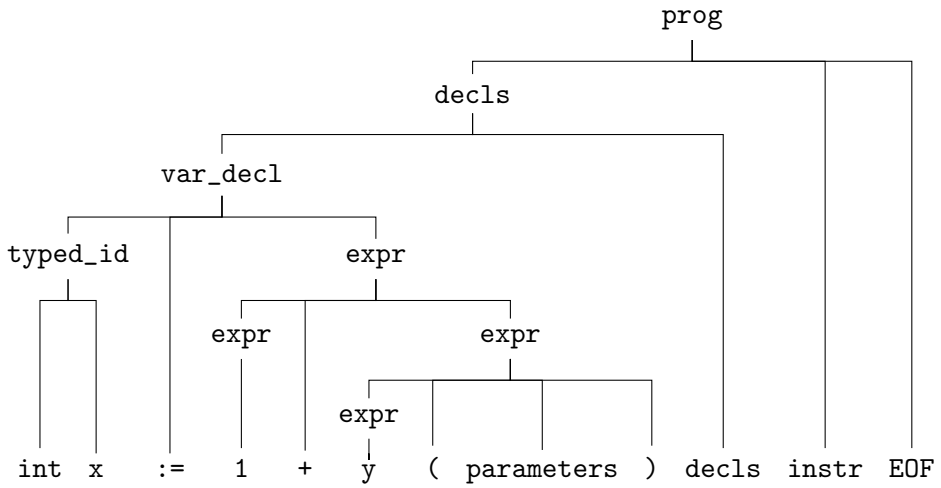
```
int x := 1 + y (2)
```

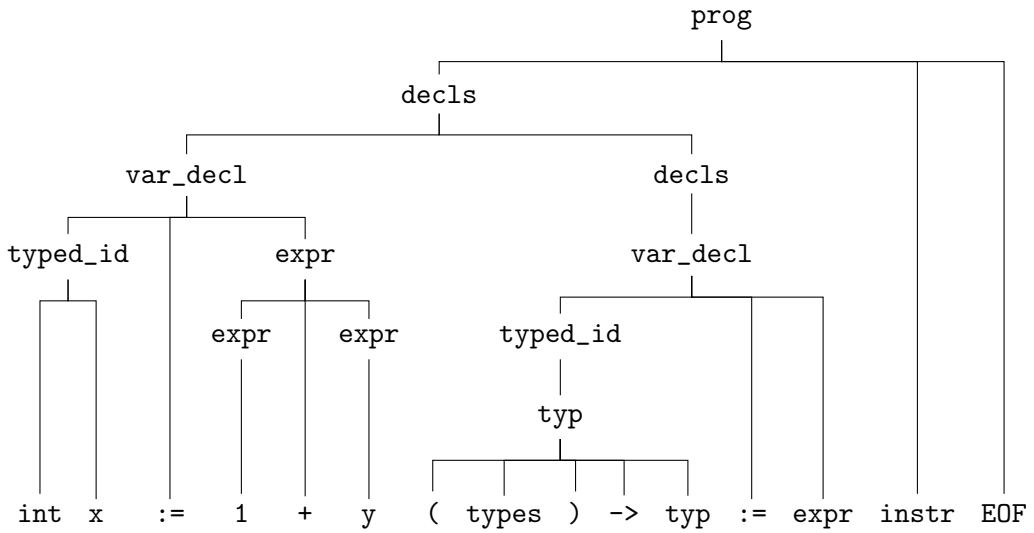
   ainsi que sur l'entrée

```
int x := 1 + y
(int) -> int f((int) -> (int) -> int g, int x) := g(x)
```

   On peut progresser, pour comprendre la parenthèse ouvrante comme annonçant les paramètres de la fonction `y`, ou réduire en considérant que la parenthèse ouvrante est le

début du type d'une nouvelle déclaration de variable.





On peut privilégier le premier scénario en déclarant la parenthèse ouvrante `(` plus prioritaire que l'addition `+` (ou inversement pour l'autre scénario).

— Le deuxième conflit est similaire. Il apparaît sur l'entrée

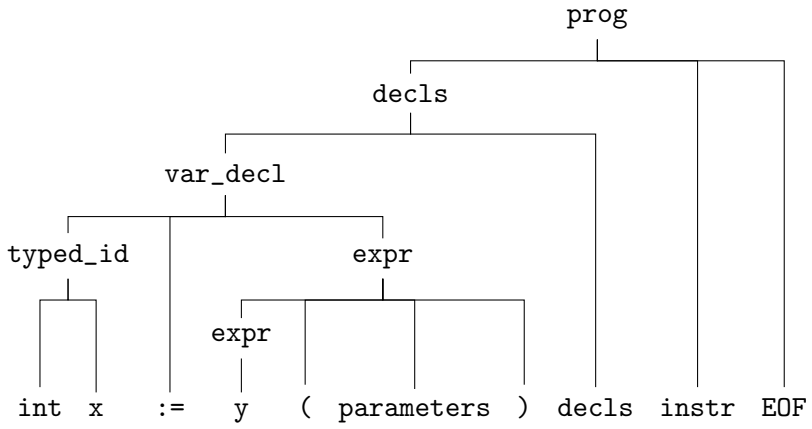
```
int x := y (2)
```

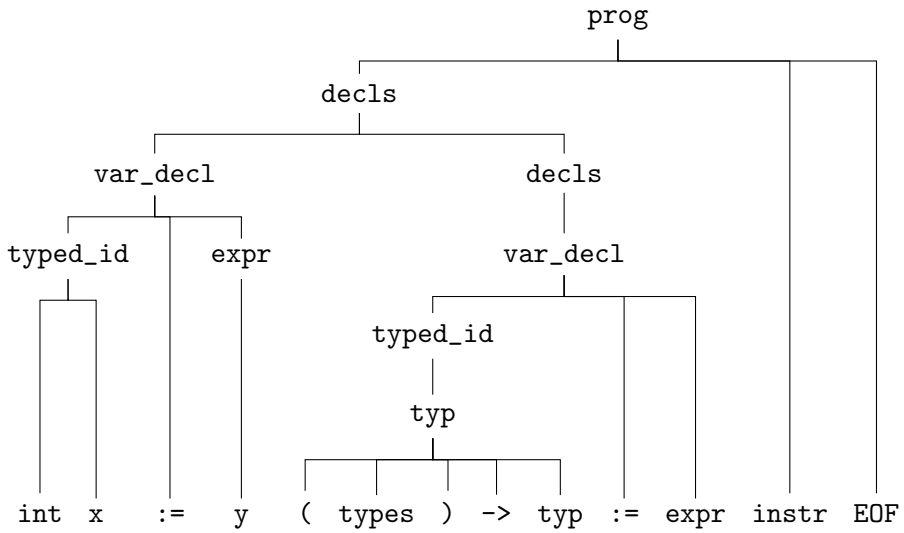
ainsi que sur l'entrée

```
int x := y
(int) -> int f((int) -> (int) -> int g, int x) := g(x)
```

On peut progresser, pour comprendre la parenthèse ouvrante comme annonçant les paramètres de la fonction `y`, ou réduire en considérant que la parenthèse ouvrante est le

début du type d'une nouvelle déclaration de variable.





On privilégie le premier scénario en déclarant la parenthèse ouvrante plus prioritaire que le symbole `:=` (ou inversement pour l'autre scénario).

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 $(T_1, \ldots, T_n) \longrightarrow T$ 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 (int, int) $\rightarrow$ ((int) $\rightarrow$ 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 :

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

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 $\Gamma \vdash e^\sigma : T$.

Correction.

1. On pourrait compléter les déclarations en

   ```
   int f(int x, int y) := x + y
   int g((int) -> int x, int y) := f(x(y), 1)
   ```

   Cela correspond à donner le type `(int, int) -> int` à `f` et le type `((int) -> int, int) -> int` à `g`.
2. On demande que e_0 ait le type d'une fonction à n arguments, et que les e_i aient respectivement les n types attendus.

$$\frac{\Gamma \vdash e_0 : (T_1, \ldots, T_n) \rightarrow T \qquad \Gamma \vdash e_1 : T_1 \qquad \ldots \qquad \Gamma \vdash e_n : T_n}{\Gamma \vdash e_0(e_1, \ldots, e_n) : T}$$

3. (a) L'expression `1 + f(x, 2)` est mal typée, car l'application de fonction `f(x, 2)` a le type `(int) -> int`, qui est incompatible avec l'addition.

   (b) Expression bien typée.

$$\dfrac{\dfrac{\dfrac{\Gamma(\texttt{f}) = \texttt{(int, int) -> (int) -> int}}{\Gamma \vdash \texttt{f} : \texttt{(int, int) -> (int) -> int}} \quad \dfrac{\Gamma(\texttt{x}) = \texttt{int}}{\Gamma \vdash x : \texttt{int}} \quad \dfrac{}{\Gamma \vdash 2 : \texttt{int}}}{\Gamma \vdash \texttt{f(x, 2)} : \texttt{(int) -> int}} \quad \dfrac{}{\Gamma \vdash 3 : \texttt{int}}}{\dfrac{\Gamma \vdash 1 : \texttt{int} \qquad \Gamma \vdash \texttt{(f(x, 2))(3)} : \texttt{int}}{\Gamma \vdash \texttt{1 + (f(x, 2))(3)} : \texttt{int}}}$$

4.
```
let rec type_expression e env = match e with
  ...
  | Add(e1, e2) ->
     if type_expression e1 env = Int && type_expression e2 env = Int
     then Int
     else failwith "type␣error"

  | Call(f, args) ->
     let return_type = match type_expression f env with
       | Function(t, targs) ->
          if List.for_all2 (fun a t -> type_expression a env = t) args targs
          then t
          else failwith "type␣error"
       | _ ->
          failwith "type␣error"
     in
     return_type
```

5. On suppose que $\Gamma \vdash f(e_1, \ldots, e_n) : T$ est dérivable. Ce jugement ne peut être obtenu que par la règle d'application. Étant donné le type $\Gamma \vdash (T_1, \ldots, T_n) \longrightarrow T : d$ e f, on a donc nécessairement des dérivations de $\Gamma \vdash e_i : T_i$, pour tout $i \in [1, n]$. On montre maintenant que pour tous e et T, si $\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash e : T$ alors $\Gamma \vdash e^\sigma : T$, par récurrence structurelle sur e.
   - Si $e = x_i$ et $\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash x_i : T_i$, alors $\Gamma \vdash x_i^\sigma : T$ car $x_i^\sigma = e_i$ et $\Gamma \vdash e_i : T$.
   - Si $e = x \notin x_1, \ldots, x_n$ et $\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash x : T$ avec $\Gamma(x) = T$, alors $\Gamma \vdash x^\sigma : T$ car $x^\sigma = x$ et $\Gamma \vdash x : T$.
   - Si $e = n$ et $\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash n : \mathsf{int}$, alors $\Gamma \vdash n^\sigma : \mathsf{int}$ car $n^\sigma = n$ et $\Gamma \vdash n : \mathsf{int}$.
   - Si $e = e_1 + e_2$ avec $\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash e_1 + e_2 : \mathsf{int}$ et $\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash e_1 : \mathsf{int}$ et $\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash e_2 : \mathsf{int}$. Par hypothèses de récurrence, $\Gamma \vdash e_1^\sigma : \mathsf{int}$ et $\Gamma \vdash e_2^\sigma : \mathsf{int}$. D'où $\Gamma \vdash e_1^\sigma + e_2^\sigma : \mathsf{int}$ par règle de typage de l'addition, et donc $\Gamma \vdash (e_1 + e_2)^\sigma : \mathsf{int}$ par définition de la substitution.
   - Cas $e = e_0(a_1, \ldots, a_k)$ similaire au cas de l'addition.

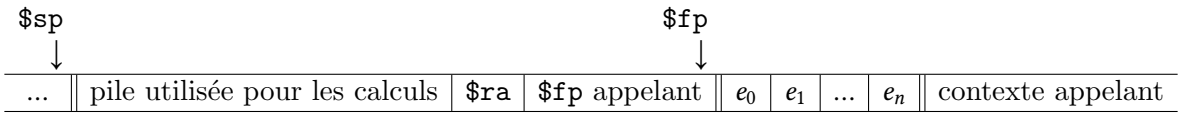
2.3 Programmation assembleur (25 minutes)

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 ?

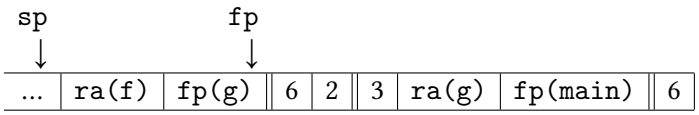
3. On se place dans le corps d'une fonction `f` à deux paramètres :
   — le premier paramètre est un entier `x`,
   — le deuxième paramètre est une fonction `g` de type `(int, int) -> int`.

   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)`.

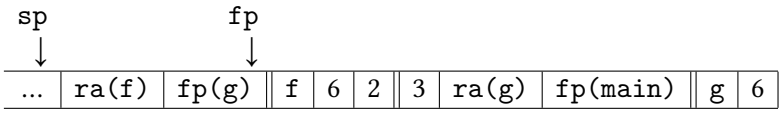
Indication : si `$r` est un registre contenant l'adresse du code d'une fonction, l'instruction MIPS `jalr $r` déclenche l'appel de cette fonction. Comme avec `jal`, l'adresse de retour est alors sauvegardée dans le registre *ra*.

Correction.

1. Précisons la chronologie des événements déclenchés par cette instruction. D'abord, on calcule la valeur du paramètre `h(5)` donné à `g` et cette valeur (`6`) est placée sur la pile (il ne reste en revanche rien d'autre de `h` ni de `5`). Ensuite, on commence l'appel à `g`. On analyse un premier opérande, valant `3` et dont on suppose ici qu'il est sauvegardé sur la pile avant de s'intéresser à l'appel `f(z, 2)`. On place sur la pile les deux paramètres `2` puis `z` (la valeur de ce dernier étant `6`). On passe alors la main à `f`, et on calcule l'expression `e`. Bilan de l'état de la pile après tout cela



Note : le schéma donné à la fin de l'énoncé faisait aussi apparaître e_0 sur la pile, ce que ne mentionnait pas le texte. En suivant ce modèle on a la variante suivante.

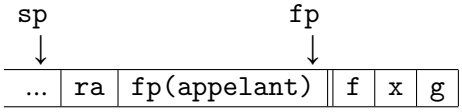


2. Évolution des registres au cours de l'exécution :

	`$t0`	`$t1`
`li $t0, 1`	1	
`li $t1, 4`	1	4
`add $t0, $t0, $t1`	5	4
`li $t1, 5`	5	5
`mul $t1, $t0, $t1`	5	25

Les deux dernières instructions ajoutent au sommet de la pile la valeur `25`, prise dans le registre `$t1`.

3. Tableau d'activation d'un appel à `f` (version où l'on fait apparaître `f` sur la pile avec les deux arguments, mais il serait admis de ne pas le faire).




```
li $t0, 1 # empilement valeur 1 en tant que 2e param de g
sw $t0, 0($sp)
addi $sp, $sp, -4
lw $t0, 8($fp) # lecture paramètre x de f
sw $t0, 0($sp) # ... et empilement en tant que 1er param de g
addi $sp, $sp, -4
lw $t0, 12($fp) # lecture paramètre g de f
sw $t0, 0($sp) # ... et empilement en tant que fonction appelée
addi $sp, $sp, -4
jalr $t0 # appel
addi $sp, $sp, 12 # après appel : nettoyage des 3 emplacements de pile
```

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 .
```