Compilation
Master Informatique M1 2010-2011

Examen Final
15 décembre 2010

Les exercices sont indépendants. L'énoncé est composé de 6 pages. L'examen dure 3 heures. Les notes de cours et de TD manuscrites et les supports de cours distribués cette année (incluant les corrigés de TD) sont les seuls documents autorisés.

Consigne Lorsqu'il est demandé de décrire une méthode, vous n'êtes pas obligés de donner du code dans tous ses détails (vous pouvez utiliser des fonctions auxiliaires dont l'implantation est directe en vous contentant de spécifier leur comportement). Vous devez par contre décrire les données manipulées et les algorithmes utilisés de manière précise.

1 Génération de code (10 points)

Soit les fonctions C suivantes que l'on souhaite compiler vers du code mips:

```
int fib (int n)
{ if (n<=0) return 0;
  else if (n==1) return 1;
  else return (fib(n-2) + fib(n-1));
}

int fibi (int x, int y, int m)
    { if (m<=0) return x; else fibi (y, x+y, m-1); }

int main () { printf("%d\n",fib(4) == fibi(0,1,4)); }
```

1. Donner la forme des tableaux d'activation des fonctions `fib` et `fibi`.
2. Proposer un code mips pour calculer ces fonctions (les principales instructions mips sont rappelées en fin de document). Dans le cas de la fonction `fibi`, on remarquera que la récursion est terminale et on proposera un code itératif ne faisant pas d'appel récursif.
3. Dans un langage intermédiaire, le calcul de `fibi`$(0,1,4)$ s'écrit de la manière suivante:

```
m ← 4
x ← 0
y ← 1
loop: t ← 0<m
if t==0 goto end
z ← x+y
x ← y
y ← z
m ← m-1
goto loop
end: v ← x
```

   (a) Calculer les variables vivantes après chacune des instructions du programme précédent (on suppose qu'à la fin du programme, seule la variable v est utilisée).

(b) Construire le graphe d'interférence.

(c) Proposer un coloriage de ce graphe en utilisant 4 couleurs.

(d) On suppose que l'on ne dispose que de trois registres. Proposer une manière de compiler ce code en instructions mips en n'utilisant que trois registres pour stocker les variables x, y, z, t, m, v.

2 Langage avec instructions `break` et `continue` (10 points)

On considère un mini-langage impératif BC comportant des conditionnelles et des boucles. Ce langage comporte de plus deux instructions `break` et `continue` qui permettent de modifier l'exécution du corps d'une boucle. Lorsque dans le corps d'une boucle, on rencontre l'instruction `break` alors l'exécution se poursuit à la fin de la boucle, lorsque l'on rencontre l'instruction `continue` alors l'exécution reprend au test de la boucle. Lorsque plusieurs boucles sont imbriquées, les instructions `break` et `continue` modifient l'exécution de la première boucle englobante.

On cherche à compiler le langage BC en faisant tout d'abord une phase d'analyse statique pour vérifier la bonne formation des expressions puis la génération de code mips.

Les expressions du langage BC sont soit des constantes entières, soit des variables (globales ou paramètres de fonction) soit une opération binaire (arithmétique ou test d'égalité) appliquée à deux expressions, soit l'appel d'une fonction. On convient que les booléens `true` et `false` sont représentés respectivement par les entiers 1 et 0.

Une instruction de BC peut être soit une affectation $x = e$, soit une conditionnelle **if** (e) s_1 **else** s_2 ou bien **if** (e) s, soit un bloc formé d'une liste d'instructions $\{s_1; \ldots; s_n\}$ soit une boucle **while** (e) s, soit une instruction d'échappement **break** ou **continue**, soit une instruction de retour de fonction **return** e.

Les types CAML suivants permettent de représenter les arbres de syntaxe abstraite de ce langage (les variables globales sont représentées par leur étiquette permettant d'accéder à la zone de données, les variables locales sont identifiées par un entier représentant le décalage par rapport au registre $fp dans le tableau d'activation):

```
type op = Plus | Mult | Sub | Div | Mod | Eq
type expr = Const of int | Glob of string | Loc of int
          | Op of op * expr * expr | Call of string * expr list
type stat =   Aff of string * expr
            | If of expr * stat * stat
            | Block of stat list
            | While of expr * stat
            | Break | Continue
            | Return of expr
```

1. Un sous-ensemble de la grammaire du langage contient les règles suivantes:

```
expr:
| CST                                { }
| IDENT                              { }

instr:
| IF LP expr RP instr                { }
| IF LP expr RP instr ELSE instr     { }
| WHILE LP expr RP instr             { }
| RETURN expr                        { }
```

```
| BREAK                                  { }
| CONTINUE                               { }
;
```

Ocamlyacc indique un conflit shift/reduce dans l'état suivant

```
20: shift/reduce conflict (shift 22, reduce 3) on ELSE
state 20
instr : IF LP expr RP instr .  (3)
instr : IF LP expr RP instr . ELSE instr  (4)

ELSE  shift 22
$end  reduce 3
```

(a) Donner un exemple d'entrée sur laquelle le conflit se produit.

(b) Indiquer comment le conflit est résolu (on s'appuiera sur les conventions des langages comme C ou Java) et à quelle stratégie (lecture ou réduction) cela correspond.

(c) Donner le choix de précédence pour supprimer le conflit dans la grammaire.

2. Écrire une fonction `check_loop` qui vérifie que dans chaque instruction, les commandes **break** et **continue** n'apparaissent que dans le corps d'une boucle.

3. Une instruction **return** correspond à la fin d'un flot de contrôle du programme. Écrire une fonction `check_return` qui vérifie que dans une instruction du langage BC, tout chemin d'exécution se termine bien par une instruction **return**. On vérifiera de plus qu'il n'y a pas d'instruction qui suit immédiatement une instruction dont le flôt de controle se termine par des **return** et qui donc n'est jamais atteinte par le flot de contrôle.
Exemples.

```
{ if (x == 0) return 0;  x=x+1;  return x; }
```

Le programme précédent est correct: le flôt après le **if** va soit dans la branche **return** 0 qui se termine par un return, soit dans la branche `x = x+1; return x;` qui se termine bien aussi par un **return**.

```
{ while (x <= 0) { x=x+1; return 0; } }
```

Le programme précédent est incorrect: le flot de contrôle de l'instruction **while** peut ne pas entrer dans la boucle auquel cas il n'y aura pas d'instruction **return**.

```
{ return 0; x=x+1 }
```

Le programme précédent est incorrect: le flot de contrôle n'atteint jamais l'instruction `x=x+1`.

On pourra découper en deux fonctions, la première qui vérifie que chaque chemin d'exécution se termine par un **return** et la seconde qui vérifie qu'il n'y a pas d'instruction non-atteignable à la suite d'une instruction dont l'exécution se termine toujours par un **return**.

4. On suppose donnée une fonction `compile_expr` qui prend en argument un registre r et une expression e et produit le code mips permettant de calculer la valeur de e dans le registre r.
Écrire une fonction de compilation des instructions de BC vers du code mips dans le cas où il n'y a pas d'instruction **break** ou **continue** dans les boucles. La valeur de l'expression renvoyée par l'instruction **return** sera stockée dans le registre $\$v0$. On pourra utiliser l'interface Ocaml pour manipuler des instructions mips donnée en fin de document ou bien utiliser de manière informelle la syntaxe des instructions mips dans le code Ocaml.

5. Soit le programme de BC :

```
sum = 0;
x = 0;
while (true) {
    x = x+1;
    if (x % 2 == 1) continue;
    if (x == 10) break;
    sum = sum + x;
    }
return(sum);
```

L'opération $n\%p$ calcule le reste de la division entière de n par p (opération modulo) et s'implante par l'opération mips `rem`.

(a) Quel est le résultat de ce programme ?

(b) Dessiner le graphe de flot de contrôle de ce programme.

(c) Donner des instructions mips correspondant au code compilé de cette fonction en supposant que sum, et x sont des variables globales.

6. Écrire une fonction de compilation des instructions dans le cas général où les corps de boucle peuvent mentionner les instructions **break** et **continue**.

Rappels

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=1	$a0=$n$	imprime l'entier n
$v0=4	$a0=$s$	imprime la chaîne de caractères s
$v0=11	$a0=$c$	imprime le caractère c
$v0=9	$a0=$n$	alloue un bloc de taille n dans le tas et renvoie l'adresse dans $v0

Interface d'un module Ocaml pour manipuler les instructions mips

```
type op = Plus | Mult | Div | Minus | Mod
        | Leq | Geq | Lt | Gt | Eq

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

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 label
  | Beqz of register * label
  | Bnez of register * label
  | Blt of register * register * label
  | Ble of register * register * label
  | Jal of label
```

```
  | 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
```

Fonctions Ocaml utiles 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
```