École Normale Supérieure

Langages de programmation et compilation

examen 2010–2011

Jean-Christophe Filliâtre

20 janvier 2010

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.

Les deux problèmes sont indépendants.

1 Analyse d'échappement

Dans ce problème, on considère un petit langage pour manipuler des entiers et des paires, appelé $\mathcal{L}$ par la suite. Un programme $\mathcal{L}$ est un ensemble de définitions de fonctions mutuellement récursives. Chaque fonction a un unique argument et un corps qui est une expression. Les expressions sont formées à partir de constantes entières, de variables, des deux opérations arithmétiques $+$ et $-$, d'une construction `let in` pour introduire une variable locale, d'une construction `letp in` pour construire une paire, des deux projections `fst` et `snd`, d'une conditionnelle et enfin de l'appel de fonction. On se donne la syntaxe abstraite suivante pour les expressions de $\mathcal{L}$:

e	::=	n	constante entière
	|	x	variable
	|	$e\ op\ e$	opération arithmétique $op \in \{+, -\}$
	|	`let` $x = e$ `in` e	variable locale
	|	`letp` $x = (e, e)$ `in` e	construction d'une paire
	|	`fst` e	première projection
	|	`snd` e	seconde projection
	|	`if` $e < e$ `then` e `else` e	conditionnelle
	|	$f\ e$	application de la fonction f

La sémantique de $\mathcal{L}$ coïncide en tout point avec le fragment de Caml que ce langage représente.

Voici par exemple la division euclidienne écrite dans ce langage. L'argument x est une paire d'entiers (a, b) supposés vérifier $0 \leq a$ et $0 < b$. Le résultat est la paire d'entiers constituée du quotient et du reste de la division euclidienne de a par b.

```
def div (x : int * int) : int * int =
  let a = fst x in
  let b = snd x in
  if a < b then
    letp r = (0, a) in
    r
  else
    letp y  = (a - b, b) in
    let  z  = div y in
    letp u = (fst z + 1, snd z) in
    u
```

Dans ce qui suit, on suppose donné un programme constitué de n fonctions $f_1, \ldots, f_n$.

Question 1 On souhaite munir $\mathcal{L}$ d'une sémantique opérationnelle à grands pas. Une valeur v est soit un entier n, soit une paire (v_1, v_2) où v_1 et v_2 sont des valeurs. Une valeur n'étant pas directement une expression de $\mathcal{L}$, on ne va pas chercher à définir la sémantique à grands pas à partir d'une opération de substitution des valeurs dans les expressions. Au lieu de cela, on va utiliser une fonction associant à toute variable sa valeur, appelée *valuation*. Plus précisément, si e est une expression et V une fonction associant à toute variable libre de e une valeur, alors le jugement $V \vdash e \Rightarrow v$ signifie « dans la valuation V, l'évaluation de e termine, sur la valeur v ».

Donner les règles d'inférence définissant la relation $V \vdash e \Rightarrow v$.

Correction : cf le code d'`eval` dans les questions suivantes

Question 2 On souhaite programmer en Caml un interprète pour $\mathcal{L}$ suivant la définition de $V \vdash e \Rightarrow v$. On suppose défini un type Caml `expr` pour la syntaxe abstraite des expressions de $\mathcal{L}$. Donner un type Caml pour les valeurs, pour les valuations et pour la fonction `eval` correspondant au jugement $V \vdash e \Rightarrow v$.

Correction :

```
type value = Vint of int | Vpair of value * value

module M = Map.Make(String)

val eval : value M.t -> expr -> value
```

Question 3 Donner le code de la fonction `eval` correspondant aux cas (1) d'une variable, (2) de la construction `letp`, (3) de la construction `fst` et (4) de l'appel de fonction. On pourra supposer que les fonctions constituant le programme sont contenues dans une table de hachage globale `defs`.

Correction :

```
type def = {
  arg    : string;
  body   : expr;
}

let defs = Hashtbl.create 17

exception Stuck

let binop b v1 v2 = match b, v1, v2 with
  | Bplus,  Vint n1, Vint n2 -> Vint (n1 + n2)
  | Bminus, Vint n1, Vint n2 -> Vint (n1 - n2)
  | _ -> raise Stuck

let cmp c v1 v2 = match c, v1, v2 with
  | Ceq, Vint n1, Vint n2 -> n1 = n2
  | Clt, Vint n1, Vint n2 -> n1 < n2
```

```
  | _ -> raise Stuck

let rec eval env = function
  | Econst n ->
      Vint n
  | Evar x ->
      M.find x env
  | Elet (x, e1, e2) ->
      eval (M.add x (eval env e1) env) e2
  | Eletpair (x, e1, e2, e3) ->
      eval (M.add x (Vpair (eval env e1, eval env e2)) env) e3
  | Efst e1 -> begin match eval env e1 with
      | Vpair (v1, _) -> v1
      | Vint _ -> raise Stuck end
  | Esnd e1 -> begin match eval env e1 with
      | Vpair (_, v2) -> v2
      | Vint _ -> raise Stuck end
  | Ebinop (b, e1, e2) ->
      binop b (eval env e1) (eval env e2)
  | Eif (c, e1, e2, e3, e4) ->
      if cmp c (eval env e1) (eval env e2) then eval env e3 else eval env e4
  | Ecall (f, e1) ->
      let f = Hashtbl.find defs f in
      eval (M.add f.arg (eval env e1) M.empty) f.body
```

Typage. On munit $\mathcal{L}$ de types simples, de la forme

$$\tau ::= \texttt{int} \mid \tau \times \tau$$

Chaque fonction f a un type de la forme $\tau_1 \to \tau_2$, que l'on note $\Delta(f)$. Un environnement Γ associe un type à toute variable. Le jugement de typage s'écrit $\Gamma \vdash e : \tau$ et signifie « dans l'environnement Γ, l'expression e a le type τ ». Les règles de typage, immédiates, sont les suivantes :

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_1 : \texttt{int}}{\Gamma \vdash e_1 \; op \; e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x : \tau_1 \times \tau_2 \vdash e_3 : \tau_3}{\Gamma \vdash \texttt{letp } x = (e_1, e_2) \texttt{ in } e_3 : \tau_3}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst } e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd } e : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad \Gamma \vdash e_3 : \tau \quad \Gamma \vdash e_4 : \tau}{\Gamma \vdash \texttt{if } e_1 < e_2 \texttt{ then } e_3 \texttt{ else } e_4 : \tau}$$

$$\frac{\Delta(f) = \tau_1 \to \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash f \; e_1 : \tau_2}$$

Enfin, chaque définition de fonction, de la forme $\texttt{def } f \; (x : \tau_1) : \tau_2 = e$ avec $\Delta(f) = \tau_1 \to \tau_2$, est bien typée si $x : \tau_1 \vdash e : \tau_2$.

Compilation. On s'intéresse maintenant à la compilation de $\mathcal{L}$. Une paire sera représentée en mémoire par deux mots consécutifs (8 octets), contenant les deux composantes de la paire. Parmi les paires allouées par une fonction f, certaines doivent survivre à l'appel à f, et doivent donc être allouées sur le tas, alors que d'autres ne survivent pas à l'appel à f et peuvent donc être avantageusement allouées sur la pile. On va s'intéresser ici au problème consistant à déterminer quelles paires peuvent être allouées sur la pile ; c'est l'*analyse d'échappement*.

On adopte ici une approche par typage, où le type d'une paire est décoré par des étiquettes permettant de tracer son origine. Les types sont donc maintenant de la forme

$$\tau ::= \texttt{int} \mid \tau \times_S \tau$$

où S est un ensemble fini d'étiquettes, les étiquettes étant des identificateurs. L'idée est alors la suivante. La construction `letp` $x = (e_1, e_2)$ `in` e_3 introduit une variable x de type $\tau_1 \times_{\{x\}} \tau_2$, c'est-à-dire un type produit uniquement étiqueté par l'unique étiquette x. De manière générale, un type de la forme $\tau_1 \times_S \tau_2$ indique que la valeur correspondante peut avoir été obtenue à partir de n'importe quelle paire dont le type est étiquetté par un élément de S.

Pour réaliser l'analyse d'échappement d'une fonction f, on va typer le corps de f avec ces types étiquetés. Si τ est le type obtenu, toute étiquette apparaissant dans τ représente une paire pouvant faire partie du résultat renvoyé par f. En particulier, s'il s'agit d'une variable locale à f, elle ne pourra pas être allouée sur la pile.

Une difficulté apparaît si la fonction f appelle une autre fonction g. Un exemple minimal est le suivant :

```
def f (x : int * int) : int * int =
  letp y = (1, 2) in g y
```

Pour déterminer si la paire `y` peut être allouée sur la pile, il faut savoir si la fonction `g` renvoie un résultat dans lequel la paire `y` peut apparaître. Les étiquettes permettent très exactement de déterminer ce flot de données. Pour toute fonction, dont le type est de la forme $\tau_1 \to \tau_2$, le type τ_1 de l'argument verra tous ses types produits étiquetés par des singletons distincts et le type τ_2 du résultat ne contiendra que des étiquettes apparaissant dans τ_1. Il n'est pas nécessaire, en effet, que le type τ_2 mentionne les étiquettes correspondant à des paires allouées par la fonction (ou par des fonctions qu'elle appelle). Ainsi, la fonction identité

```
def id(x : int * int) : int * int =
  x
```

a le type $\texttt{int} \times_{\{x\}} \texttt{int} \to \texttt{int} \times_{\{x\}} \texttt{int}$ alors que la fonction de copie

```
def copy(x : int * int) : int * int =
  letp r = (fst x, snd x) in r
```

a le type $\texttt{int} \times_{\{x\}} \texttt{int} \to \texttt{int} \times_{\emptyset} \texttt{int}$. Dans l'exemple de la fonction `f` ci-dessus, si `g` a le même type que `id` alors la paire `y` ne peut pas être allouée en pile mais si `g` a le même type que `copy` alors elle peut l'être.

Question 4 Donner le type de la fonction `div`.

Correction :

$$\texttt{int} \times_{\{x\}} \texttt{int} \to \texttt{int} \times_{\emptyset} \texttt{int}$$

Question 5 Donner un exemple de fonction du type

$$(\texttt{int} \times_{\{x\}} \texttt{int}) \times_{\{y\}} (\texttt{int} \times_{\{z\}} \texttt{int}) \to \texttt{int} \times_{\{x,z\}} \texttt{int}.$$

Correction :

```
def foo x =
   let x1 = fst x in
   let x2 = snd x in
   if fst x1 < fst x2 then x1 else x2
```

Opérations sur les types. Étant donnés deux types τ_1 et τ_2 de même structure, on définit le type $\tau_1 \sqcup \tau_2$ de la manière suivante :

$$\begin{array}{rcl} \texttt{int} \sqcup \texttt{int} & = & \texttt{int} \\ (\tau_1^1 \times_{S_1} \tau_1^2) \sqcup (\tau_2^1 \times_{S_2} \tau_2^2) & = & (\tau_1^1 \sqcup \tau_2^1) \times_{S_1 \cup S_2} (\tau_1^2 \sqcup \tau_2^2) \end{array}$$

Soit σ une fonction des étiquettes vers les ensembles d'étiquettes. Si S est un ensemble d'étiquettes, on définit $\sigma(S)$ par

$$\sigma(S) = \bigcup_{x \in S} \sigma(x).$$

D'autre part, pour un type étiqueté τ, on définit $\sigma(\tau)$ de la manière suivante :

$$\begin{array}{rcl} \sigma(\texttt{int}) & = & \texttt{int} \\ \sigma(\tau_1 \times_S \tau_2) & = & \sigma(\tau_1) \times_{\sigma(S)} \sigma(\tau_2). \end{array}$$

Question 6 Donner les nouvelles règles de typage pour les types étiquetés. Le jugement a toujours la forme $\Gamma \vdash e : \tau$ où Γ associe à toute variable un type étiqueté et où τ est également un type étiqueté. On supposera que Δ donne le type étiqueté de chaque fonction. (On ne cherche pas ici à faire d'inférence de types.)

Correction :

```
let rec typing env = function
  | Econst n ->
      Tint
  | Evar x ->
      M.find x env
  | Elet (x, e1, e2) ->
      typing (M.add x (typing env e1) env) e2
  | Eletp (x, e1, e2, e3) ->
      let ty = Tpair (typing env e1, S.singleton x, typing env e2) in
      typing (M.add x ty env) e3
  | Efst e1 -> begin match typing env e1 with
      | Tpair (ty1, _, _) -> ty1
      | Tint -> raise TypingError end
  | Esnd e1 -> begin match typing env e1 with
      | Tpair (_, _, ty2) -> ty2
      | Tint -> raise TypingError end
```

```
  | Ebinop (_, e1, e2) -> begin match typing env e1, typing env e2 with
      | Tint, Tint -> Tint
      | _ -> raise TypingError end
  | Eif (e1, e2, e3, e4) -> begin match typing env e1, typing env e2 with
      | Tint, Tint -> sup (typing env e3) (typing env e4)
      | _ -> raise TypingError end
  | Ecall (f, e1) ->
      let f = Hashtbl.find deftyps f in
      let s = unify f.targ (typing env e1) in
      apply s f.tresult
```

Question 7 En Caml, on représente les étiquettes par le type `string`, les ensembles d'étiquettes par le type `S.t` et les dictionnaires indexés par des étiquettes par le type `M.t`, où les modules `S` et `M` sont définis par

```
module S = Set.Make(String)
module M = Map.Make(String)
```

On se donne alors le type Caml suivant pour les types étiquetés :

```
type typ = Tint | Tpair of typ * S.t * typ
```

Écrire une fonction `unify : typ -> typ -> S.t M.t` qui prend en arguments deux types τ_1 et τ_2, en supposant que les produits de τ_1 sont tous étiquetés par des singletons, et renvoie, si elle existe, une substitution σ telle que $\sigma(\tau_1) = \tau_2$.

Correction :

```
let unify ty1 ty2 =
  let rec unify s = function
    | Tint, Tint ->
        s
    | Tpair (ty11, s1, ty12), Tpair (ty21, s2, ty22) ->
        assert (S.cardinal s1 = 1);
        let x1 = S.choose s1 in
        let s = M.add x1 s2 s in
        unify (unify s (ty11, ty21)) (ty12, ty22)
    | _ ->
        raise TypingError
  in
  unify M.empty (ty1, ty2)
```

Question 8 Écrire une fonction `apply : S.t M.t -> typ -> typ` qui prend en arguments σ et τ et renvoie $\sigma(\tau)$. On considérera que σ vaut l'identité sur toutes les étiquettes pour lesquelles elle n'est pas définie.

Correction :

```
let rec apply s = function
  | Tint ->
      Tint
  | Tpair (ty1, ls, ty2) ->
      let ls =
        S.fold
          (fun x ls ->
             let sx = try M.find x s with Not_found -> S.singleton x in
             S.union sx ls)
          ls S.empty
      in
      Tpair (apply s ty1, ls, apply s ty2)
```

Question 9 En utilisant les fonctions `unify` et `apply`, donner le code Caml correspondant au typage d'une application de fonction.

Correction : (déjà donné plus haut)

```
  | Ecall (f, e1) ->
      let f = Hashtbl.find deftyps f in
      let s = unify f.targ (typing env e1) in
      apply s f.tresult
```

Question 10 Pour la fonction `div` donnée plus haut, indiquer quelles sont les paires qui peuvent être allouées sur la pile.

Correction : seulement la paire `letp y = (a - b, b) in`

Question 11 Donner le code MIPS de la fonction `div`, en allouant sur la pile les paires qui peuvent l'être.

Correction :

```
diveucl:lw      $t0, 0($a0)     # ($t0, $t1) = $a0
        lw      $t1, 4($a0)
        blt     $t0, $t1, Ldone
        addi    $sp, -12        # alloue y = ($t0-$t1, $t1) en pile
        sw      $ra, 8($sp)     # sauve $ra
        sub     $t2, $t0, $t1
        sw      $t2, 0($sp)
        sw      $t1, 4($sp)
        move    $a0, $sp
        jal     diveucl
        lw      $t0, 0($v0)     # ($t0, $t1) = div y
        lw      $t1, 4($v0)
        li      $a0, 8          # alloue et renvoie ($t0+1, $t1)
```

```
        li      $v0, 9
        syscall
        addi    $t0, 1
        sw      $t0, 0($v0)
        sw      $t1, 4($v0)
        lw      $ra, 8($sp)
        addi    $sp, 12         # restaure $ra
        jr      $ra
Ldone:  li      $a0, 8          # alloue et renvoie (0, $t0)
        li      $v0, 9
        syscall
        sw      $zero, 0($v0)
        sw      $t0, 4($v0)
        jr      $ra
```

Question 12 Expliquer pourquoi l'analyse d'échappement devient plus difficile si le langage autorise les effets de bord, par exemple sous la forme de références. Donner un exemple.

Correction : Une paire allouée localement peut être stockée dans une référence. Elle échappe alors à la fonction, sans pour autant faire partie de son résultat.

---

2 Coroutines

On introduit ici la notion de *coroutine* dans le cadre du langage assembleur MIPS. Les coroutines généralisent la notion usuelle de *fonction*.

Les fonctions obéissent à une logique d'appel et de retour organisée autour d'une pile. Si une fonction f appelle une fonction g, alors l'exécution de f reprendra lorsque l'exécution de g sera terminée. Lorsque l'exécution de f sera à son tour terminée, c'est l'exécution de la fonction ayant appelée f qui reprendra, et ainsi de suite. C'est pour cela que l'on peut, et que l'on doit, utiliser une *pile d'appels*.

Les coroutines obéissent à une logique d'appel différente. Si une coroutine A appelle une coroutine B, alors l'exécution de A ne reprendra que *lorsque A sera appelée à nouveau*, par B ou par une autre coroutine. L'exécution de A reprendra là où elle avait été interrompue, juste après l'appel à B. Il n'y a pas de notion de fin d'exécution pour une coroutine (si ce n'est la fin de l'exécution du programme tout entier, le cas échéant). En particulier, la coroutine B peut à son tour appeler A, qui rappellera B, et ainsi de suite. La notion d'appel pour les coroutines est complètement symétrique, alors qu'elle ne l'est pas pour les fonctions (différence appelant/appelé).

Un exemple typique est celui d'un modèle producteur / consommateur. Deux programmes potentiellement complexes collaborent. L'un produit des données que le second consomme au fur et à mesure de leur production (qui peut être infinie). Les deux programmes sont naturellement deux coroutines. Le producteur appelle le consommateur dès lors qu'il produit une donnée, et le consommateur appelle le producteur dès lors qu'il a besoin d'une donnée.

Question 13 Proposer un schéma général pour réaliser des coroutines avec le jeu d'instructions MIPS, en indiquant (1) quelles sont les données associées à chaque coroutine et (2) quelles sont les instructions MIPS correspondant à l'appel de la coroutine B par la coroutine A.

Correction : À chaque coroutine C est associée une ou plusieurs données propres à cette coroutine. On y trouve évidemment l'adresse où doit reprendre son calcul au prochain appel. Mais on peut y trouver également des variables locales à la coroutines et/ou des sauvegardes de registres, exactement comme dans un tableau d'activation. Plus subtilement encore, on peut y trouver une pile propre à la coroutine, si son traitement est par nature récursif, etc.

Ces données peuvent être stockées dans des registres spécifiques à la coroutine et/ou en mémoire (par exemple sur le segment de données). Soit j_C l'emplacement contenant l'adresse où doit reprendre le calcul de la coroutine C. Initialement, j_C est positionnée au début du code de C.

Quand une coroutine A appelle une coroutine B, elle doit sauvegarder l'adresse située immédiatement après le point d'appel dans j_A, puis saute à l'adresse indiquée par j_B. On peut le faire astucieusement avec `jal`. Quand A appelle B, on fait juste

```
    jal  A_call_B
```

avec

```
A_call_B:
    move $jA, $ra  # sauvegarde l'exécution de A
    jr   $jB       # reprend l'exécution de B
```

Question 14 On considère maintenant un exemple précis de producteur / consommateur. Un programme MIPS lit une séquence infinie d'entiers avec l'appel système `read_int`. Le producteur lit ces entiers deux par deux. Lorsqu'il lit l'entier x puis l'entier y, il produit x occurrences de l'entier y. Le consommateur, pour sa part, affiche les entiers produits par le producteur, par lignes de trois. Il n'affiche une ligne que lorsque celle-ci est complète.

Ainsi, si la séquence d'entiers lue jusqu'à présent est

```
1 2 3 3 1 4 5 7 1 8
```

alors la sortie est pour l'instant formée des trois lignes suivantes :

```
2 3 3
3 4 7
7 7 7
```

On notera que la quatrième ligne n'est pas encore affichée car seuls deux entiers sont pour l'instant disponibles (un 5-ième 7 et un 8).

Écrire un programme MIPS pour ce producteur / consommateur en utilisant deux coroutines.

Correction :

```
        .text
main:   la      $t0, producer   # $t0 = reprend producer
        la      $t1, consumer   # $t1 = reprend consumer
        jr      $t0

# lit n et x et produit n occurrences de x
# l'entier produit est dans $v1
producer:
        li      $v0, 5          # read_int
        syscall
        move    $s2, $v0        # $s2 = n
        li      $v0, 5          # read_int
        syscall
        move    $v1, $v0        # $v1 = x
p_loop: beqz    $s2, producer   # n = 0, on reprend du debut
        jal     call_consumer
        addi    $s2, -1
        j       p_loop          # boucle infinie

# indirections pour utiliser astucieusement jal
call_consumer:
        move    $t0, $ra
        jr      $t1
call_producer:
        move    $t1, $ra
        jr      $t0

# consomme les entiers et les affiche trois par ligne
consumer:
        move    $s3, $v1
        jal     call_producer
```

```
move    $s4, $v1
jal     call_producer
move    $s5, $v1
move    $a0, $s3
li      $v0, 1
syscall
li      $v0, 11
li      $a0, 32
syscall
move    $a0, $s4
li      $v0, 1
syscall
li      $v0, 11
li      $a0, 32
syscall
move    $a0, $s5
li      $v0, 1
syscall
li      $v0, 11
li      $a0, 10
syscall
jal     call_producer
j       consumer        # boucle infinie
```

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
`la`	r_1, L	charge l'adresse de l'étiquette L 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`

Les appels systèmes (`syscall`) utiles ici sont les suivants :

appel	`$v0` (entrée)	`$a0` (entrée)	`$v0` (sortie)
`print_char`	11	caractère à afficher	
`print_int`	1	entier à afficher	
`print_string`	4	pointeur vers la chaîne à afficher	
`read_int`	5		entier lu
`sbrk` (`malloc`)	9	nombre d'octets à allouer	pointeur vers le bloc alloué