Compilation
Master Informatique M1 2010-2011

Examen Final
1 juin 2011

Les exercices sont indépendants. L'énoncé est composé de 5 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.

1 Question de cours (2 points)

Indiquer les principales étapes et langages intermédiaires utilisés lors de la génération de code pour passer de l'arbre de syntaxe abstraite du programme jusqu'à un code MIPS raisonnablement optimisé. *La réponse ne devra pas dépasser une page.*

2 Analyse syntaxique (3 points)

On considère un langage avec trois symboles terminaux `CTE`, `ARRAY` et `TIMES`, `LP`, `RP` et les règles suivantes :

```
typ :
| CTE                          { }
| typ ARRAY                    { }
| typ TIMES typ                { }
| LP typ RP                      { }
;
```

Ocamlyacc indique deux conflits shift/reduce dans l'état suivant

```
10: shift/reduce conflict (shift 7, reduce 3) on ARRAY
10: shift/reduce conflict (shift 8, reduce 3) on TIMES
state 10
typ : typ . ARRAY  (2)
typ : typ . TIMES typ  (3)
typ : typ TIMES typ .  (3)

ARRAY  shift 7
TIMES  shift 8
$end  reduce 3
RP  reduce 3
```

1. Donner pour chacun des conflits un exemple d'entrée sur laquelle le conflit se produit.
2. On souhaite que l'opérateur `TIMES` associe à gauche et que le symbole `ARRAY` ait une précédence plus forte que `TIMES`. Indiquer dans l'état précédent, quelle action (shift ou reduce) doit être faite lorsque le symbole à lire est `ARRAY` et lorsque c'est `TIMES` pour respecter ces conventions.
3. Compléter la grammaire avec le choix de précédence sur les terminaux pour supprimer les conflits dans la grammaire.

3 Génération de code (8 points)

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

```
int comb (int n, int k)
{ if (n<0) return 0;
  else if (k<0 || k>n) return 0;
  else if (k==0) return 1;
  else return (comb(n-1,k-1) + comb(n-1,k));
}

int main () { int i;
  for (i=0;i<=5;i++){printf("%d\n",comb(5,i));}
}
```

1. Donner la forme du tableau d'activation de la fonction `comb`. On indiquera où sont stockés les arguments et le résultat de la fonction et ce qui doit être stocké dans la pile à chaque appel.
2. Proposer un code mips pour calculer les fonctions `comb` et `main` en utisant la mémoire comme proposé à la question précédente.
   Remarque: les principales instructions mips sont rappelées en fin de document.
3. Dans un langage intermédiaire, un calcul de $\texttt{comb}(5,3)$ s'écrit de la manière suivante:

   ```
   k ← 3
   a[0] ← 1
   i ← 1
   loop1: if k < i goto end1
   a[i] ← 0
   i ← i+1
   goto loop1
   end1: m ← 1
   loop2: if 5 < m goto end2
   l ← k
   loop3: if l < 1 goto end3
   x ← a[l]
   j ← l-1
   y ← a[j]
   a[l] ← x+y
   l ← l-1
   goto loop3
   end3: m ← m+1
   goto loop2
   end2: v ← a[k]
   ```

   (a) Construire le graphe de flot de contrôle de ce programme.

   (b) 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).

   (c) Construire le graphe d'interférence.

   (d) Proposer un coloriage de ce graphe en utilisant le minimum possible de couleurs.

4 Compilation d'un langage avec tableaux dynamiques (8 points)

Ce problème étudie la compilation d'un mini-langage fonctionnel permettant de manipuler des tableaux de taille arbitraire contenant des données qui peuvent être elles-mêmes des tableaux.

Syntaxe du langage

Un programme du langage est composé d'une suite de déclarations.

Déclarations Une *déclaration de variable* s'écrit : **let** *id* = *exp* avec *id* le nom de la variable et *exp* sa valeur.

Une *déclaration de fonction* s'écrit : **let** *id* $(id_1, \ldots, id_n)$ = *exp* avec *id* le nom de la fonction, id_i le nom du i-ème paramètre, *exp* une expression représentant le corps de la fonction et pouvant faire référence à *id*.

Expressions Une *expression* peut être :

- une variable, déclarée dans un **let** ou bien comme paramètre d'une fonction.
- une valeur entière ou booléenne (`true` ou `false`).
- une expression exp_1 `op` exp_2 formée d'un opérateur binaire prédéfini `op` appliqué à deux expressions exp_1 et exp_2 avec `op` $\in \{+, -, =, <, \leq\}$
- un appel de fonction *id*(exp_1,...,exp_n)
- une expression conditionnelle : **if** *exp* **then** exp_1 **else** exp_2.
- une expression avec déclaration locale *decl* **in** *exp* où *decl* est une déclaration de variable ou de fonction éventuellement récursive avec la même syntaxe que les déclarations globales.

Fonctions prédéfinies On suppose que les fonctions suivantes permettant la manipulation des tableaux sont prédéfinies :

`create` qui étant donné un entier n et une expression x, crée un tableau de taille n initialisé par la valeur de x;

`length` qui étant donné un tableau renvoie sa longueur;

`get` qui étant donné un tableau t et un entier i compris entre 1 et la longueur de t, renvoie la valeur $t[i]$ du tableau t à l'indice i;

`set` qui étant donné un tableau t, un entier i compris entre 1 et la longueur de t, et une expression x, affecte la valeur de x à l'emplacement d'indice i du tableau et renvoie le tableau t ainsi modifié.

4.1 Typage

Les fonctions peuvent prendre des tableaux en arguments et renvoyer des tableaux. Les données manipulées sont des entiers, des booléens ou bien des tableaux formés d'objets quelconques. Les fonctions peuvent être polymorphes, c'est-à-dire s'appliquer à des tableaux contenant des objets arbitraires.

Les *types* du langage sont soit le type de base des entiers `int`, soit le type des booléens `bool`, soit un type tableau contenant des objets de type τ, noté τ `array`.

Pour traiter le polymorphisme, on introduit les *schémas de type* qui peuvent contenir des variables de type qui représentent des types arbitraires. Un schéma de type est donc soit `int`, soit `bool`, soit une variable de type (notée $\alpha, \beta, \ldots$), soit un schéma de type tableau noté σ `array` avec σ un schéma de type.

Un schéma de type représente un ensemble de types obtenus en remplaçant les variables de type par des types quelconques. Ainsi le schéma de type α représente tous les types, le schéma de type

α `array` représente tous les types de la forme τ `array` avec τ un type (ie `int array`, `bool array`, (`int array`) `array`, ((`int array`) `array`) `array` ...)

À toute fonction est associée un *profil* de la forme $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ où σ_i sont des schémas de type.

On peut associer aux opérations primitives les profils suivants :

- `create` : `int` $\times \alpha \to \alpha$ `array`
- `length` : α `array` $\to$ `int`
- `get` : α `array` $\times$ `int` $\to \alpha$
- `set` : α `array` $\times$ `int` $\times \alpha \to \alpha$ `array`

1. Dire ce que font les fonctions suivantes et donner leur profil.

```
let f(x,y)=x
let get_matrix(m,i,j) = get(get(m,i),j)
let set_matrix(m,i,j,v) = set(get(m,i),j,v)
```

2. Écrire une fonction `add_array` qui prend en argument deux tableaux d'entiers x et y de même taille n et renvoie un nouveau tableau z de taille n tel que pour tout i compris entre 1 et n on ait $z[i] = x[i] + y[i]$. Donner le profil de la fonction.

4.2 Génération de code

Le langage est compilé vers le code mips dont les instructions sont rappelées à la fin de ce document.

Les tableaux sont alloués dans la zone du tas. La valeur d'un tableau sera l'adresse du bloc dans lequel le tableau est stocké. Un tableau n'est jamais recopié lors d'un appel de fonction, seule l'adresse est passée en argument. La seule fonction qui alloue de la place pour un tableau dans le tas est la fonction `create`.

1. Peut-on connaître statiquement (à la compilation) la taille du tableau correspondant à la valeur d'une expression quelconque du langage, en particulier pour l'expression correspondant au corps d'une fonction ?
2. Proposer une manière de représenter les tableaux qui permette que tous les objets manipulés dans la pile ou les registres (entiers ou tableaux) aient la même taille et d'implanter la fonction `length`.
3. On veut coder les opérations primitives (`length`, `get`, `set` et `create`). Expliquer l'organisation du tableau d'activation créé par l'appel de ces fonctions et donner le code mips correspondant au corps de chacune de ces procédures (on prendra soin de commenter le code).
4. Soit le programme :

```
let x = create(2,0)
let y = x
let x' = set(x,1,2)
let v = get(y,1)
```

   On suppose les variables `x`, `y`, `v` stockées dans des registres. Indiquer l'état de ces registres et du tas après l'exécution de chaque déclaration.

5. Donner le code machine correspondant à la fonction `add_array` définie précédemment.

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 $\$v0$