Examen de majeure 2
Compilation

(Durée 2h)

23 mars 2001

On attachera une importance particulière à la clarté, à la précision, mais aussi à la ***concision*** *de la rédaction. Les trois exercices sont indépendants. La plupart des questions sont également indépendentes dans les exercices 1 et 2.*

1 Interruptions de boucles

Nous ajoutons à Pseudo-Pascal une construction **break** qui peut être utilisée à l'intérieur d'une boucle **while** comme dans le langage `C` : l'exécution de l'instruction **break** à l'intérieur d'une boucle **while** interrompt l'exécution de cette boucle et continue avec l'instruction qui suit immédiatement la boucle (c'est une erreur si l'opération **break** apparaît en dehors d'une boucle **while**).

1.1 Exemple d'utilisation de la construction **break**.

a) Donner une fonction ou une procédure Pseudo-Pascal qui utilise l'instruction **break**.

Réponse: *Nous donnons un procédure de recherche d'une valeur* `x` *dans un tableau d'entiers positifs* `t` *de taille* `n` *: La valeur trouvée est retournée, sauf en cas d'échec, où la valeur* -1 *est retournée.*

```
function recherche
  (var x : integer, var n : integer, var t : array of integer) : integer;
var i : integer;
begin
   i := 0;
   while i < n do if t[i] = x then break else i := i+1;
   if  i < n then recherche := t[i] else recherche := −1
end;
```

b) Donner un programme équivalent (qui effectue le même calcul) sans utiliser l'instruction **break**.

Réponse:

```
function recherche
  (var x : integer, var n : integer, var t : array of integer) : integer;
var i : integer;
var b : boolean;
begin
   i := 0;
```

```
    b := i < n;
    while b do
      if t[i] = x then b := false else begin i := i+1; b := i < n end;
    if i < n then recherche := t[i] else recherche := -1
end;
```

1.2 On ajoute à l'arbre de syntaxe abstraire de Pseudo-Pascal (type `Pp.instruction` dans le cours) un nœud `Break`. On donne le squelette d'un interprète du langage Pseudo-Pascal définit par deux fonctions récursives :

```
val instruction : environnement -> Pp.instruction -> unit
val expression : environnement -> Pp.expression -> valeur
```

dont on rappelle la structure ci-dessous :

```
let rec instruction env = function
  | While (e, i1) ->
      while bool (expression env e)
      do instruction env i done
  | ...

and expression env = function
    ...
```

où `bool : valeur -> bool` convertit son argument en un booléen si c'est possible et lance un erreur d'exécution sinon.

Décrire une modification très simple du cas `While` et l'ajout du cas `Break` dans la fonction `instruction` afin de traiter la construction **break**.

Réponse:

```
exception Break_while
let rec instruction env = function
  | While (e, i) ->
      begin try
        while bool (expression env e)
        do instruction env i done
      with Break_while -> ()
      end
  | Break -> raise Break_while
      ...
and ...
```

1.3 Décrire la compilation en code intermédiaire des instructions `While` et `Break` (et d'autres modifications éventuelles à apporter à la fonction de compilation). On utilisera de préférence une description mathématique à l'aide des fonctions $[\![\ ]\!]^e_\rho$ et $[\![\ ]\!]^s_\rho$ utilisés dans le cours et que l'on pourra raffiner.

Réponse: *Pour traduire la boucle* `While` *on ajoute une étiquette optionnelle en paramètre à la fonction d'évaluation. Cette étiquette est propagée au travers des séquences. La traduction de l'instruction* `Break` *utilise cette étiquette. Les autres instructions ignorent cet argument et*

se traduisent comme auparavant.

$$
\begin{aligned}
[\![\texttt{While}\,(e,s)]\!]^s{}_\rho &= \quad \texttt{Stm}\,[\,[\![\texttt{While}\,(e,s)]\!]^e{}_{\rho,\ell};\texttt{Label}\;\ell\,] \quad \textit{où } \ell \textit{ est une étiquette fraîche}\\
[\![\texttt{Break}]\!]^s{}_{\rho,\ell} &= \quad \texttt{Jump}\;\ell\\
[\![\texttt{Seqence}\,[s_1;\ldots s_n]]\!]^s{}_{\rho,\ell} &= \quad [\![s_1]\!]^s{}_{\rho,\ell};\ldots;[\![s]\!]^s{}_{\rho,\ell}\\
[\![s]\!]^s{}_{\rho,\ell} &= \quad [\![s]\!]^s{}_\rho
\end{aligned}
$$

2 Exercice sur la canonisation

Un des buts principaux de la canonisation du code intermédiaire est d'extraire les expressions `Call` emboîtées, pour n'avoir au plus qu'une seule expression `Call` dans une instruction et, le cas échéant, figurant en tête. La canonisation d'une expression retourne une séquence d'instructions extraite et une expression résiduelle. La canonisation d'une instruction retourne une séquence d'instructions.

2.1 Dans la canonisation du code intermédiaire, il est parfois utile de savoir si une expression commute avec une instruction.

a) Donner un exemple de canonisation du code intermédiaire qui utilise la commutation.

Réponse: *L'expression* `t1 + Call (f, [])` *peut être canonisée en le code réduit à l'instruction* `Move (t2, Call (f, []))` *et l'expression résiduelle* `t1 + t2` *parce que* `t1` *commute avec* `Move (t2, Call (f, []))`.

b) Donner une expression et une instruction qui ne commutent pas.

Réponse: *L'expression* `Mem t1` *ne commute pas avec* `Move (t2, Call (f, []))` *: l'appel à f peut, a priori, écrire en mémoire et changer la valeur à l'adresse* `t1`.

c) Que doit-on faire dans ce cas là ?

Réponse: *Il faut calculer l'expression* `Mem t1` *en premier et placer le résultat dans un temporaire auxiliaire. Par exemple, une forme canonique de l'expression :* `t1 + Call (f, [])` *est composée du code extrait [* `Move (t3, Mem t1); Move (t2, Call (f, []))` *] et de l'expression résiduelle* `t3 + t2`.

d) Expliquer pourquoi le code produit peut être moins bon lorsqu'il n'y a pas commutation.

Réponse: *De façon générale, les temporaire auxiliaires peuvent avoir des durée de vie assez longue, ce qui augmente la pression en registre, conduisant à plus de placement en pile.*

Dans l'exemple précédent, le registre auxiliaire `t3` *survit à l'appel de la fonction f qui, par défaut, ne préserve que les registres sauvés par l'appelant. Ainsi, t*3 *sera au mieux placé dans un tel registre qui devra à son tour être sauvé en pile.*

2.2 Dans le cours nous avons choisi un prédicat de commutation très simple, en disant qu'une expression constante commute avec toute autre instruction. Nous considérons ici une définition plus fine : une expression commute avec une instruction si, à la fois,

- les temporaires lus par l'expression ne sont pas écrits par l'instruction, et
- soit l'expression ne lit pas en mémoire, soit l'instruction n'écrit pas en mémoire.

On rappelle le code intermédiaire :

```
type exp =                          and stm =
  Const of int                      | Label of label
| Name of label                     | Move_temp of temp * exp
| Temp of temp                      | Move_mem of exp * exp
| Mem of exp                        | Exp of exp
| Bin of binop * exp * exp          | Seq of stm list
| Call of frame * exp list          | Jump of exp
                                    | Cjump of relop * exp * exp
                                      * label * label
```

a) Écrire en Ocaml une fonction `lus_par_exp` de type `exp -> temp list` qui prend une expression canonique (sans `Call`) du code intermédiaire et retourne la liste des temporaires lus par celle-ci (on admet les répétitions dans le résultat).

Réponse:

```
let rec lus_par_exp = function
  | Const _ | Name _ -> []
  | Bin (_, e1, e2) -> lus_par_exp e1 @ lus_par_exp e2
  | Mem e -> lus_par_exp e
  | Temp t -> [ t ]
  | Call (_, _) -> assert false (* expression canonique *);;
```

b) Écrire en Ocaml une fonction `écrits_par_instr` de type `stm -> temp list` qui prend une instruction canonique (sans saut, sans séquence et sans `Call` emboîtés) du code intermédiaire et qui retourne la liste des temporaires écrits par celle-ci.

Réponse:

```
let rec écrits_par_stm = function
  | Move_temp (t, _) -> [ t ]
  | Move_mem (_, _) | Exp _ | Label _ -> []
  | Seq _ | Cjump (_,_,_,_,_) | Jump _ ->
      assert false (* instruction canonique *);;
```

c) Comment peut-on prendre en compte simplement l'écriture et la lecture en mémoire dans les fonctions `lus_par_exp` et `écrits_par_stm`?

Réponse: *Il suffit d'ajouter un temporaire spécial (utilisé nulle part ailleurs)* `m` *et de considérer la lecture en mémoire comme la lecture de* m *et l'écriture en mémoire comme l'écriture de* m*. On écrira alors les lignes*

```
  | Mem e -> m :: lus_par_exp e
```

et

```
  | Move_mem (t, e) -> [ m ]
```

à la place des lignes correspondantes dans `lus_par_exp` *et* `écrits_par_stm`*.*

2.3 Traiter la mémoire comme un seul bloc est très grossier. En particulier, une lecture en mémoire commute avec une écriture en mémoire lorsque ces deux adresses mémoire sont distinctes.

a) Pourquoi, dans le langage Pascal, peut-on facilement considérer la mémoire comme une partition de zones disjointes ? (Donner quelques exemples d'opérations en mémoire qui peuvent être considérées comme opérant dans des zones différentes)

Réponse: *Le langage Pascal est typé de façon monomorphe. C'est-à-dire deux valeurs de type différents sont forcément à des emplacements-mémoire différents. D'autre part, deux indices différents d'un tableau désignent toujours des emplacements-mémoire différents, même si les tableaux sont du même type voire identiques.*

Ces informations (de types et d'indice) peuvent être transmises au code intermédiaire : deux opérations en mémoire à des emplacements-mémoire disjoints ne peuvent pas interférer.

b) Pourquoi est-ce plus difficile en C ?

Réponse: *Dans le langage C on ne peut pas s'appuyer sur les types parce qu'il est possible de faire des coercions de types. On ne peut pas non plus s'appuyer sur les indices des tableaux, car on peut faire des calculs sur les adresses.*

c) Est-ce encore possible en Ocaml ?

Réponse: *Le polymorphisme peut poser un problème. Il y a donc des situations où les types sont polymorphes et on ne saura pas exactement s'il y a risque d'interférence. Par contre, lorsque deux valeurs ont des types suffisemment connus et incompatibles, on saura, comme en Pascal, que ces valeurs sont à des emplacements-mémoire disjoints.*

Comme en Pascal, on peut également s'appuyer sur les indices.

3 Allocation de registres dans les processeurs du type CISC

Le processeur MIPS, utilisé en cours, est du type RISC. En particulier, les registres sont nombreux et (presque) tous équivalents. Cependant, de vieux processeurs tels que le 68000 distinguent les registres d'adresse (`A0`, ..., `A7`) des registres de donnée (`D0`, ..., `D7`). Pour simplifier, nous considérons un processeur expérimental *CIRC* dont le jeu d'instructions est identique à celui du code SPIM mais dont les registres sont séparés en registres d'adresses (`A0`, ..., `A7`) et registres de données (`D0`, ..., `D7`) et dont certaines occurrences des registres dans les instructions sont contraintes. En particulier, on considère les contraintes suivantes,

- Les opérations arithmétiques (**add**, **mul**, etc.) n'utilisent que des registres de donnée.
- Dans les écritures et lectures en mémoire delà forme **lw** $D, n(A)$ et **sw** $D, n(A)$ le registre D est un registre de donnée, le registre A est un registre d'adresse (et n un entier).
- Les autres instructions sont non contraintes, en particulier **move** R, S peut indépendemment utiliser des registres d'adresses ou de données pour R et S.

On appelle *code machine abstrait* le code de type `Ass.code` dans le cours, abstrait par rapport au choix final des registres et *code machine final* le code émit à la fin de la compilation et pouvant être lu par l'assembleur. Dans l'une ou l'autre expression, on remplace *machine* par SPIM ou CIRC pour préciser la version du cours pour l'architecture SPIM ou la version de cet exercice pour une architecture CIRC.

3.1 Expliquer en quoi l'architecture CIRC pose, a priori, un problème pour l'allocation de registres.

Réponse: *À la génération des instructions, on voudrait par exemple émettre une instruction abstraite (*i.e. *dans laquelle les temporaires ne sont pas encore choisis)* `add` t_0, t_1, t_2 *où*

t_0, t_1 et t_2 sont des temporaires, et laisser le soin à l'allocateur de registres de choisir les registres pour t_0, t_1 et t_2 au mieux, en fonction du graphe d'interférence.

Sans précaution, l'allocateur pourra alors violer la contrainte décrite ci-dessus en choisissant indifféremment des registres d'adresse ou de donnée.

3.2 À la recherche d'une solution.

a) Expliquer succinctement comment on peut exprimer ces contraintes dans le graphe d'interférence.

Réponse: *Pour forcer un temporaire t à être un registre de donnée (resp. d'adresse) il suffit de lui interdire d'être un registre d'adresse (resp. de donnée), donc de le déclarer comme interférant avec tous les registres d'adresse (resp. de donnée).*

b) Proposer une modification légère du code SPIM abstrait en un code CIRC abstrait permettant de mettre simplement en œuvre la contrainte ci-dessus ? (Préciser si besoin, informellement et succinctement, quelle autre partie de la chaîne de compilation il faudra modifier)

Réponse: *On peut ajouter simplement un champ aux instructions de type* `Oper`*, de type* `(temp * temp list) list` *représentant des ensembles de contraintes entre un registre particulier et un ensemble de registres.* i.e. *avoir*

```
type Oper =
   ...
 | Oper of string * temp list * temp list * label list option
       * (  temp * temp list) list
```

La seule autre modification à effectuer est de bien prendre en compte ces contraintes lors de la création du graphe d'interférence pour ajouter le sous graphe $\{t\} \times q$ pour chaque contrainte $(t, q) \in c$ de chaque nœud `Oper`(i, s, d, l, c).

c) On considère le code intermédiaire `Bin (Plus, Temp t1, Temp t2)`. Décrire le code CIRC abstrait produit pour ce code On ne se préoccupera pas ici du contexte dans lequel l'instruction est placée. (On pourra supposer que les variables `rA` et `rD` sont liées à l'ensemble des registres d'adresses et de données respectivement.)

Réponse: *Le code est alors :*

```
Oper ("add ^d0, ^s0, ^s1", [ t1; t2 ], [  t0 ], None, [ t0, rA; t1, rA; t2, rA ])
```

3.3 À la poursuite de la solution.

a) Donner la traduction du code intermédiaire suivant en code SPIM abstrait (on suppose que le nom associé à l'étiquette `debut` est *"debut"*) :

```
[ Label debut;
  Move_mem (Temp t1, Temp t2);
  Move_temp (t1, Bin (Plus (Temp t1, Temp t3)));
  Jump (debut); ]
```

Réponse:

```
[ Label ("debut", debut);
  Oper ("sw   ^d0, ^s0", [ t1 ], [  t2 ], None);
  Oper ("add  ^d0, ^s0, ^s1", [ t1; t3 ], [  t1 ], None);
  Oper ("j debut", [], [],   Some [ debut ]); ]
```

b) Donner un code SPIM final possible.

Réponse: *Les trois temporaires peuvent, par exemple, être placés dans trois registres de la classe t différents.*

```
debut:
        sw    t2, 0(t1)
        add   t1, t1, t3
        j     debut
```

c) Quel est le problème lorsqu'on essaye de générer du code pour l'architecture CIRC ? Donner un code CIRC final équivalent au code SPIM final de la question b.

Réponse: *Il n'est pas possible d'utiliser le même registre pour le temporaire t_0 dans les instructions* `sw` *et* `add`*.*

Réponse:

```
debut:
        sw    D2, 0(A1)
        move  D1, A1
        add   D1, D1, D3
        move  A1, D1
        j     debut
```

d) Quel code CIRC abstrait aurait permis d'obtenir le code CIRC final de la question c ?

Réponse: *Il suffit de permettre d'utiliser deux registres différents pour le même temporaire. Pour cela on introduit une instruction* `move` *intermédiaire :*

```
[ Label (debut);
  Oper ("sw   ^d0, ^s0", [ t1 ], [  t2 ], None, [ t1, rD; t2, rA ]);
  Move ("move ^d0, ^s0", [ t1 ], [ t1' ]);
  Oper ("add  ^d0, ^s0, ^s1", [ t1'; t3 ], [  t1' ], None, [ t1', rA; t3, rA ]);
  Move ("move ^d0, ^s0", [ t1' ], [ t1 ]);
  Oper ("j    debut", Some debut); ]
```

3.4 La démarche de la question 3.3 revient, sur un cas particulier, à une effectuer une transformation du code CIRC de façon à rendre les contraintes solvables.

a) En s'inspirant d'une transformation vue en cours, proposer une mécanisation de cette transformation.

Réponse: *En s'inspirant de la mise en pile, on peut remplacer chaque occurrence d'un temporaire contraint par un temporaire éphémère. Plus précisément, pour chaque instruction instr de type* `Oper` *:*

- *on remplace chaque occurrence contrainte i d'un temporaire t dans instr par un temporaire éphémère t_i ;*
- *on mémorise les contraintes portant sur t_i dans instr ;*
- *si t_i est utilisé en lecture, on ajoute une instruction* **move** *t_i, t avant instr ;*
- *si t_i est utilisé en écriture, on ajoute une instruction* **move** *t, t_i après instr ;*

L'instruction **move** *d, s est une abréviation pour* `Move ("move ^d0, ^s0", d, s)`*.*

En fait, on pourrait ne considérer que les temporaires qui apparaissent au moins une fois dans le code avec des contraintes incompatibles, ce qui en pratique sera assez peu fréquent.

c) Expliquer pourquoi le code résultant sera plus efficace qu'il n'y paraît.

Réponse: *Cette transformation introduit beaucoup de* `move` *intermédiaires. Mais l'allocateur de registres devrait pourvoir éliminer au moins la moitié de ces* `move`*. En effet, considérons un temporaire contraint t, et ses registres éphémères associés $t_1, \ldots t_k$. Les contraintes sur les t_i n'étant que de deux types (registre d'adresse ou registre de donnée), un bon allocateur devrait identifier t avec un registre respectant la contrainte la plus fréquente de façon à éliminer le plus d'instructions* `move`*!*

Les instructions `move` *restantes ne seront pas éliminés.*

d) Avez-vous un exemple typique qui sera moins efficace que ce que l'on pourrait obtenir manuellement ?

Réponse: *Après élimination de la moitié des* `move` *par l'allocateur de registres comme décrit à la question c), il peut rester une configuration de la forme :*

move $t,\ t_1$
...
move $t_2,\ t$

où t_2 et t_1 sont des registres d'adresses et t un registre de donnée, et t n'apparaît pas dans "...". Or, il n'est pas nécessaire ici de sauvegarder t_1 dans t entre t_1 et t_2 : un transfert direct de t_1 dans t_2 suffirait et permettrait d'identifier t_1 avec t_2. On peut donc faire mieux ! (Voir, par exemple, l'examen de l'an 2000 et sa suite.)