Compilation
Master Informatique M1 2005-2006

Examen Final

9 janvier 2006

L'énoncé est composé de 7 pages. L'examen dure 3 heures. Les notes de cours et de TD manuscrites ainsi que les énoncés et supports de cours distribués cette année sont les seuls documents autorisés.

1 Analyse Syntaxique (7 pts)

1. Soit la grammaire G_1 définie sur l'ensemble des terminaux $\{\texttt{,}, \texttt{id}, \texttt{(}, \texttt{)}\}$.

E' := E♯	S := (E)
E := `id` E	S := `id`
E := (E , E)	
E := S	

2. Cette grammaire est-elle LR(0), SLR(1)? Vous justifierez votre réponse.
3. Cette grammaire est-elle LL(1) ? Le justifier.
4. Construire une grammaire G_1' LL(1) équivalente à G en factorisant les préfixes communs de G_1.
5. Calculer pour la grammaire G_1' les ensembles PREM(X) et SUIV(X) pour chaque non-terminal.
6. On considère la grammaire G_2 issue de G_1.

E' := E♯	S := (E)
E := `id` E	S := `id`
E := E , E	
E := S	

On supposera que la construction de paire `,` est associative à gauche et que l'application de fonction `id` E est plus prioritaire que la construction de paire. Écrire une grammaire Yacc en utilisant les précédences et permettant de reconnaître le langage reconnu par G_2. On ne cherchera pas ici à construire un arbre de syntaxe abstraite et les règles produiront une valeur de type `unit`.

2 Compilation d'un langage avec tableaux dynamiques (13pts)

Ce problème étudie la compilation d'un petit langage fonctionnel permettant de manipuler des tableaux de taille arbitraire pouvant eux-mêmes contenir des tableaux.

2.1 Syntaxe du langage

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

Déclarations : Une déclaration de fonction s'écrit `fun` $id_1\,(id_2 : t_2) : t_1 = e$ où id_1 est le nom de la fonction, id_2 est le nom de son paramètre, e est une expression représentant le corps de la fonction et pouvant faire référence à id_1 et id_2. t_1 est le type du résultat de e et t_2 est le type de id_2.

La signature de type de id_1 sera notée $t_2 \rightarrow t_1$.

Expressions : Une expression peut être:

- une variable id, déclarée comme paramètre d'une fonction ou par un `let/in`;
- une valeur entière notée i;
- l'application d'une fonction id à une expression e et notée $id(e)$;
- une déclaration locale $\texttt{let}\, x = e_1 \,\texttt{in}\, e_2$;
- une expression $\texttt{[|}e_1; \ldots ; e_n\texttt{|]}$ désignant un tableau de n élements qui doivent tous être du même type. Dans ce tableau, le premier élément est d'indice 1 et le dernier élément est d'indice n où n désigne le nombre d'éléments;
- l'accès à un élément du tableau $e_1.(e_2)$ où e_1 est un tableau et e_2 est l'indice accédé dans ce tableau. Le type du résultat est celui des éléments du tableau.
- la modification physique d'un tableau notée $e_1.(e_2)\texttt{<-}\, e_3$. L'expression $e_1.(e_2)\texttt{<-}\, e_3$ retourne le tableau e_1 dans lequel l'élément d'indice e_2 a été remplacé par la valeur de e_3. Le type du résultat est le type de e_1.

La grammaire de ce langage est définie ainsi:

$$
\begin{array}{lcl}
d & ::= & \texttt{fun}\, id\,(id : t) : t = e \\
 & & \\
e & ::= & id \mid i \mid id(e) \mid \texttt{let}\, x = e \,\texttt{in}\, e \\
 & & \quad \mid \texttt{[|}e; \ldots ; e\texttt{|]} \\
 & & \quad \mid e.(e) \mid e_1.(e_2)\texttt{<-}\, e_3 \\
 & & \\
t & ::= & \texttt{int} \mid t\,\texttt{array} \\
 & & \\
schema & ::= & t \rightarrow t
\end{array}
$$

Les programmes sont représentés par des valeurs du type Caml suivant:

```
type decl = Efun of string * string * typ * typ * expr
and  expr = Elet of string * expr * expr
          | Eapp of string * expr
          | Evar of string
          | Eint of int
          | Earray of expr list
          | Eset of expr * expr * expr
          | Eget of expr * expr

and typ = Tint | Tarray of typ
and schema = typ * typ
```

2.2 Typage

Les règles de typage et de portées de ce langage sont celles vues en cours et en TD. On suppose dans la suite que toutes les variables locales ont des noms différents. On supposera également l'existence des fonctions suivantes permettant d'accéder à la table des symboles:

```
val ajout: string -> typ -> unit
val trouve: string -> typ
val fajout: string -> schema -> unit
val ftrouve: string -> schema
```

1. Écrire une fonction `verif_expr: expr -> typ` de vérification des types pour les expressions. Une exception `Erreur_de_type` sera levée en cas d'erreur de type. Cette fonction mettra à jour la table des symbole
2. Écrire une fonction `verif_decl: decl list -> unit` de vérification des types pour les déclarations.

2.3 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 dans les parties suivantes. 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 stocké à l'adresse [`fp`+`index` (x)]. `creer_index x dec` indique que la variable `x` est stockée à l'adresse [`fp`+`dec`].

1. Écrire une fonction de compilation des expressions. On se limite ici aux variables, constantes entières et définitions locales. Cette fonction de compilation devra mettre à jour, en utilisant la fonction `creer_index`, la table des symboles associant le déplacement relatif à chacune des variables locales.
2. Montrer le code et la structure de la pile correspondant à la compilation d'une application de fonction $id(e)$. Étendre la fonction de compilation des expressions en conséquence.
3. Écrire une fonction de compilation des déclarations de fonctions. *Indication:* On rappelle qu'il est nécessaire de réserver la place sur la pile au début du code de la fonction afin de stocker les variables locales. Les accès aux variables locales se font relativement à `fp`.

2.4 Allocation des tableaux dans le tas

On étudie maintenant la compilation des instructions de création, de modification et d'accès aux tableaux. 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 de l'appel d'une fonction, seule l'adresse est passée en argument.

Pour créer un tableau de taille n, on écrira l'expression `[|`$e_1;\ldots;e_n$`|]`. On décide de représenter un tableau par une structure allouée dans le tas de la manière suivante:

Un tableau de longueur n est représenté par un tableau de longueur $n+1$ où le premier élément indique la taille du tableau. On notera `length`(e) la longueur n du tableau (la longueur effective du tableau est $n+1$). Ainsi, l'exécution du code correspondant à une expression `[|`$e_1;\ldots;e_n$`|]` retourne (sur la pile), l'adresse de début du tableau.

1. Quel est le code (avec les instructions de la machine à pile) de la fonction `length`(e) retournant la taille du tableau e ?
2. Énoncer le principe de la compilation de la construction `[|`$e_1;\ldots;e_n$`|]`. Étendre la fonction de compilation des expressions pour traiter cette construction.
3. Étendre la fonction de compilation des expressions pour traiter les constructions $e_1.(e_2)$ et $e_1.(e_2)$ `<-` e_3. La construction $e_1.(e_2)$ doit retourner en résultat (stocké en sommet de pile). Le contenu de la case $v_1.(v_2)$ si v_1 désigne la valeur de e_1 et v_2 désigne la valeur de e_2. La construction $e_1.(e_2)$ `<-` e_3 permet de modifier physiquement le contenu du tableau e_1. Elle retourne (en sommet de pile) l'adresse du tableau e_1.

2.5 Allocation des tableaux en pile

On souhaite maintenant améliorer l'efficacité de la compilation des structures de données en remarquant que, dans certains cas, un tableau peut être stocké dans la pile. L'allocation et la désallocation du tableau sont confiées au compilateur permettant d'éviter d'utiliser le ramasse-miettes.

Lorsque l'on rencontre la construction particulière `let` $x =$ `[|`$e_1; \ldots; e_n$`|]` `in` e, on peut dans certains cas, allouer le tableau sur la pile. C'est le cas, par exemple, avec le programme suivant:

```
let t = [|6;7;1|] in
t.(1) + t.(2)
```

En supposant que le début du tableau `t` est stocké à l'adresse [`fp`+`42`], ce programme peut être compilé ainsi:

```
PUSHI 3  - taille du tableau
STOREL 42
PUSHI 6
STOREL 43
PUSHI 7
STOREL 44
PUSHI 1
STOREL 45

PUSHL 43
PUSHL 44
ADD
```

Lors de la compilation d'une construction `let` $x =$ `[|`$e_1; \ldots; e_n$`|]` `in` e, on évalue les arguments e_i que l'on stocke dans la pile les uns au-dessus des autres à partir de l'emplacement réservé pour x. Ainsi, la taille allouée pour x est égale à $n + 1$ (alors qu'elle était égale à 1 dans la partie précédente).

Afin de déterminer si une variable locale x peut être allouée en pile, on supposera l'existence de deux fonctions globales:

```
val stocker_en_pile: string -> unit
val en_pile: string -> bool
```

`stocker_en_pile x` indique que la variable `x` désigne un tableau qui est alloué en pile. `en_pile x` retourne la valeur vraie dans ce cas.

1. Énoncez le principe de la compilation d'une expression de la forme `let` $x =$ `[|`$e_1; \ldots; e_n$`|]` `in` e lorsque le tableau peut être alloué en pile. Étendre la fonction de compilation des expressions pour la construction `Elet(x, e1, e2)` lorsque `x` peut être alloué en pile.
2. Que doit-on modifier dans la fonction de compilation des accès aux éléments d'un tableau? Peut-on améliorer le code de la compilation de $x.(i)$ et de $x.(i)$ `<-` e dans le cas où x est stocké en pile?
3. Cette compilation en pile n'est pas toujours possible. Indiquer, parmi les programmes suivants, les variables définissant des tableaux pouvant être alloués en pile.

P_1:
```
fun f(x:int array):int array array =
  let p = [|x;x|] in
  p.(1)
```

P_2:
```
fun f(x:int array):int =
  let p = [|x;x|] in
  p.(1).(2)
```

P_3:
```
let t1 = [|1;2|] in
let t2 = [|t1.(0);t1.(1)|] in
t2.(1)
```

P_4:
```
let t1 = [|1;2|] in
let t2 = [|t1;[|3;4|]|] in
t2.(0)
```

2.6 Analyse de portée

On souhaite maintenant effectuer une analyse à la compilation permettant de déterminer les déclarations de tableaux pouvant être allouées en pile. Le résultat de l'analyse doit mettre à jour la table des symboles avec cette information (en utilisant la fonction globale `stocker_en_pile`). Cette analyse sera fondée sur une technique de typage.

Le principe consiste à associer un identifiant unique au type d'un tableau introduit dans une construction $\texttt{let}\, x = \texttt{[|}\, e_1; \ldots; e_n \,\texttt{|]}\ \texttt{in}\, e$. Si l'on peut démontrer que cet identifiant unique n'apparaît pas dans le type du résultat de e, alors cela signifie que le tableau reste local à la construction `let/in`. Autrement dit, il n'apparaît pas dans le résultat du calcul. Il peut donc être alloué localement.

On modifie le langage des types de base de la manière suivante:

$$t' \ ::= \ t' \,\texttt{array}[I] \mid \texttt{int}$$

$$I \ ::= \ \{id_1, \ldots, id_n\}$$

I est un ensemble d'identifiants. Ceci correspond au type Caml suivant:

```
type typ' = Mint | Marray of l * typ'
and  l = set
```

On supposera l'existence des définitions suivantes:

```
type set
val vide : set
val ajout : string -> set -> set
val appartient : string -> set -> bool
val union : set -> set -> set
```

Étant donnés deux types t_1 et t_2, on définit l'opération $t_1 \sqcup t_2$ qui retourne un nouveau type obtenu en "rassemblant" les identifiants. Cette opération est définie seulement lorsque les types s'accordent (t_1 et t_2 désignent le même ensemble de valeurs). Cette opération est définie structurellement par:

$$\begin{array}{lcl} \texttt{int} \sqcup \texttt{int} & = & \texttt{int} \\ (t_1\, \texttt{array}[I_1]) \sqcup (t_2\, \texttt{array}[I_2]) & = & (t_1 \sqcup t_2)\, \texttt{array}[I_1 \cup I_2] \end{array}$$

Ainsi, $(\texttt{int}\,\texttt{array}[\{x\}] \sqcup (\texttt{int}\,\texttt{array}[\{y\}] = \texttt{int}\,\texttt{array}[\{x, y\}]$. Si $t_1, \ldots, t_n$ sont des types, on notera $\sqcup_{1 \leq i \leq n}(t_i) = t_1 \sqcup t_2 \cdots \sqcup t_n$.

On modifie le typage des expressions pour pouvoir produire des types de t'. On dira que l'identificateur x peut être alloué en pile lorsque la règle suivante s'applique:

$$\text{(LETPILE)}\ \frac{H \vdash e_1 : t'_1 \ \ldots \ H \vdash e_n : t'_n \quad t' = \sqcup_{1 \leq i \leq n}(t'_i) \quad H, x : t' \,\texttt{array}[\{x\}] \vdash e : t'' \quad n \notin noms(t'')}{H \vdash \texttt{let}\, x = \texttt{[|}\, e_1; \ldots; e_n \,\texttt{|]}\ \texttt{in}\, e : t''}$$

Cette règle signifie que si les expressions $e_1, \ldots, e_n$ sont de type t'_i tels que $t' = \sqcup_{1 \leq i \leq n} t'_i$ et que l'on peut typer e dans un environnement dans lequel x est de type $t'\,\texttt{array}[\{x\}]$ annoté avec un identifiant x et obtenir un type t'' dans lequel x n'apparaît pas, alors x peut être alloué en pile. Sinon, il devra être alloué en tas. Cela signifie que le tableaux x n'apparait pas dans le résultat du calcul: il peut donc être alloué localement. $noms(t'')$ est la liste de tous les identifiants apparaissant dans t''.

De même, les règles de typage des fonctions d'accès aux tableaux deviennent:

$$\text{(TAB)}\ \frac{H \vdash e_1 : t'_1 \quad \dots \quad H \vdash e_n : t'_n \quad t' = \sqcup_{1\leq i\leq n}(t'_i)}{H \vdash \texttt{[|}e_1 \ldots e_n\texttt{|]} : t'\,\texttt{array}[\emptyset]}$$

$$\text{(ACCES)}\ \frac{H \vdash e_1 : t'_1\,\texttt{array}[I] \quad H \vdash e_2 : \texttt{int}}{H \vdash e_1.(e_2) : t'_1}$$

$$\text{(MODIF)}\ \frac{H \vdash e_1 : t'_1\,\texttt{array}[I_1] \quad H \vdash e_2 : \texttt{int} \quad H \vdash e_3 : t'_3 \quad t' = t'_1 \sqcup t'_3}{H \vdash e_1.(e_2)\,\texttt{<-}\,e_3 : t'\,\texttt{array}[I_1]}$$

1. Écrire une fonction `apparait: string -> typ' -> bool` telle que `apparait n t' = true` lorsque $n \in \mathit{noms}(t')$.
2. On suppose ici que les expressions ne contiennent pas d'appel de fonction. Écrire une nouvelle fonction de typage des expressions permettant de mettre à jour la table des symboles avec l'information de marquage. À la fin de cette analyse, `en_pile x` est vrai pour toute variable locale déclarant un tableau pouvant être stocké en pile.
3. On souhaite maintenant traiter un cas plus général de programmes contenant des appels de fonctions ayant des arguments ou des résultats de type tableau.

   Proposer une règle de typage dans le cas où les fonctions appliquées rentrent dans l'un des deux cas suivants:

   (a) application d'une fonction prenant un entier en argument;

   (b) application d'une fonction retournant un entier en résultat.
4. Que pensez-vous de la règle suivante ?

   $$\text{(APPLY)}\ \frac{\texttt{ftrouve}(f) = t_1 \to t_2 \quad H \vdash e : t'_1 \quad t_1 \rhd t'_1 \quad t_2 \rhd t'_2}{H \vdash f(e) : t'_2}$$

   avec $\texttt{int} \rhd \texttt{int}$, $t\,\texttt{array} \rhd t'\,\texttt{array}[\emptyset]$ quand $t \rhd t'$.
5. Proposer une piste pour résoudre le cas général.

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
`PUSHG` n	empile la valeur située n cases au dessus de gp.
`STOREG` n	dépile x et le stocke à l'emplacement situé n cases au dessus de gp.
`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 gp.
`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.
`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.
`CALL`	dépile une adresse de code *c* sauve la valeur courante de *fp* et le pointeur d'instructions, affecte à *fp* la valeur de la première case libre de la pile, saute à l'instruction d'adresse *c*.
`RETURN`	restaure les valeurs de *fp*,*sp* et *pc*.
`ALLOC` n	alloue sur le tas un bloc structuré de taille n et empile l'adresse correspondante.