Compilation
Master Informatique M1 2011-2012

Examen Final
15 décembre 2011

L'examen dure **3 heures**. Les questions sont largement indépendantes. L'énoncé est composé de 7 pages et comporte en section 4 à partir de la page 8 des rappels sur le code MIPS et les fonctions Ocaml.

Le seul document autorisé est une feuille A4 recto-verso manuscripte

Consigne: Lorsqu'il est demandé de décrire une méthode, vous pouvez utiliser des fonctions auxiliaires dont l'implantation est directe à partir de structures de données standards comme les tables de hachage, ensembles, maps...Vous devez par contre spécifier le comportement de chaque fonction utilisée ou définie. En particulier vous indiquerez pour chaque fonction *le type de ses arguments et de son résultat* ainsi que le comportement attendu.

1 Questions de cours (3 points)

Dire si les affirmations suivantes sont vraies ou fausses et justifier *brièvement* votre réponse:

1. Ocamlyacc indique des conflits dans ma grammaire, je peux en déduire que celle-ci est ambigüe.
2. Ma grammaire est ambigüe. Sans indication de précédences, Ocamlyacc signale la présence de conflits.
3. Je peux évaluer n'importe quelle expression arithmétique (formée à partir de variables globales, de constantes et des quatre opérations arithmétiques) en utilisant les instructions MIPS et 4 registres mais sans la pile.
4. Soit un langage où le début d'un commentaire est défini par la suite de caractères `(*` et la fin par la suite de caractères `*)`, les commentaires peuvent être imbriqués. Il est possible de trouver une expression régulière qui corresponde exactement aux commentaires de mon langage.

Correction :

1. *Faux: l'analyse dans Ocamlyacc construit une dérivation en ne regardant qu'un seul caractère en avance. Une grammaire peut ne pas être ambiguë mais présenter des conflits si on se limite à ce seul caractère.*
2. *Vrai: une grammaire ambigüe admet plusieurs arbres de dérivation pour la même entrée et présentera donc des conflits qui peuvent éventuellement se résoudre par un choix de précédences.*
3. *Faux: l'évaluation d'une expression arithmétique peut demander la sauvegarde d'un nombre arbitraire de valeurs intermédiaires.*
4. *Faux: les commentaires imbriqués comme les expressions bien parenthésées forment un langage qui n'est pas régulier (il faut "compter" pour s'assurer que le bon nombre de parenthèses a été fermé) et ne peut donc être décrit par une expression régulière.*

2 Langage avec des types records (13 points)

On considère RECL, un langage avec des types records pour représenter des structures.

Types Les types de RECL sont
- soit un type de base `int`, `bool` ou `unit`,
- soit un type record $\{\texttt{lab}_1 : \tau_1; \ldots; \texttt{lab}_n : \tau_n\}$ où les noms des champs $\texttt{lab}_i$ sont des identifiants tous distincts et les types des champs τ_i sont des types quelconques.

Deux types records sont considérés comme égaux s'ils ont les mêmes labels (pas forcément dans le même ordre) et que pour chaque label, les types associés dans les deux types records sont égaux.

Expressions Les expressions du langage sont de l'une des formes suivantes:
- constante: entier de type `int`, booléen `true` ou `false` de type `bool` et `()` de type `unit`;
- variable: une variable globale, locale ou bien un paramètre de fonction;
- un record explicite $\{\texttt{lab}_1 = e_1; \ldots; \texttt{lab}_n = e_n\}$;
- l'accès dans un record $e.\texttt{lab}$ avec `lab` l'une des étiquettes du type record de e;
- la mise à jour d'un champ d'un record $e_1.\texttt{lab} := e_2$ avec `lab` l'une des étiquettes du type record de e_1;
- l'application d'une fonction f à ses arguments $f(e_1, \ldots, e_n)$;
- une déclaration locale **let** $x = e_1$ **in** e_2;
- une conditionnelle **if** b **then** e_1 **else** e_2;
- une séquence $e_1; e_2$, où e_1 a typiquement le type `unit`.

Programmes Un programme est formé d'une suite de déclarations de fonctions de la forme **let** $f\,(x_1 : \tau_1, \ldots, x_n : \tau_n) : \sigma = e$ avec τ_i et σ des types et e une expression dans laquelle f peut apparaître de manière récursive. On suppose dans la suite que les noms de fonctions sont uniques.

Exemple le programme suivant définit la fonction `double` qui modifie un vecteur en le doublant.

```
let double(v:{ dx:int; dy:int }) : unit =
  v.dx := 2 * v.dx ;
  v.dy := 2 * v.dy
```

2.1 Typage (8 points)

1. Compléter les types Ocaml `typ`, `expr` et `decl` pour représenter les arbres de syntaxe abstraite du langage RECL.

```
type ident = string
type typ =
  | Tint | Tbool | Tunit          (* types de base *)
  | Trec of ...                   (* {l_1:t_1;...;l_n:t_n} *)
type cte = Int of int | Bool of bool | Unit

type expr =
  |  Cte of cte | Var of ident
  |  Rec of ...                   (* {l_1=e_1;..;l_n=e_n} *)
  |  Acc of ...                   (* e.l_i *)
  |  Upd of ...                   (* e.l_i := e' *)
  |  App of ...                   (* f(e_1,..,e_n) *)
  |  If of ...                    (* if b then e1 else e2 *)
  |  LetIn of ...                 (* let x = e1 in e2 *)
  |  Seq of ...                   (* e1 ; e2 *)

type decl =
  | Fun of ident * ...     (* let f (x_1:t_1,..,x_n:t_n):t = e *)

type prog = decl list
```

Correction :

```
type typ =  Tint | Tbool | Tunit | Trec of (ident * typ) list
type expr = Cte of cte | Var of ident |  Rec of (ident * expr) list
  |  Acc of expr * ident               |  Upd of expr * ident * expr
  |  App of ident * expr list          |  If of expr * expr * expr
  |  LetIn of ident * expr * expr      |  Seq of expr * expr
type decl = Fun of ident * (ident * typ) list * typ * expr
```

2. On s'intéresse tout d'abord à un typage simple. Les règles pour les variables, constantes, application de fonction, conditionnelles, séquence et déclarations locales sont usuelles. Les règles pour les records sont les suivantes:
   (a) L'accès à un champ $e.l$ est bien typé de type τ si e est bien typé dans un type record t qui possède un label l de type τ;

(b) La mise à jour $e_1.l := e_2$ est bien typée de type `unit` si e_2 est de type τ et e_1 est bien typé dans un type record t qui possède un label l de type τ;

(c) Le record $\{l_1 = e_1; \ldots; l_n = e_n\}$ est bien typé de type $\{l_1 : \tau_1; \ldots; l_n : \tau_n\}$ si chaque expression e_i est bien typée de type τ_i.

Compléter les règles d'inférence suivantes pour le typage :

$$\frac{\ldots}{\Gamma \vdash x : \tau} \qquad \frac{\ldots}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau} \qquad \frac{\ldots}{\Gamma \vdash \textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2 : \tau}$$

$$\frac{\ldots}{\Gamma \vdash e.l : \tau} \qquad \frac{\ldots}{\Gamma \vdash e_1.l := e_2 : \texttt{unit}} \qquad \frac{\ldots}{\Gamma \vdash \{l_1 = e_1; \ldots; l_n = e_n\} : \ldots}$$

Correction :

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 : \tau} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ e_1\ \textbf{\textit{else}}\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1; \ldots; l_n : \tau_n\} \quad l = l_i \quad \tau = \tau_i}{\Gamma \vdash e.l : \tau} \qquad \frac{\Gamma \vdash e_1 : \{l_1 : \tau_1; \ldots; l_n : \tau_n\} \quad l = l_i \quad \Gamma \vdash e_2 : \tau_i}{\Gamma \vdash e_1.l := e_2 : \mathit{unit}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \ldots \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 = e_1; \ldots; l_n = e_n\} : \{l_1 : \tau_1; \ldots; l_n : \tau_n\}}$$

3. On souhaite maintenant implanter la fonction qui vérifie qu'une expression e est bien formée dans un certain environnement Γ et calcule son type τ.

   (a) Expliquer quelles informations doivent être stockées dans la table des symboles afin de pouvoir effectuer les opérations de typage.

   Correction : *On garde dans la table des symboles pour chaque variable son type et pour chaque symbole de fonctoin sa signature (liste des types des arguments et type du résultat).*

   (b) Compléter la fonction de typage pour les records pour les trois cas suivants:

```
exception TypeError
let rec type_expr env e = match e with
  | Rec ... -> ...      | Acc ... -> ...      | Upd ... -> ...
```

Correction : *L'égalité des types record est définie modulo l'ordre des labels. Afin de simplifier le test d'égalité, on peut décider de représenter les types record sous une fomre canonique, par exemple par ordre alphabétique de labels.*

```
let mkTrec labt = Trec (List.sort (fun (l1,_) (l2,_) -> compare l1 l2) labt)

let rec type_expr env e = match e with
 | Rec labe -> mkTrec (List.map (fun (li,ei) -> li,type_expr env ei) labe)
 | Acc (e,l) ->
     (match type_expr env e with
        | Trec labt -> (try List.assoc l labt with Not_found -> raise TypeError)
        | _ -> raise TypeError)
 | Upd(e1,l,e2) ->
     (match type_expr env e1 with
        | Trec labt ->
            let tau = try List.assoc l labt with Not_found -> raise TypeError
            in if type_expr env e2 = tau then Tunit else raise TypeError
        | _ -> raise TypeError)
```

4. On souhaite maintenant avoir un système un peu plus souple dans lequel il y a du sous-typage entre les records. On introduit une relation $\tau \leq \sigma$ entre les types de la manière suivante:
   - si $\tau = \sigma$ alors $\tau \leq \sigma$;
   - sinon, si τ ou σ n'est pas un type record alors $\tau \not\leq \sigma$;
   - un type record τ est plus petit qu'un type record σ si tout label l de σ est aussi un label de τ et si le type du label l dans τ est plus petit que le type du label l dans σ (un type plus petit aura autant ou plus de champs).

Par exemple, avec les types suivants on a `point3d` $\leq$ `point2d` et `seg3d` $\leq$ `seg2d`:

```
point2d = {x:int;y:int}
point3d = {x:int;y:int;z:int}
  seg2d = {left:point2d;right:point2d}
  seg3d = {left:point3d;right:point3d}
```

Toute expression e de type τ peut aussi être considérée comme une expression de type σ si $\tau \leq \sigma$. Le système contient donc une règle de typage :

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \sigma}{\Gamma \vdash e : \sigma}$$

(a) Dire comment les types suivants se comparent à ceux de l'exemple précédent et entre eux :

```
  rgbpoint = {x:int;y:int;color:{r:int;g:int;b:int}}
  colpoint = {x:int;y:int;color:int}
colpoint2d = {p:point2d;color:int}
colpoint3d = {p:point3d;color:int}
```

Correction : *On a* `rgbpoint` $\leq$ `point2d` *et* `colpoint` $\leq$ `point2d` *mais* `rgbpoint` *et* `colpoint` *sont incomparables entre eux. On a aussi* `colpoint3d` $\leq$ `colpoint2d`*.*

(b) Ecrire une fonction `subtype` qui étant donnés deux types τ et σ teste si $\tau \leq \sigma$.
Correction :

```
let rec subtype t1 t2 = match t1 with
  Trec labt1 ->
    (match t2 with
      | Trec labt2 ->
        (List.for_all
           (fun (l,tau2) ->
             try let tau1 = List.assoc l labt1 in subtype tau1 tau2
             with Not_found -> false)
           labt2)
      | _ -> false)
  | _ -> t1 = t2
```

(c) La fonction de typage d'une expression va chercher à calculer le type le plus petit possible pour une expression. Lors de l'application et de la mise à jour d'un champ, au lieu de vérifier l'égalité entre les types, il suffit de vérifier la relation de sous-typage. Par exemple une fonction `coordx` qui attend un argument p de type `point2d` peut être appliquée à une expression q de type `point3d`. De même si s a pour type `seg2d`, l'affectation s.`left` $:= q$ est bien formée.
Compléter les deux cas suivants de la fonction de typage pour prendre en compte le sous-typage:

```
let rec type_expr env e = match e with
  | App ... -> ...     | Upd ... -> ...
```

Correction :

```
let rec type_expr env e = match e with
  | App(f,le) -> let lt,t = find_sign f env in
                 let lt' = List.map (type_expr env) le in
                 (try if List.for_all2 subtype lt' lt then t
                    else raise TypeError
                  with Invalid_argument _ -> raise TypeError)
  | Upd(e1,l,e2) ->
    (match type_expr env e1 with
      | Trec labt ->
        let tau = try List.assoc l labt
                  with Not_found -> raise TypeError
        in if subtype (type_expr env e2) tau then Tunit
           else raise TypeError
      | _ -> raise TypeError)
```

5. Le cas de la conditionnelle est plus complexe, car il faut trouver un type plus grand que le type des deux branches.

   (a) Dire si les fonctions suivantes sont bien typées et si oui compléter le type:

```
let f1(b:bool,p:colpoint,q:point2d):... = if b then p else q
let f2(b:bool,p:colpoint,q:point3d):... = if b then p else q
let f3(b:bool,p:colpoint,q:rgbpoint):... = if b then p else q
let f4(b:bool,p:int,q:bool):... = if b then p else q
```

   Correction :

```
let f1(b:bool,p:colpoint,q:point2d):point2d = if b then p else q
let f2(b:bool,p:colpoint,q:point3d):point2d = if b then p else q
let f3(b:bool,p:colpoint,q:rgbpoint):point2d = if b then p else q
```

   `f4` est mal typée car il n'y a pas de type qui soit à la fois plus grand que `int` et `bool`.

   (b) Implanter une fonction `sup` qui étant donnés deux types τ_1 et τ_2 calcule le plus petit type σ tel que $\tau_1 \leq \sigma$ et $\tau_2 \leq \sigma$.

   Correction : *On rappelle que le plus petit type est celui qui a le plus de champs. On cherche donc à garder tous les labels communs qui ont des types compatibles. On peut par exemple parcourir les labels du type record t_2 et pour chaque label l qui apparait aussi dans t_1 le garder avec un type qui est à la fois plus grand que le type de l dans $t-1$ et dans t_2. Il est possible d'avoir comme résultat un type record qui ne contiendra aucun champs.*

```
exception NoSup
let rec sup t1 t2 = match t1,t2 with
  (Trec labt1, Trec labt2) ->
        let rec collect = function
            [] -> []
          | (l,tau2)::labt2 ->
             (try let tau1 = List.assoc l labt1
                  in (l,sup tau1 tau2)::collect labt2
              with Not_found | NoSup -> collect labt2)
        in mkTrec (collect labt2)
  | (_,_) -> if t1 = t2 then t1 else raise NoSup
```

   (c) En déduire l'implémentation de la fonction de typage pour le cas de la conditionnelle.

   Correction :

```
let rec type_expr env e = match e with
  | If(b,e1,e2) ->
    let tb = type_expr env b and t1 = type_expr env e1
    and t2 = type_expr env e2
    in if tb = Tbool then try sup t1 t2 with NoSup -> raise TypeError
       else raise TypeError
```

2.2 Génération de code (5 points)

La compilation du langage RECL est faite en allouant les records dans le tas. La valeur d'un record est l'adresse du bloc. On considère le programme suivant où l'on utilise les abbréviations `point2d` pour `{ x:int; y:int }` et `seg2d` pour `{ left:point2d; right:point2d }`:

```
let milieu (s:seg2d) : point2d =
  { x=(s.left.x+s.right.x)/2; y=(s.left.y+s.right.y)/2 }
let cut (s:seg2d, n:int) : point2d =
  if n=0 then s.right else
    ( s.right:=milieu(s) ; cut(s,n-1) )
```

1. Donner le code assembleur mips correspondant à ce programme. On prendra soin de préciser le tableau d'activation de chaque fonction (où sont stockés les arguments et le résultat, ainsi que les registres à sauvegarder).

   Correction : *On peut décider de passer les arguments dans les registres $\$a0$ et $\$a1$ et le résultat dans le registre $\$v0$.*

La fonction `milieu` *stockera donc la valeur de* s *dans* $\$a0$*. Elle ne fait pas d'appels à d'autres fonctions et donc il n'est pas nécessaire de sauvegarder* $\$ra$*. Par contre elle fait appel à* `syscall` *qui utilise* $\$a0$*. Le registre* $\$t0$ *sera utilisé pour sauvegarder la valeur de rega0, les registres* $\$t0$*,* $\$t1$*,* $\$t2$*, ...seront utilisés pour les calculs intermédiaires.*
La fonction `cut` *stockera la valeur de* s *dans* $\$a0$ *et la valeur de* n *dans rega1. Cette fonction fait un appel à la fonction* `milieu` *ce qui nécessite de sauvegarder la valeur de* $\$ra$ *dans la pile. A priori, il faut aussi sauvegarder la valeur de l'argument de la fonction* $\$a0$ *(dans ce cas particulier, la valeur de* $\$a0$ *vaut encore* s *à la sortie de la fonction* `milieu` *et la valeur de rega1 n'est pas modifiée, et donc la sauvegarde n'est pas nécessaire. On remarque que l'appel à la fonction* `cut` *est terminal et on fait par conséquent une compilation optimisée.*
On a besoin pour chaque label de son décalage associé dans le bloc, c'est une information facile à calculer au moment du typage et qui pourra être conservée dans l'arbre de syntaxe abstraite.
Le label x *correspond à un décalage de* 0 *et le label* y *correspond à un décalage de* 4*, le label* `left` *correspond à un décalage de* 0 *et le label* `right` *correspond à un décalage de* 4*.*

```
milieu:
# sauvegarde de s
   move $t0,$a0
# allocation d'un bloc pour 2 valeurs
   li $v0,9
   li $a0,8
   syscall
# restauration de s dans $a0
   move $a0,$t0
# x <- (s.left.x + s.right.x)/2
   lw $t0,0($a0)    # t0 <- s.left
   lw $t0,0($t0)    # t0 <- s.left.x
   lw $t1,4($a0)    # t1 <- s.right
   lw $t1,0($t1)    # t1 <- s.right.x
   add $t0,$t0,$t1  # t0 <- t0+t1
   li $t1,2
   div $t0,$t0,$t1  # t0 <- t0/2
   sw $t0,0($v0)    # x <- t0
# y <- (s.left.y + s.right.y)/2
   lw $t0,0($a0)    # t0 <- s.left
   lw $t0,4($t0)    # t0 <- s.left.y
   lw $t1,4($a0)    # t1 <- s.right
   lw $t1,4($t1)    # t1 <- s.right.y
   add $t0,$t0,$t1  # t0 <- t0+t1
   li $t1,2
   div $t0,$t0,$t1  # t0 <- t0/2
   sw $t0,4($v0)    # y<-t0
# return
   jr $ra
```

```
cut:
# sauvegarde de $ra en sommet de pile
   "pushr" $ra
# label debut fonction recursive terminale
begin:
# test n=0
   beqz $a1,fin
# appel de milieu(s), valeur dans $v0
   jal milieu
   sw $v0,4($a0)
# appel terminal de cut(s,n-1)
   addi $a1,$a1,-1
   j begin
fin:
# result = s.right
   lw $v0,4($a0)
# restauration de $ra/return
   "popr" $ra
   jr $ra
```

2. On se place dans un cadre sans sous-typage. Donner la fonction de codage des expressions qui stocke le résultat de l'évaluation dans le registre $\$v0$ pour les quatre cas suivants. On prendra soin de préciser les informations qui doivent être préalablement calculées sur les types records et stockées dans la table des symboles.

```
let rec code_expr e = match e with
  | Acc ... -> ... | Upd ... -> ... | Rec ... -> ...| If ... -> ...
```

Correction :

```
let rec code_expr e = match e with
| Acc(e,l) -> code_expr e ++ lw $v0,dec(l)($v0)
| Upd(e1,l,e2) -> code_expr e1 ++ pushr $v0 ++ code_expr e2
                          ++ popr $t0 ++ sw $v0,dec(l)($t0)
| Rec(le) -> li $v0,9 ++ li $a0,4*List.length l ++ syscall ++ pushr $v0
      ++ List.iter (fun (l,e) -> code_expr e ++ lw $t0,0($sp) ++ sw $v0,dec(l)($t0)) le
      ++ popr $v0
| If(b,e1,e2) -> let labelse = newlabel() and labend=newlabel() in
                    code_expr b ++ beqz $v0,labelse
                  ++ code_expr e1 ++ j labend
                  ++ labelse: code_expr e2 ++ labend:
```

3. On suppose maintenant que le langage autorise le sous-typage. Quelle conséquence cela a-t-il sur la représentation des records en mémoire?

   Correction : *S'il y a du sous-typage alors il faut que le code travaille de la même manière pour un record dans un type avec des labels $l_1, \ldots, l_k$ ou dans un sous-type (qui a plus de labels). Il faut en particulier que l'accès à un champs de nom l puisse se faire avec le même code. Comme on un même record peut-être un sous-type de deux records différents, on ne peut pas choisir un décalage unique pour le label l sans laisser des "trous" inutilisés dans la structure. On est dans la même situation que de l'héritage multiple dans les langages objets. On décide d'un décalage unique et on utilise un descripteur de type de record unique qui indique pour chaque label don décalage dans le record.*

3 Allocation de registres (4 points)

Soit le programme suivant, on suppose que les variables `r1` et `r3` sont les seules utilisées dans la suite du programme :

```
  c <- r3
  a <- r1
  b <- r2
  d <- 0
  e <- a
loop:
  d <- d+b
  e <- e-1
  if e > 0 then goto loop
  r1 <- d
  r3 <- c
```

1. Indiquer en chaque point de programme l'ensemble des variables vivantes **après** l'exécution de l'instruction correspondante.

   Correction :

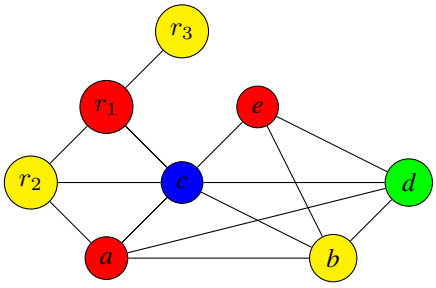
c <- r3	*{c, r1, r2}*
a <- r1	*{a, c, r2}*
b <- r2	*{a, b, c}*
d <- 0	*{a, b, c, d}*
e <- a	*{b, c, d, e}*
loop: d <- d+b	*{b, c, d, e}*
e <- e-1	*{b, c, d, e}*
***if** e > 0 goto loop*	*{b, c, d, e}*
r1 <- d	*{r1, c}*
r3 <- c	*{r1, r3}*

2. Construire le graphe d'interférence associé.

   Correction :

3. Proposer un coloriage de ce graphe avec quatre couleurs.

   Correction :



4. Est-il possible de n'utiliser que trois couleurs ? justifier votre réponse.

   Correction : *Il y a quatre variables simultanément vivantes ce qui nécessite au moins 4 regstres sauf si on décide de stocker l'une des variables en mémoire.*

Fin de l'énoncé

4 Rappels

4.1 Instructions mips

– initialisation

`li`	$r0, C	$r0 ←C
`lui`	$r0, C	$r0 ←$2^{16}$ C
`move`	$r0, $r1	$r0 ←$r1

– arithmétique entre registres:

`add`	$r0, $r1, $r2	$r0 ←$r1+$r2
`addi`	$r0, $r1, C	$r0 ←$r1+C
`sub`	$r0, $r1, $r2	$r0 ←$r1-$r2
`mul`	$r0, $r1, $r2	$r0 ←$r1 × $r2 (pas d'overflow)
`div`	$r0, $r1,$r2	$r0 ←$r1/$r2 (division entière)
`rem`	$r0, $r1,$r2	$r0 ←$r1 % $r2 (reste modulo)
`neg`	$r0, $r1	$r0 ←-$r1

– Test égalité et inégalité

`slt`	$r0, $r1, $r2	$r0 ←1 si $r1<$r2 et $r0 ←0 sinon
`slti`	$r0, $r1, C	$r0 ←1 si $r1< C et $r0 ←0 sinon
`sle`	$r0, $r1, $r2	$r0 ←1 si $r1≤$r2 et $r0 ←0 sinon
`seq`	$r0, $r1, $r2	$r0 ←1 si $r1=$r2 et $r0 ←0 sinon
`sne`	$r0, $r1, $r2	$r0 ←1 si $r1≠$r2 et $r0 ←0 sinon

– Stocker une adresse :

`la`	$r0, adr	$r0 ←adr

– Lire en mémoire. Une adresse est donnée soit par une étiquette soit sous la forme d'un décalage n par rapport à une adresse dans un registre $\$r$ noté $n(\$r)$

`lw`	$r0, adr	$r0 ←mem[adr], lit un mot

– Stocker en mémoire

`sw`	$r0, adr	mem[adr] ←$r0, stocke un mot

– Instructions de saut

`beq`	$r0, $r1, label	branchement conditionnel si $r0=$r1
`beqz`	$r0, label	branchement conditionnel si $r0=0
`bgt`	$r0, $r1, label	branchement conditionnel si $r0>$r1
`bgtz`	$r0, label	branchement conditionnel si $r0>0
`b`	label	branchement inconditionnel

– Saut inconditionnel dont la destination est stockée sur 26 bits

`j`	label	branchement inconditionnel
`jal`	label	branchement inconditionnel avec sauvegarde dans $ra de l'instruction suivante
`jr`	$r0	branchement inconditionnel à l'adresse dans $r0

– Appels système : commande `syscall`

$\$v0$=9 $\$a0$=$n$	alloue un bloc de taille n dans le tas et renvoie l'adresse dans $\$v0$

4.2 Interface d'un module Ocaml pour manipuler les instructions mips

```
(* create a fresh label using the given string as a prefix *)
val new_label : string -> string

type op = Plus | Mult | Div | Minus | Mod | Leq | Geq | Lt | Gt | Eq
type register = | A0 | A1 | V0 | S0 | RA | SP | FP | GP | T0 | T1
type address = | Alab of string | Areg of int * register
type operand = | Oimm of int | Oreg of register

type instruction =
  | Move of register * register
  | Li of register * int   | La of register * string
```

```
  | Lw of register * address | Sw of register * address
  | Bin of op * register * register * operand
  | B of string
  | Beqz of register * string | Bnez of register * string
  | Blt of register * register * string
  | Ble of register * register * string
  | Jal of string | Jr of register
  | Syscall | Label of string

type data = | Asciiz of string * string | Word of string * int

(* abstract type for sequences of instructions *)
type code
(* empty list of instructions *)
val nop : code
(* code given by a list of instructions *)
val mips : instruction list -> code
(* concatenation of two sequences of instructions *)
val (++) : code -> code -> code

type program = { text : code; data : data list; }
(* push the contents of register on the stack *)
val pushr : register -> code
(* pop the value at the top of the stack and put it in the register *)
val popr : register -> code
```

4.3 Fonctions Ocaml élémentaires sur les listes

Rappel de quelques fonctions sur les listes présentes dans la bibliothèque `List` de Ocaml.

```
val length : 'a list -> int
val rev : 'a list -> 'a list
val iter : ('a -> unit) -> 'a list -> unit
val map : ('a -> 'b) -> 'a list -> 'b list
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
val for_all : ('a -> bool) -> 'a list -> bool
val exists : ('a -> bool) -> 'a list -> bool
val mem : 'a -> 'a list -> bool
val find : ('a -> bool) -> 'a list -> 'a
val filter : ('a -> bool) -> 'a list -> 'a list
```