Compilation
Master Informatique M1 2006-2007

Examen Final

18 décembre 2006

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

1 Analyse ascendante (8 points)

1. Donner un exemple de grammaire ambiguë.
2. Une grammaire ambiguë peut-elle servir pour l'analyse syntaxique d'un langage de programmation ? Dans quelles conditions ?
3. On se donne un langage de types permettant de décrire le type des entiers ou celui de fonctions à valeur entière, prenant en argument des entiers ou d'autres fonctions de même nature.
   Un type est donc soit la constante `int` soit de la forme $\tau_1 * \ldots * \tau_n \rightarrow \texttt{int}$ avec τ_i des types. Pour reconnaître ce langage de types, on se donne la grammaire suivante avec comme ensemble de terminaux $\{\sharp, \rightarrow, *, \texttt{int}\}$ et comme ensemble de non-terminaux $\{S, A, T\}$ avec S le symbole de départ:

S	:= T $\sharp$
T	:= `int`
T	:= A $\rightarrow$`int`
A	:= T
A	:= T * A

   (a) Calculer l'ensemble des suivants de T et de A.

   (b) Construire la table d'analyse SLR(1) de cette grammaire en indiquant en cas de conflit les différentes actions possibles. Cette grammaire est-elle SLR(1) ?

   (c) Expliquer la nature du conflit obtenu en donnant un exemple d'entrée où ce conflit se produit. La grammaire donnée est-elle ambiguë ?

   (d) Que suggérez-vous pour remédier à ce problème ?

2 Manipulation de tableaux par tranches (12 points)

On se donne un mini-langage de programmation permettant de manipuler des valeurs scalaires et des tableaux. Il dispose des opérations arithmétiques classiques (e.g., addition, multiplication) et celles-ci sont implicitement applicables à des tableaux. Par exemple, si `t1` et `t2` désignent deux tableaux de taille n contenant des entiers, l'expression `t1 + t2` retourne un nouveau tableau obtenu en additionnant les éléments de même indice de `t1` et `t2` [1]. De même, $\texttt{t1} \leq \texttt{t2}$ est vrai si et seulement si pour tout i, $0 \leq i < n$, $\texttt{t1}[i] \leq \texttt{t2}[i]$.

L'objectif de cet exercice est d'étudier le typage de ce langage et sa génération de code pour la machine abstraite vue en cours.

[1] On retrouve ce type de construction dans les langages de description de circuits logiques (un tableau de booléens représentant naturellement un groupe de fils).

On considère un petit langage d'expressions (*exp*). Une expression peut être, soit une définition locale de la forme `let` $x = e_1$ `in` e_2, soit une expression conditionnelle `if` *cmp* `then` e_1 `else` e_2 où *cmp* désigne une expression booléenne, l'application d'un opérateur arithmétique binaire *op* à deux arguments $op(e_1, e_2)$, une opération de modification d'un tableau avec recopie `copy` e_1 `with` i `<-` e_2 où i est un entier positif, une opération de manipulation de tranches de tableaux e`[`i_1`-`i_2`]` où i_1 et i_2 sont des entiers, un tableaux de n éléments `[|`$e_1; ...; e_n$`|]`, l'accès à l'élément d'indice i du tableau e, noté e`[`i`]`, et enfin un entier (i) ou une variable (x).

$$
\begin{array}{lcl}
exp & ::= & \texttt{let } x = exp \texttt{ in } exp \mid \texttt{if } cmp \texttt{ then } exp \texttt{ else } exp \mid op(exp, exp) \\
 & & \mid \texttt{copy } e \texttt{ with } i \texttt{ <- } e \mid e\texttt{[}i\texttt{-}i\texttt{]} \mid \texttt{[|}exp; ...; exp\texttt{|]} \mid e\texttt{[}i\texttt{]} \\
 & & \mid i \mid x \\
op & ::= & + \mid - \mid * \mid / \\
cmp & ::= & exp \leq exp \mid cmp \texttt{ or } cmp \mid cmp \texttt{ \& } cmp
\end{array}
$$

Les opérations arithmétiques ($+$, $-$, $*$, $/$) peuvent s'appliquer à la fois à des valeurs entières et à des tableaux.

1. Si t_1 et t_2 sont deux tableaux de taille n, alors $op(t_1, t_2)$ retourne un nouveau tableau t de taille n d'indices allant de 0 à $n-1$ et tel que $t[i] = op(t_1[i], t_2[i])$.
2. Si l'expression e_1 s'évalue en un tableau de taille n (dont les indices vont de 0 à $n-1$), l'opération `copy` e_1 `with` i `<-` e_2 retourne un nouveau tableau égal à e_1 mais dont l'élément d'indice i est égal à la valeur de e_2.
3. Si e désigne un tableau de taille n, e`[`i_1`-`i_2`]` retourne un nouveau tableau t de taille $i_2 - i_1 + 1$ tel que $t[i - i_1] = e[i]$, pour $i_1 \leq i \leq i_2$. e`[`i_1`-`i_2`]` désigne une "tranche" du tableau x. Cette opération impose que le tableau e contienne bien des éléments d'indices i_1 à i_2 avec $i_1 \leq i_2$.
4. L'expression `[|`$e_0; e_2; ...; e_{n-1}$`|]` désigne un tableau neuf de taille n initialisé avec les valeurs des expressions e_i.
5. L'expression e`[`i`]` rend l'élément d'indice i du tableau e. Cette opération est définie à condition que e contienne un élément d'indice i.

Ce langage est défini par le type Caml suivant.

```
type exp =
    | Elet of string * exp * exp
    | Eifthenelse of cmp * exp * exp
    | Eop of op * exp * exp
    | Ecopy of exp * int * exp
    | Eslide of exp * int * int
    | Eacces of expr * int
    | Earray of exp list
    | Eint of int
    | Evar of string

and cmp =
    | Eless of exp * exp
    | Eor of cmp * cmp
    | Eand of cmp * cmp

and op = Eplus | Eminus | Etimes | Ediv
```

Partie I: vérifications statiques

On voit ici que la manipulation des tableaux impose des contraintes sur les tailles. Ainsi, il n'est pas possible d'additionner des tableaux de taille différentes. La vérification des tailles est confiée à la phase de typage.

1. (1pt) Indiquer, parmi les deux programmes suivants, ceux qui sont bien typés et le cas échéant le résultat du calcul. Vous justifierez votre réponse.

```
let x = [|1;2;3;4|] in
let z = (x + x) - [|1;1;1;1|] in
if z[2] <= 0 then z[2-3] + [|1;2|]
else z[0-1]
```

```
let x = [|1;2;3;4|] in
let z = (x + x) - [|1;1;1;1|] in
if z[2] <= 0 then z[2-3] + [|1;2|]
else copy x 1 <- 4
```

2. (1pt) On introduit les types `int` et $\texttt{int}[n]$. Le type `int` est le type de base. $\texttt{int}[n]$ est le type d'un tableau d'entiers de taille n et d'indices 0 à $n-1$.

   On note $H \vdash e : t$ le jugement de typage qui signifie que l'expression e est de type t dans l'environnement H. On note $H(x)$ le type associé à x dans l'environnement H et $H + x : t$ l'environnement H augmenté de la liaison $x : t$. Les règles de typage sont données ci-dessous.

$$\frac{H(x) = t}{H \vdash x : t} \qquad \frac{}{H \vdash i : \texttt{int}} \qquad \frac{H \vdash e_1 : t_1 \qquad H + x : t_1 \vdash e_2 : t_2}{H \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t_2}$$

$$\frac{H \vdash c \quad H \vdash e_1 : t \quad H \vdash e_2 : t}{H \vdash \texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t} \qquad \frac{H \vdash e_1 : t \quad H \vdash e_2 : t}{H \vdash op(e_1, e_2) : t} \qquad \frac{H \vdash e : \texttt{int}[n] \quad i_1 \leq i_2 < n}{H \vdash e\texttt{[}i_1\texttt{-}i_2\texttt{]} : \texttt{int}[i_2 - i_1 + 1]}$$

   Le jugement de typage $H \vdash c$ signifie que la comparaison c est bien typée. La règle de vérification du type des comparaisons est donnée ci-dessous.

$$\frac{H \vdash e_1 : t \quad H \vdash e_2 : t}{H \vdash e_1 \leq e_2} \qquad \frac{H \vdash c_1 \quad H \vdash c_2}{H \vdash c_1\ \texttt{or}\ c_2} \qquad \frac{H \vdash c_1 \quad H \vdash c_2}{H \vdash c_1\ \texttt{\&}\ c_2}$$

   Le systèmes de règles de typage est incomplet. Donnez la règle de typage pour les constructions $\texttt{copy}\ e_1\ \texttt{with}\ i\ \texttt{<-}\ e_2$, $\texttt{[|}e_1;...;e_n\texttt{|]}$ et $e\texttt{[}i\texttt{]}$.

3. (0.5pt) Les types du langage sont donnés par la définition suivante. On suppose disposer également d'un module de manipulation des environnements de typage. `find x h` lève l'exception `Not_found` si `x` n'est pas défini dans `h`.

```
type typ = Int | Array of int

module Env :
  sig
    exception Not_found
    type key = string
    type 'a t
    val empty : 'a t
    val is_empty : 'a t -> bool
```

```
    val add : key -> 'a -> 'a t -> 'a t
    val find : key -> 'a t -> 'a
  end
```

Écrire, en suivant les règles d'inférence, une fonction de typage `type_expr : typ Env.t -> exp -> typ` qui retourne le type d'une expression et une fonction `type_comp : typ Env.t -> cmp -> unit` de typage des comparaisons. Ces deux fonctions affichent un message en cas d'erreur de typage. Proposer un traitement d'erreur permettant de distinguer les cas suivants:
- Identificateur inconnu
- Accès à un indice indéfini d'un tableau
- Valeurs de tailles incompatibles

4. (0,5pt) On souhaite maintenant que le résultat du typage d'une expression retourne son type et une expression annotée par la taille de son résultat. On adopte la convention qu'un entier est de taille 1. On introduit pour cela le langage des expressions typées suivant:

```
type texp = { desc: exp_desc; size: int }
and exp_desc =
      | Tlet of string * texp * texp
      | Tifthenelse of tcmp * texp * texp
      | Top of op * texp * texp
      | Tcopy of texp * int * texp
      | Tslide of texp * int * int
      | Tarray of texp list
      | Tint of int
      | Tvar of string

and tcmp =
      | Tless of texp * texp
      | Tor of cmp * cmp
      | Tand of cmp * cmp
```

Reprendre la fonction de typage précédente afin d'écrire une fonction de typage `type_expr : typ Env.t -> exp -> typ * texp`.

Partie II: génération de code

Le langage est compilé vers le code d'une machine à pile dont les instructions sont celles vues en cours et en TD. Elles sont brièvement rappelées à la fin de ce document.

On considère d'abord le cas de la compilation des constructions de base. Les tableaux seront traités plus loin. Les variables locales introduites par la construction `let/in` sont stockées dans la pile avec un décalage relatif à `fp`. Pour déterminer la localisation d'une variable, on supposera l'existence de deux fonctions globales:

```
val index : string -> int
val creer_index : string -> int -> unit
```

de sorte que `index x` indique le déplacement relatif de la variable `x` par rapport à `fp`. `x` est stockée à l'adresse [`fp`+`index` (x)]. `creer_index x dec` indique que la variable `x` est stockée à l'adresse [`fp`+`dec`].

On fait l'hypothèse que toutes les variables définies sont différentes deux à deux. On suppose également l'existence d'une fonction `gen_label : unit -> string` retournant un nom de label neuf.

5. (1.5pt) Écrire une fonction de compilation des expressions typées. On se limite ici aux variables, constantes entières, définitions locales, opérations arithmétiques entre entiers et conditionnelles dans lesquelles les deux branches retournent un entier. Cette fonction de compilation devra mettre à jour la table des symboles associant le déplacement relatif à chacune des variables locales en utilisant la fonction `creer_index`.

On suppose que les tableaux sont stockés dans la pile. Les éléments d'un tableau t de taille n et d'indices 0 à $n-1$ sont stockés dans des cases consécutives de la pile. Le résultat de l'évaluation d'une expression retournant un tableau de taille n est stocké en sommet de pile aux adresses $sp-n; ...; sp-1$, l'élément d'indice 0 étant stocké à l'adresse $sp-n$. On supposera l'existence de deux nouvelles instructions de la machine abstraite.

`LOADSP` n	dépile un décalage d et stocke en sommet de pile le contenu de $sp+d+n$.
`STORESP` n	dépile une valeur v, un décalage d et stocke la valeur v à l'adresse $sp+d+n$.

6. (0,5pt) Soit t un tableau stocké en sommet de pile aux adresses $sp-n; ...; sp-1$. Écrire le code d'une fonction *get* n i retournant la valeur de l'indice i en sommet de pile. Le tableau argument est donc effacé.
7. (0,5pt) Écrire le code de compilation d'une expression `[|`$e_0; ...; e_{n-1}$`|]`. Le résultat est stocké en sommet de pile.
8. (1,5pt) Si t_1 et t_2 sont deux tableaux de taille n dont le premier est stocké aux adresses $sp-2n; ...; sp-n-1$ et le second aux adresses $sp-n; ...; sp-1$, écrire le code sous forme d'instructions de la machine à pile pour l'addition de deux tableaux. Le résultat sera stocké à la place de t_1. Le sommet de pile est alors décrémenté de n (il pointe sur la première case libre au dessus du résultat de t_1+t_2). Nous noterons cette opération d'addition *addt* n. Vous donnerez schématiquement le contenu de la pile avant et après cette opération.

   Note: Toute solution produisant du code de taille proportionnelle à n sera jugée fausse.
9. (1pt) Sur le même modèle, définir les opérations *mult* n, *divt* n et *subt* n, respectivement pour la multiplication, la division et la soustraction.
10. (0,5pt) Que doit-on modifier dans le cas de la compilation de conditionnelles lorsque le résultat des deux branches est un tableau ?
11. (0,5pt) Écrire le code d'une fonction *copyt* n d permettant de copier en sommet de pile le contenu d'un tableau (de taille n) stocké à l'adresse $fp+d$. À la fin de cette opération, le sommet de pile est incrémenté de n.
12. (0,5pt) Écrire le code d'une fonction *storet* n d permettant de stocker le contenu d'un tableau de taille n stocké aux adresses $sp-n; ...; sp-1$ vers les adresses $fp+d; ...; fp+d+n-1$.
13. (0,5pt) Soit t un tableau de taille n stocké en sommet de pile (de $sp-n$ à $sp-1$). Écrire le code d'une opération *slicet* n_1 n_2 retournant en sommet de pile une copie de taille n_2-n_1+1. Le tableau précédent est donc effacé. Montrer schématiquement l'organisation de la pile avant et après cette opération.
14. (1,5pt) A l'aide des définitions précédentes, compléter la fonction de compilation des expressions.
15. (0,5pt) Proposer une définition des instruction `LOADSP` et `STORESP` à partir des instructions de la machine à pile existente.

3 Instructions de la machine à pile

`PUSHI` n	empile la constante entière n.
`PUSHA` l	empile l'adreese correspondant au code de label l.
`ADD`	dépile y puis x et empile $x+y$
`SUB`	dépile y puis x et empile $x-y$
`MUL`	dépile y puis x et empile $x*y$
`EQ`	dépile y puis x et empile 1 si $x=y$ et 0 sinon
`NOT` n	dépile n qui doit être un entier et empile le résultat de $n=0$
`PUSHL` n	empile la valeur située n cases au dessus de *fp*.
`STOREL` n	dépile x et le stocke à l'emplacement situé n cases au dessus de *fp*.
`PUSHGP`	empile l'adresse *gp*.
`PUSHFP`	empile l'adresse *fp*.
`PADD`	dépile un entier n et une adresse a et empile l'adresse $a+n$.
`LOAD` n	dépile une adresse a et empile la valeur située à l'adresse $a+n$
`STORE` n	dépile une valeur v et une adresse a et stocke la valeur v à l'adresse $a+n$.
`PUSHN` n	empile n valeurs nulles sur la pile.
`SWAP`	échange les contenus des deux éléments consécutifs en sommet de pile.
`POP` n	dépile n valeurs de la pile.
`JZ` l	dépile x, si $x=0$ le pointeur de code saute à l'instruction de label l.
`JUMP` l	saute à l'instruction de label l.