Compilation
Master Informatique M1 2004-2005

Examen Final
17 janvier 2005

L'énoncé est composé de 6 pages. L'examen dure 3 heures. Les notes de cours et de TD **manuscrites** ainsi que les énoncés et supports de cours sont les seuls documents autorisés.

1 Commentaires (3 points)

La technique la plus simple de description de commentaires dans un langage de programmation est de choisir une suite de symboles pour indiquer le début du commentaire, une suite de symboles pour indiquer la fin de commentaire et d'interpréter comme un commentaire tous les caractères situés après la lecture du "début de commentaire" et avant la première apparition de la séquence de "fin de commentaire".
Citer deux autres techniques utilisées et discuter pour chacune de ses trois méthodes les avantages et les inconvénients en terme de confort pour l'utilisateur et de méthode d'analyse.

2 Langage M (18 points)

On s'intéresse au typage d'un mini-langage fonctionnel M. Ce langage permet de manipuler des types de base mais aussi des listes d'objets dans un type de base. Les fonctions peuvent être passées en argument. On s'intéresse à l'analyse sémantique des programmes de ce langage ainsi qu'à la génération de code. Les questions sont largement indépendantes.

Instructions Un programme source est une séquence d'instructions. Les instructions sont de deux types, définition de variable ou de fonction globales:

`var` *id* `=` *expression*
`fun` *id* `(` var_1 `:` $type_1$ `,` var_2 `:` $type_2$ `,` ...`)` `:` *type* `=` *expression*

Par exemple on écrira:

```
fun square (y:int) : int = y*y
var x = 4+5;
fun main () : int = (square x)+3;
```

Types Les types du langage M sont soit des types de base soit des types listes d'objets dans un type de base soit des types fonctionnels dont le résultat est un type de base (pas une liste ou une fonction).

types de base	`bool`, `int`	ordinaires
types listes	*type* `list`	*type* est un type de base
types fonctionnels	$type_1$ `*` ⋯ `*` $type_n$ `->` *type*	*type* est un type de base

Les arguments d'une fonction peuvent être eux-mêmes des fonctions mais le type résultat d'une fonction doit toujours être un type de base.

Exemple

```
fun f(h: int -> int, x:int) : int = x * h(x)
fun decr(n:int) : int = n-1
var res = f(decr,10)
```

Expressions Les expressions possibles sont:

constante booléenne	`true`, `false`	
constante numérique	3, 4...	
identificateur	x, y ...	
définition locale	`let` *id* `=` $expr_1$ `in` $expr_2$	
conditionnelle	`if` $expr_1$ `then` $expr_2$ `else` $expr_3$	
liste d'objets	$[e_1; \ldots; e_n]$	
opération binaire	e_1 `op` e_2	`op` opérateur
application de fonction	$f(e_1, \ldots, e_n)$	f identificateur de fonction
opérateurs (`op`)	`+`,`-`, `*`, `/`, `and`, `or`, `<`, `<=`, `=`, `<>`	
fonctions prédéfinies	`length`,`val`,`it_list`	

Les expressions de base obéissent aux règles de typage usuelles avec une portée statique des variables. Les fonctions peuvent être récursives. Dans une définition locale, la variable déclarée peut être dans un type de base ou un type liste mais pas dans un type fonctionnel. La fonction `length` s'applique à n'importe quel objet dans un type liste et renvoie la longueur de la liste. La fonction `val` s'applique à une liste l et un entier n, si n est compris entre 1 et la longueur de la liste, alors `val`(l,n) renvoie le n-ème élément de la liste l. La fonction `it_list` prend en argument une fonction f de type $(\tau \times \sigma \rightarrow \tau)$, un objet x de type τ et une liste l d'objets de type σ. Si la liste l est $[a_1; \ldots; a_n]$ alors le résultat de `it_list`(f, x, l) est $f(\ldots f(f(x, a_1), a_2) \ldots, a_n)$.

2-A Analyse sémantique (6 points)

1. Proposer des types CAML pour représenter les arbres de syntaxe abstraite des types, des expressions et des instructions de ce langage. On prendra soin de distinguer les types de base et les types composés. On suppose donnés les types suivants et on complètera la définition.

   ```
   type ident = string
   type op = Plus|Minus|Mult|Div|And|Or|Lt|Le|Eq|Neq
   type cte = Bool of bool|Int of int
   type type_base = ...
   type type_M = ...
   type expr = ...
   type instr = ...
   ```

2. On suppose que l'analyse de portée est faite, que chaque identificateur est unique et que la table des symboles comporte pour chaque identificateur (variable ou fonction) le type déclaré. Ce type est accessible par une fonction CAML `type_id` de type `ident` →`type_M`. Donner le code d'une fonction CAML `type_of` qui prend en entrée une expression et renvoie son type ou bien échoue si l'expression n'est pas bien typée.

3. On veut limiter l'utilisation des expressions explicites de liste $[e_1; \ldots; e_n]$ à des déclarations de variables globales (`var` x=$[e_1; \ldots; e_n]$) ou locales (`let` x=$[e_1; \ldots; e_n]$ `in` e). On peut toujours se ramener à ce cas en remarquant que `let` x=$[e_1; \ldots; e_n]$ `in` x

représente la même valeur que $[e_1; \ldots; e_n]$.
Donner le code d'une fonction CAML `transform` qui transforme une expression quelconque en une expression où la seule occurrence de la construction $[e_1; \ldots; e_n]$ est dans une déclaration locale `let` x=$[e_1; \ldots; e_n]$ `in` e. On pourra utiliser une fonction `new_name` qui crée des identificateurs nouveaux. On ne traitera que les cas de l'appel de fonction, de la construction `let in`, des listes explicites $[e_1; \ldots; e_n]$ et des constantes.

4. On se donne l'expression suivante :

   ```
   it_list(f,val([1;2;3],1),[true;false;true])
   ```

   Quel doit être le type de f pour que cette expression soit bien typée ? Donner l'expression équivalente qui ne contient des listes explicites qu'associées à des variables.

5. Ce langage ne comporte pas de fonctions mutuellement récursives, cependant le passage de fonctions en argument permet de ramener les fonctions mutuellement récursives au cas simple de fonctions récursives.
   Une définition mutuellement récursive a la forme :

   `fun f (x1:`τ_1`,...,xn:`τ_n`):`τ `= ea`
   `and g (y1:`σ_1`,...,yp:`σ_p`):`σ `= eb`

   Elle est bien typée si en associant à `f` le type $(\tau_1 \times \ldots \times \tau_n) \rightarrow \tau$ et à `g` le type $(\sigma_1 \times \ldots \times \sigma_p) \rightarrow \sigma$ alors l'expression `ea` (resp. `eb`) a pour type τ (resp. σ).
   On remplace cette définition de deux fonctions mutuellement récursives par la déclaration de trois fonctions qui n'utilisent que de la récursion simple

   `fun f'(g:(`$\sigma_1\times$ `... ×` σ_p`)→`σ`, x1:`τ_1`,...,xn:`τ_n`):`τ `= ea'`
   `fun g(y1:`σ_1`,...,yp:`σ_p`):`σ `= eb'`
   `fun f(x1:`τ_1`,...,xn:`τ_n`):`τ `= f'(g,x1,...,xn)`

   avec `ea'` (resp. `eb'`) la même expression que `ea` (resp. `eb`) dans laquelle on a remplacé les appels récursifs à `f(e1,...,en)` par `f'(g,e1,...,en)`.
   Donner un programme sans récurrence mutuelle équivalent à

   ```
   fun even(n:int) : bool = if n=0 then true else odd(n-1)
   and odd(n:int) : bool = if n=0 then false else even(n-1)
   ```

   Généraliser la construction au cas des trois fonctions mutuellement récursives :

   ```
   fun mod0(n:int) : bool = if n=0 then true else mod2(n-1)
   and mod1(n:int) : bool = if n=0 then false else mod0(n-1)
   and mod2(n:int) : bool = if n=0 then false else mod1(n-1)
   ```

2-B Génération de code pour la machine à pile (6 points)

Le langage précédent présente plusieurs difficultés à la compilation : la manipulation des listes et le passage de fonctions en paramètres. Dans cette partie, on compile les programmes du langage M vers les instructions de la machine à pile vue en cours qui sont rappelées dans la section 3. Lors de la définition des fonctions de génération de code, on précisera les informations utiles qui devront être stockées dans la table des symboles.

1. Pour la manipulation des fonctions, on choisit qu'une expression qui est une fonction aura pour valeur l'adresse du code de la fonction. Ce schéma convient car les seules fonctions manipulées sont les fonctions déclarées par l'utilisateur ou bien passées en argument.

Donner le code de la fonction `gener` de génération de code des expressions dans le cas de l'appel de fonction $f(e_1, \ldots, e_n)$.
On suppose que le passage des arguments et de la valeur de retour se font comme dans le cours, la difficulté à traiter concerne f qui peut-être une fonction définie par l'utilisateur ou un paramètre.

2. Donner le code à engendrer pour la fonction suivante :

```
fun f(h: int -> int, x:int) : int = x * h(x)
fun decr(n:int) : int = n-1
var res = f(decr,10)
```

3. On choisit de se restreindre au sous-langage où les expressions explicites de liste $[e_1; \ldots; e_n]$ ne se trouvent que dans des déclarations de variables globales ou locales. On peut ainsi réserver l'emplacement mémoire nécessaire pour ces variables puis ne manipuler les valeurs de type liste que par leur adresse.
   (a) Expliquer succintement une autre représentation possible des valeurs de type liste sur la pile. Quelle méthode vous semble préférable ?
   (b) Comment choisir la représentation des listes en mémoire afin d'engendrer aisément le code de la fonction `length` ?
   (c) Compléter le code de la fonction `gener` de génération d'instructions de la machine à pile dans le cas des expressions suivantes :
   `let` x $= [e_1; \ldots; e_n]$ `in` e; x (variable); `val`(e_1, e_2); et `length`(e).
   (d) Donner un code pour la compilation de l'expression `it_list`(f, x, l). On pourra commencer par construire un programme calculant la valeur de `it_list`(f, x, l) en utilisant une boucle et une variable supplémentaire.

2-C Génération de code pour une machine à registres (6 points)

Dans cette partie, on utilise un sous-ensemble du langage M qui ne contient pas de listes. On cherche à compiler ce langage vers une machine comportant des registres.

Pour cela on commence par traduire le programme vers un langage I formé d'une suite d'instructions élémentaires : les opérations ne se font que sur des valeurs *immédiates* (notée i, i_1, i_2 dans le tableau suivant) qui sont des constantes entières ou des variables. Les instructions peuvent être précédées d'un label.

valeurs immédiates (i)	id, 2,3...	variables ou constantes entières
affectation simple	$id \leftarrow i_1$ `op` i_2	
saut conditionnel	`if` i `goto` l	l est un label valide, saut si $i \neq 0$
saut	`goto` l	l est un label valide
appel de fonction	$id \leftarrow$ `call`$(f, i_1, \ldots, i_n)$	
retour	`return`	
sauvegarde sur la pile	$M[i_1] \leftarrow i_2$	stocke i_2 à l'adresse i_1
lecture dans la pile	$id \leftarrow M[i]$	donne à id la valeur à l'adresse i

1. Soit le programme :

```
      x1 ←2
      n1 ←2
      x0 ←x1
      n0 ←n1+1
      p ←x1
exp:  t ←n0 = 1
      if t goto end
      n0 ←n0-1
      p ←p*x0
      goto exp
end:  res ←p
```

Indiquer les variables vivantes à chaque point de programme en supposant que seule la variable *res* est utilisée dans la suite du programme. Construire le graphe d'interférence et proposer un coloriage à l'aide de 4 registres. Est-ce le nombre minimal ?

2. Donner le code CAML d'une fonctions `interm` qui transforme une expression du langage M (restreint) vers une suite d'instructions du langage intermédiaire I qui stocke la valeur attendue dans une variable donnée x. On pourra engendrer autant de variables nouvelles que nécessaire. Donner le résultat de votre fonction sur l'expression :

   ```
   if x=4 then 1 else let z = 2*x*x in f(2*y-4+z)-3*z
   ```

3. Afin de compiler le corps de la fonction f, il faut que l'appelant et l'appelé aient un schéma commun pour l'emplacement des paramètres et de la valeur de retour.

   (a) On peut adopter un schéma où les paramètres et la valeur de retour sont alloués dans la pile. On suppose que l'on a des variables particulières *sp*, *fp* et des instructions du langage intermédiaire `call`(f) et `return` qui ont le même comportement que dans la machine à pile vue en cours. Modifier la génération de code intermédiaire pour l'appel de fonction $f(e_1, \ldots, e_n)$ et la déclaration de fonction `fun` $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e$

   (b) Une solution alternative est de réserver des variables spécifiques (toujours les mêmes qui correspondront à des registres particuliers) pour stocker les paramètres formels dans les appels et la valeur de retour.
   On décide d'appeler ces variables $a_1, a_2, \ldots$ (en nombre non borné) et une variable r pour le résultat.

   On se place d'abord dans le cas où le corps de la fonction f à compiler ne contient pas d'autres appels de procédure. Donner le schéma de génération de code intermédiaire pour l'appel de cette fonction $f(e_1, \ldots, e_n)$ et sa déclaration `fun` $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e$

   (c) Que se passe-t-il si le corps de la fonction f fait lui-même un appel de fonction ? Dans quels cas peut-on garder le même schéma de compilation ? Que peut-on faire dans le cas général pour garder un schéma de compilation correct.

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.