École Normale Supérieure

Langages de programmation et compilation

examen 2009–2010

Jean-Christophe Filliâtre

28 janvier 2010

Aucun document n'est autorisé.

Dans ce problème, on considère un fragment très simple du langage Python, sans qu'il soit nécessaire de connaître ce langage. Ce fragment se compose des quatre entités suivantes :

programme : Un programme est simplement une liste de définitions de fonctions ; on ne se préoccupe pas ici du point d'entrée du programme.

fonction : Une fonction prend en argument un n-uplet de valeurs entières ($n \geq 1$) et renvoie un m-uplet de valeurs entières ($m \geq 1$). Le corps d'une fonction est une liste d'instructions.

instruction : Une instruction est soit un retour de fonction (`return` e), soit l'affectation simultanée d'un n-uplet de valeurs entières à un n-uplet de variables, soit une conditionnelle (`if`) dont le test compare deux expressions entières, soit enfin un bloc formé d'une liste d'instructions.

expression : Une expression est soit une constante entière, soit une variable, soit l'application d'un opérateur arithmétique ($+$, $-$, $\times$, $/$) à deux expressions entières, soit un appel de fonction.

Toutes les variables contiennent des entiers. Voici trois exemples de programmes Python :

```
def div_eucl(a, b):
  if a < b:
    return (0, a)
  else:
    (q, r) = div_eucl(a-b, b)
    return (q+1, r)
```

```
def is_prime(d, n):
  if d*d > n: return 1
  if d * (n/d) == n: return 0
  return is_prime(d+1, n)

def count_primes(n, m):
  if n > m:
    return 0
  else:
    s = count_primes(n+1, m)
    return is_prime(2, n) + s
```

```
def g(x, y):
  if x > 10:
    x = 10
    if y > 10:
      y = 10
  (x,y) = (y,x)
  return x-y
```

La partie **??** étudie un aspect particulier de l'analyse syntaxique de Python. La partie **??** formalise la sémantique de ce fragment de Python et la partie **??** propose quelques vérifications de typage. Enfin, la partie **??** étudie la compilation vers l'assembleur MIPS. Les parties peuvent être traitées indépendamment, mais peuvent utiliser des informations contenues dans les parties précédentes.

1 Analyse syntaxique

Comme on le devine sur les exemples ci-dessus, la structure de bloc est définie par l'indentation des lignes, c'est-à-dire le nombre d'espaces en début de ligne (on omet ici les caractères de tabulation). L'objectif de cette partie est d'étudier l'analyse syntaxique correspondante. L'idée est simple : l'analyseur lexical produit des lexèmes `NEWLINE`, `BEGIN` et `END`, correspondant respectivement aux fins de lignes et à l'augmentation ou la diminution de l'indentation, et la grammaire prend alors la

forme suivante (on ne donne ici que le fragment intéressant) :

$$
\begin{array}{rcl}
\textit{instruction} & ::= & \textit{instruction-simple}\ \texttt{NEWLINE} \\
& | & \texttt{IF}\ \textit{condition}\texttt{:}\ \textit{suite} \\
& | & \texttt{IF}\ \textit{condition}\texttt{:}\ \textit{suite}\ \texttt{ELSE:}\ \textit{suite} \\
\textit{instruction-simple} & ::= & \texttt{RETURN}\ \textit{tuple-expr} \\
& | & \textit{tuple-ident}\ \texttt{=}\ \textit{tuple-expr} \\
\textit{suite} & ::= & \textit{instruction-simple}\ \texttt{NEWLINE} \\
& | & \texttt{NEWLINE BEGIN}\ \textit{instruction}^{+}\ \texttt{END} \\
\textit{def} & ::= & \texttt{DEF IDENT (}\ \textit{paramètres}\ \texttt{):}\ \textit{suite}
\end{array}
$$

Les non terminaux sont écrits en italique et $\textit{instruction}^{+}$ représente une liste d'au moins une occurrence du non terminal *instruction*. Sur une telle grammaire, on peut utiliser directement un outil de la famille yacc comme ocamlyacc ou menhir (on ne demande pas de le faire). Tout le travail autour de l'indentation se situe donc dans l'analyseur lexical.

Pour simplifier les choses, on suppose que les lignes vides ont été supprimées et qu'il n'y a pas de notion de commentaire. On propose alors l'algorithme suivant : l'analyseur lexical maintient une pile d'entiers, représentant les indentations en cours successives. Cette pile est triée, avec la valeur la plus grande au sommet. Initialement, la pile contient une unique valeur, à savoir 0. Lorsque l'analyseur lexical rencontre un retour-charriot, il produit un lexème `NEWLINE` puis mesure l'indentation au début de la ligne suivante, soit n, et la compare avec celle se trouvant au sommet de la pile, soit m. Trois cas se présentent :

- si $n = m$, on ne fait rien ;
- si $n > m$, on empile n et on produit un second lexème, à savoir `BEGIN` ;
- si $n < m$, alors on dépile jusqu'à trouver la valeur n, en produisant un lexème `END` pour chaque valeur strictement plus grande que n retirée de la pile (la valeur n restant en sommet de pile) ; si n n'apparaît pas dans la pile, on échoue en déclarant l'indentation incorrecte.

Question 1 Pour la fonction `g` page **??**, donner l'état de la pile et les lexèmes `NEWLINE`, `BEGIN` et `END` produits pour chaque fin de ligne.

Correction :

```
0       NEWLINE BEGIN
0,2     NEWLINE BEGIN
0,2,4   NEWLINE
0,2,4   NEWLINE BEGIN
0,2,4,6 NEWLINE END END
0,2     NEWLINE
0,2     NEWLINE END
```

Question 2 On utilise ocamllex pour écrire un analyseur lexical pouvant produire un ou plusieurs lexèmes à chaque appel, c'est-à-dire une fonction de type :

```
val next_tokens : lexbuf -> token list
```

Cet analyseur lexical prend la forme suivante :

```
rule next_tokens = parse
  | "def" { [DEF] }
```

```
| "if"  { [IF]  }
| "+"   { [PLUS] }
| ...
| eof   { [EOF] }
```

Écrire le code correspondant au traitement du caractère retour-charriot et de l'indentation de la ligne suivante. (On pourra utiliser `failwith` pour signaler une mauvaise indentation.)

Correction : dans le prélude on se donne

```
let stack = ref [0]  (* pile pour l'indentation *)
let rec unindent n = match !stack with
  | m :: _ when m = n -> []
  | m :: st when m > n -> stack := st; END :: unindent n
  | _ -> failwith "mauvaise indentation"
```

et le traitement de la fin de ligne s'écrit alors

```
| '\n' (space* '\n')* (space* as s)
    { let n = String.length s in
      match !stack with
        | m :: _ when m < n ->
            stack := n :: !stack;
            [NEWLINE; BEGIN]
        | _ ->
            NEWLINE :: unindent n
    }
```

Question 3 Pour pouvoir être utilisé avec un outil comme ocamlyacc ou menhir, la fonction d'analyse lexicale doit fournir les lexèmes *un par un*, c'est-à-dire être de la forme suivante :

```
val next_token : lexbuf -> token
```

Écrire une telle fonction en utilisant la fonction `next_tokens` précédente.

Correction :

```
let next_token =
  let tokens = Queue.create () in (* tokens à envoyer *)
  fun lb ->
    if Queue.is_empty tokens then begin
      let l = next_tokens lb in
      List.iter (fun t -> Queue.add t tokens) l
    end;
    Queue.pop tokens
```

---

2 Sémantique

On se donne la syntaxe abstraite suivante pour les expressions de Python :

e	::=	n	constante entière
	|	x	variable
	|	$e\ op\ e$	opération arithmétique, $op \in \{+, -, \times, /\}$
	|	$f(e, \ldots, e)$	application
s	::=	`return` $(e, \ldots, e)$	retour de fonction
	|	$(x, \ldots, x)$ = $(e, \ldots, e)$	affectation simultanée
	|	`if` $e\ c\ e$ `then` s `else` s	conditionnelle, $c \in \{=, \neq, <, \leq, >, \geq\}$
	|	`begin` $s \ldots s$ `end`	bloc
d	::=	`def` f`(`$x, \ldots, x$`)` s	définition de fonction
p	::=	$d \ldots d$	programme

Le langage Python est muni d'une sémantique d'appel par valeur *i.e.* les arguments d'une fonction sont évalués avant l'appel. On suppose d'autre part que, dans une affectation simultanée, les n variables du membre gauche sont distinctes.

Question 4 L'ordre d'évaluation (des arguments d'une fonction ou des éléments d'un n-uplet dans l'instruction `return` ou l'affectation) importe-t-il ? Pourquoi ?

Correction : Il n'importe pas car la seule expression ayant des effets est l'appel de fonction, et un appel n'a pas d'effet observable (il n'y a pas de variables globales, ni d'instruction d'entrée/sortie).

On souhaite donner une sémantique à grands pas pour Python. Pour cela, on distingue la notion de *valeur* d'une expression, notée v et limitée ici à un n-uplet de constantes entières, et celle de *résultat* d'une instruction, noté r et valant soit `none` pour une instruction ne donnant pas de `return`, soit une valeur v sinon. On se donne également une notion d'état S, qui associe à toute variable une valeur. Enfin on suppose que le programme est formé d'un ensemble de fonctions `def` f_i`(`$x_1, \ldots, x_{n_i}$`)` s_i fixé.

Question 5 Donner les règles d'inférence définissant d'une part la relation $S, e \xrightarrow{v} v$ indiquant que l'expression e s'évalue en la valeur v dans l'état S, et d'autre part la relation $S, s \xrightarrow{r} S', r$ indiquant que l'instruction s donne, en partant de l'état S, le résultat r et l'état final S'.

Correction :

$$\frac{}{S, n \xrightarrow{v} n} \qquad \frac{}{S, x \xrightarrow{v} S(x)} \qquad \frac{S, e_1 \xrightarrow{v} n_1 \quad S, e_2 \xrightarrow{v} n_2}{S, e_1\ op\ e_2 \xrightarrow{v} op(n_1, n_2)}$$

pour un appel de fonction, on déplie sa définition :

$$\frac{S, e_i \xrightarrow{v} n_i \quad S[x_i \leftarrow n_i], s_j \xrightarrow{r} S', v}{S, f_j(e_1, \ldots, e_{n_j}) \xrightarrow{v} v}$$

pour `return` et l'affectation, on distingue le cas d'une unique expression dont la valeur est un n-uplet :

$$\frac{S, e \xrightarrow{v} v}{S, \texttt{return}\ e \xrightarrow{r} v} \qquad \frac{S, e_i \xrightarrow{v} n_i \quad m \geq 2}{S, \texttt{return}\ (e_1, \ldots, e_m) \xrightarrow{r} S, (n_1, \ldots, n_m)}$$

$$\frac{S, e \xrightarrow{v} v \quad v = (n_1, \ldots, n_m)}{S, (x_1, \ldots, x_m) = e \xrightarrow{r} S[x_i \leftarrow n_i], \texttt{none}} \qquad \frac{S, e_i \xrightarrow{v} n_i \quad m \geq 2}{S, (x_1, \ldots, x_m) = (e_1, \ldots, e_m) \xrightarrow{r} S[x_i \leftarrow n_i], \texttt{none}}$$

pour le `if`, il faut faire deux cas (une branche pourrait ne pas terminer) :

$$\frac{S, e_1 \xrightarrow{v} n_1 \quad S, e_2 \xrightarrow{v} n_2 \quad n_1\ c\ n_2 = \texttt{true} \quad S, s_1 \xrightarrow{r} S', r}{S, \texttt{if}\ e_1\ c\ e_2\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \xrightarrow{r} S', r}$$

$$\frac{S, e_1 \xrightarrow{v} n_1 \quad S, e_2 \xrightarrow{v} n_2 \quad n_1\ c\ n_2 = \texttt{false} \quad S, s_2 \xrightarrow{r} S', r}{S, \texttt{if}\ e_1\ c\ e_2\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \xrightarrow{r} S', r}$$

pour le bloc, il faut s'arrêter à la première instruction dont le résultat n'est pas `none` :

$$\frac{}{S, \texttt{begin end} \xrightarrow{r} S, \texttt{none}} \qquad \frac{S, \texttt{begin}\ s_1 \ldots s_{m-1}\ \texttt{end} \xrightarrow{r} S', \texttt{none} \quad S', s_m \xrightarrow{r} S'', r}{S, \texttt{begin}\ s_1 \ldots s_m\ \texttt{end} \xrightarrow{r} S'', r}$$

$$\frac{S, \texttt{begin}\ s_1 \ldots s_{m-1}\ \texttt{end} \xrightarrow{r} S', r \quad r \neq \texttt{none}}{S, \texttt{begin}\ s_1 \ldots s_m\ \texttt{end} \xrightarrow{r} S', r}$$

---

3 Typage

Dans cette partie, on souhaite effectuer quelques vérifications de typage sur les programmes Python [1]. On se donne les types Caml suivants pour représenter la syntaxe abstraite de Python :

```
type binop = Add | Sub | Mul | Div
type cmp = Eq | Neq | Lt | Le | Gt | Ge

type expr =
  | Cst of int
  | Var of string
  | Binop of binop * expr * expr
  | Call of string * expr list
```

```
type stmt =
  | If of (expr * cmp * expr) * stmt * stmt
  | Return of expr list
  | Assign of string list * expr list
  | Block of stmt list

type def = string * string list * stmt

type program = def list
```

On se donne d'autre part l'exception suivante pour signaler toute erreur à l'analyse sémantique :

```
exception SemError of string
```

(On ne se soucie pas ici de localiser les erreurs.)

1. Le langage Python est en fait un langage typé *dynamiquement* mais le fragment considéré ici est suffisamment simple pour être typé *statiquement*.

Question 6 En premier lieu, on souhaite tester la bonne utilisation de l'instruction `return` dans le corps de chaque fonction, c'est-à-dire :

1. toute exécution parvient à une instruction `return` ;
2. aucune instruction n'est située au delà d'une instruction `return` (instruction inatteignable).

Ainsi les deux programmes suivants ne sont pas corrects :

```
def f(x):
  if x==0:
    return 0
  else:
    x = 1
```

```
def f(x):
  if x>=1:
    return 1
  return 2
  x = 2
```

car le premier ne contient pas de `return` dans la branche `else` et le second contient une instruction inatteignable (`x = 2`).

Écrire une fonction `val check_return : stmt -> bool` qui, pour une instruction s quelconque, détermine s'il y a effectivement une instruction `return` dans chaque branche du flot de contrôle de s. Dans le même temps, cette fonction devra lever l'exception `SemError` si s contient une instruction inatteignable.

Correction :

```
let rec check_return = function
  | Assign _ ->
      false
  | Return _ ->
      true
  | If (_, s1, s2) ->
      check_return s1 && check_return s2
  | Block sl ->
      let rec check_block = function
        | [] -> false (* rappel : pas de else = bloc vide *)
        | [s] -> check_return s
        | s :: _ when check_return s -> raise (Error "code inatteignable")
        | _ :: sl -> check_block sl
      in
      check_block sl
```

Dans toute la suite, on suppose avoir effectué cette vérification.

Question 7 Dans cette question, on souhaite garantir que toute variable a bien été introduite avant d'être utilisée. Une variable est introduite soit comme paramètre de fonction, soit comme membre gauche d'une affectation. Sa portée est dynamique (*i.e.* c'est l'exécution qui détermine si une variable a été introduite) et s'étend jusqu'à la fin de la fonction. Ainsi dans la fonction `div_eucl` donnée page **??**, les variables `q` et `r` sont introduites par l'affectation `(q,r) = div_eucl(a - b, b)`, ce qui justifie leur utilisation à la ligne suivante. De même, la fonction suivante est correcte

```
def f(x):
  if x==0: y = 1
  else: y = 2
  return y
```

car la variable `y` est bien introduite quel que soit le flot de contrôle emprunté. En revanche, les programmes suivants sont incorrects car susceptibles à chaque fois d'utiliser une variable non introduite :

```
def f(x):
  return f(y)
```

```
def f(x):
  if x==1:
    y = 2
  return y
```

```
def f(x):
  (x,y) = (y, x)
  return 0
```

On se donne le module suivant pour représenter un ensemble de variables :

```
module V = Set.Make(String)
```

Écrire une fonction `expr : V.t -> expr -> unit` qui prend en arguments un ensemble v de variables et une expression e et vérifie que toutes les variables utilisées dans e apparaissent bien dans v, et lève l'exception `SemError` sinon. Écrire ensuite une fonction `stmt : V.t -> stmt -> V.t` qui prend en arguments un ensemble v de variables et une instruction s et vérifie la bonne utilisation des variables dans s vis-à-vis de v. Cette fonction renvoie l'union de v et des variables introduites par s, le cas échéant.

Correction :

```
let rec expr vars = function
  | Cst _ ->
      ()
  | Var x ->
      if not (V.mem x vars) then
        raise (Error ("variable " ^ x ^ " non définie"))
  | Binop (_, e1, e2) ->
      expr vars e1;
      expr vars e2
  | Call (_, el) ->
      List.iter (expr vars) el

let rec stmt vars = function
  | If ((e1, _, e2), s1, s2) ->
      expr vars e1;
      expr vars e2;
      let vars1 = stmt vars s1 in
      let vars2 = stmt vars s2 in
      V.inter vars1 vars2
  | Return el ->
      List.iter (expr vars) el;
      vars
  | Assign (xl, el) ->
      List.iter (expr vars) el;
      List.fold_right V.add xl vars
  | Block sl ->
      List.fold_left stmt vars sl
```

Dans toute la suite, on suppose avoir effectué cette vérification.

Question 8 Enfin, on souhaite vérifier le respect des arités dans les expressions, les instructions `return` et les affectations. On rappelle que toutes les variables contiennent des entiers. Cependant, si un n-uplet ($n \geq 2$) ne peut être stocké dans une variable, il peut être immédiatement déstructuré par une affectation (comme dans `(q,r) = div_eucl(a - b, b)` dans l'exemple page ??) ou passé en argument à une fonction attendant un n-uplet (comme dans `f(div_eucl(100, 17))` pour une fonction `f` attendant une paire en argument). On suppose que tout appel de fonction fait référence à la fonction en cours de définition (fonction récursive) ou à une fonction préalablement définie. La difficulté tient au fait que l'arité d'entrée d'une fonction (nombre d'arguments) est connu mais pas l'arité de sortie (nombre de résultats). Il faut donc calculer cette arité de sortie avant, ou en même temps, que l'on effectue les vérifications de typage.

Écrire une fonction `typage : program -> unit` qui vérifie qu'un programme est bien typé, et lève l'exception `SemError` sinon. On pourra introduire des fonctions intermédiaires si nécessaire (en les spécifiant clairement, le cas échéant).

Correction :

```
(* nombre de paramètres (en particulier, indique si la fonction existe)  *)
let nparams = Hashtbl.create 17

(* arité du résultat *)
let arity = Hashtbl.create 17

(* vérifie une expression et renvoie son arité *)
let rec expr = function
  | Cst _ | Var _ ->
      1
  | Binop (_, e1, e2) ->
      expr1 e1;
      expr1 e2;
      1
  | Call (f, el) ->
      try
        let n = Hashtbl.find nparams f in
        let m = tuple el in
        if m = 0 then (* dans ce cas, el = f(...) *)
          Hashtbl.add arity f n
        else if m <> n then
          raise (Error ("nombre de paramètres de " ^ f ^ " non respecté"));
        Hashtbl.find arity f
      with Not_found ->
        raise (Error ("fonction " ^ f ^ " non définie"))

(* vérifie qu'une expression est un entier *)
and expr1 e =
  let n = expr e in
  if n > 1 then raise (Error ("expression entière attendue"))

(* nombre d'éléments d'un tuple *)
and tuple = function
  | [e] -> expr e
  | el -> List.iter expr1 el; List.length el
```

```
let rec stmt f = function
  | If ((e1, _, e2), s1, s2) ->
      expr1 e1;
      expr1 e2;
      stmt f s1;
      stmt f s2
  | Return el ->
      let m = tuple el in
      let n = Hashtbl.find arity f in
      if n = 0 then
        Hashtbl.add arity f m
      else if n <> m then
        raise (Error ("arité de " ^ f ^ " non respectée"))
  | Assign (xl, el) ->
      let n = List.length xl in
      let m = tuple el in
      if m = 0 then (* dans ce cas, el = f(...) *)
        Hashtbl.add arity f n
      else if n <> m then
        raise (Error ("arité non respectée dans affectation multiple"))
  | Block sl ->
      List.iter (stmt f) sl

let check_def (f, vl, s) =
  Hashtbl.add nparams f (List.length vl);
  Hashtbl.add arity f 0;
  stmt f s

let program = List.iter check_def
```

Dans toute la suite, on suppose avoir calculé les arités des fonctions et effectué les vérifications de typage ci-dessus. L'arité (n, m) d'une fonction Python f est notée $\texttt{int}^n \to \texttt{int}^m$, pour signifier qu'elle reçoit un n-uplet et renvoie un m-uplet. On suppose qu'une table globale fournit l'arité de chaque fonction, sous la forme de la fonction Caml suivante :

```
val arité : string -> int * int
```

4 Production de code

On s'intéresse enfin à la compilation de Python vers l'assembleur MIPS (un aide-mémoire MIPS est donné à la fin de ce sujet). On se propose d'adopter des conventions d'appel différentes des conventions usuelles[2] : les 6 premiers arguments seront passés dans les registres `$a0`, `$a1`, `$a2`, `$a3`, `$v0` et `$v1`, dans cet ordre, et les suivants sur la pile, le cas échéant ; symétriquement, les 6 premiers résultats seront passés dans ces mêmes registres et les suivants sur la pile. Ainsi la fonction `div_eucl` recevra ses deux arguments `a` et `b` dans les registres `$a0` et `$a1` et renverra ses deux résultats dans ces *mêmes* registres `$a0` et `$a1`.

2. On rappelle que les conventions usuelles consistent à passer les 4 premiers arguments dans `$a0`, `$a1`, `$a2` et `$a3` — et les autres sur la pile — et les résultats dans `$v0` et `$v1`.

Question 9 Quel intérêt voyez-vous à ces conventions d'appel ?

Correction : Dans une composition de fonctions f(g(x1...xn)) les arguments de f se trouvent déjà dans les bons registres.

Question 10 Donner le code MIPS pour la fonction `div_eucl` (page **??**), pour les conventions d'appel ci-dessus.

Correction :

```
div_eucl:
        sub     $sp, $sp, 4
        sw      $ra, 0($sp)
        bge     $a0, $a1, L1
        ## a < b
        move    $a1, $a0
        li      $a0, 0
        b       L2
        ## a > = b
L1:
        sub     $a0, $a0, $a1
        jal     div_eucl
        addi    $a0, 1
        ## return
L2:     lw      $ra, 0($sp)
        add     $sp, $sp, 4
        jr      $ra
```

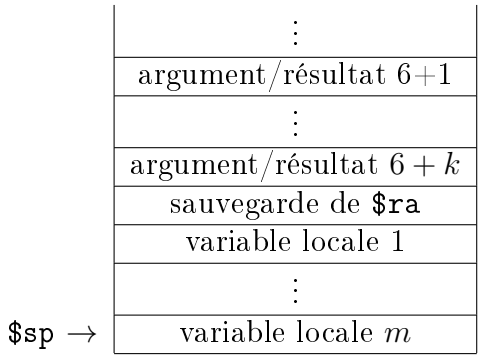
Question 11 De même, donner le code MIPS pour la fonction `is_prime` (page **??**), en optimisant l'appel terminal.

Correction :

```
is_prime:
        mul     $t0, $a0, $a0
        ble     $t0, $a1, L1
        ## return 1
        li      $a0, 1
        jr      $ra
        ##  d*d <= n
L1:     div     $t0, $a1, $a0
        mul     $t0, $t0, $a0
        bne     $t0, $a1, L2
        ## return 0
        li      $a0, 0
        jr      $ra
        ## d * (n/d) == n
```

```
L2:     addi    $a0, 1
        j       is_prime
```

De manière générale, pour une fonction f d'arité $\texttt{int}^n \to \texttt{int}^m$, la taille prise sur la pile par ses arguments et ses résultats vaut $k = \max(0, \max(n, m) - 6)$ et son tableau d'activation prend la forme suivante



où les m variables locales contiennent les calculs intermédiaires qui n'ont pu être stockés dans des registres physiques.

Question 12 Expliquer pourquoi il n'est pas nécessaire de conserver, dans ce tableau d'activation, de pointeur vers le tableau d'activation précédent (l'appelant).

Correction : Les règles de portée ne permettent d'accéder qu'aux variables locales (pas de fonctions imbriquées).

Question 13 Expliquer à quelle condition il est nécessaire de sauvegarder la valeur de `$ra` dans le tableau d'activation. Écrire une fonction `sauvegarde_ra : def -> bool` qui détermine s'il est nécessaire de sauvegarder `$ra` pour une fonction donnée.

Correction : Il est nécessaire de sauvegarder `$ra` si f fait un appel à elle-même ou à une autre fonction ; il suffit donc de parcourir les instructions et les expressions à la recherche d'un appel éventuel.
Si on choisit par ailleurs d'optimiser l'appel terminal, il faut qu'un tel appel soit non-terminal.

```
(* itère f sur tous les appels ; le booléen t indique un appel terminal *)
let rec expr f t = function
  | Cst _ | Var _ -> ()
  | Binop (_, e1, e2) -> expr f false e1; expr f false e2
  | Call (g, el) -> List.iter (expr f false) el; f t g el

(* itère f sur tous les appels *)
let rec stmt f = function
  | If ((e1, _, e2), s1, s2) ->
      expr f false e1; expr f false e2; stmt f s1; stmt f s2
  | Return [e] -> expr f true e
```

```
  | Assign (_, el) | Return el -> List.iter (expr f false) el
  | Block sl -> List.iter (stmt f) sl

let sauvegarde_ra s =
  let leaf = ref true in
  stmt (fun t _ _ -> if not t then leaf := false) s;
  !leaf
```

(Si on ne souhaite pas faire l'optimisation de l'appel terminal, il suffit de ne pas prendre `t` en compte dans la dernière fonction.)

Question 14 Afin d'affiner l'analyse de durée de vie, on souhaite calculer pour chaque fonction le sous-ensemble des 6 registres $\{\texttt{\$a0}, \texttt{\$a1}, \texttt{\$a2}, \texttt{\$a3}, \texttt{\$v0}, \texttt{\$v1}\}$ qu'un appel à cette fonction est susceptible de modifier (comme ce n'est pas décidable, on va calculer une sur-approximation de cet ensemble, la plus petite possible).

Écrire une fonction `effet : def -> int` qui calcule, pour une fonction f donnée, un entier k tel que seuls les k premiers registres de la liste $[\texttt{\$a0}; \texttt{\$a1}; \texttt{\$a2}; \texttt{\$a3}; \texttt{\$v0}; \texttt{\$v1}]$ sont susceptibles d'être modifiés par un appel à f. On pourra supposer avoir déjà fait ce calcul pour les fonctions précédentes.

Correction : on réutilise la fonction `expr` précédente

```
let effects = Hashtbl.create 17

let effect (f, xl, s) =
  Hashtbl.add effects f 0;
  let add_effect s =
    let ef = Hashtbl.find effects f in
    Hashtbl.replace effects f (max ef s)
  in
  let call _ g el =
    let _, ng = arité g in
    add_effect (min (max ng (List.length el)) 6)
  in
  let rec stmt = function
    | If ((e1, _, e2), s1, s2) ->
        expr call false e1; expr call false e2; stmt s1; stmt s2
    | Return el ->
        List.iter (expr call false) el;
        add_effect (min (List.length el) 6)
    | Assign (_, el) ->
        List.iter (expr call false) el
    | Block sl ->
        List.iter stmt sl
  in
  stmt s;
  Hashtbl.find effects f
```

Question 15 Expliquer comment le calcul de la question précédente permet d'améliorer l'allocation de registres.

Correction : Dans le calcul des variables vivantes (*liveness*), on a une information plus précise, au lieu de supposer tous les *caller-saved* écrasés par l'appel. On aura donc moins de registres à sauvegarder lors d'un appel.

Annexe : aide-mémoire MIPS

On donne ici un fragment du jeu d'instructions MIPS suffisant pour réaliser ce problème. (Vous êtes cependant libre d'utiliser tout autre élément de l'assembleur MIPS.) Les instructions susceptibles d'être utiles sont les suivantes (où les r_i désignent des registres, n une constante entière et L une étiquette de code) :

`li`	r_1, n	charge la constante n dans le registre r_1
`addi`	r_1, r_2, n	calcule la somme de r_2 et n dans r_1
`add`	r_1, r_2, r_3	calcule la somme de r_2 et r_3 dans r_1 (on a de même `sub`, `mul` et `div`)
`move`	r_1, r_2	copie le registre r_2 dans le registre r_1
`lw`	r_1, n(`(`r_2`)`)	charge dans r_1 la valeur contenue en mémoire 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
`beq`	r_1, r_2, L	saute à l'adresse désignée par l'étiquette L si $r_1 = r_2$ (on a de même `bne`, `blt`, `ble`, `bgt` et `bge`)
`jr`	r_1	saute à l'adresse contenue dans le registre r_1
`j`	L	saute à l'adresse désignée par l'étiquette L
`jal`	L	saute à l'adresse désignée par l'étiquette L, après avoir sauvegardé l'adresse de retour dans `$ra`