Compilation
Maîtrise d'Informatique 2003-2004

# Examen Final
15 janvier 2004

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 Allocation de variables locales (10 points)

On étudie un langage pour des expressions qui contiennent des définitions locales de variables. Ce langage est décrit par la grammaire suivante dans laquelle `cte` est un token correspondant à une valeur entière $(1, 2, \ldots)$, `ident` est un token correspondant à un identificateur qui peut représenter une variable locale ou bien une fonction prédéfinie et finalement `let` et `in` sont des mots clés et `=`, `(`, `,` et `)` sont des symboles. $E$ est un non-terminal permettant de dériver les expressions du langage.

$$E ::= \texttt{cte} \mid \texttt{ident} \mid \texttt{let ident = } E \texttt{ in } E \mid \texttt{ident(}E\texttt{,}E\texttt{)} \mid \texttt{(}E\texttt{)}$$

Un type CAML possible pour les arbres de syntaxe abstraite de ce langage est

```
type ast = Cte of int | Var of string
        | Let of string * ast * ast | Fun of string * ast * ast
```

Dans une expression `let` $x$ `=` $e_1$ `in` $e_2$, $x$ est une nouvelle variable qui peut être utilisée uniquement dans l'expression $e_2$. L'évaluation de l'expression complète consiste à évaluer $e_1$, à stocker la valeur obtenue dans une case mémoire $m$ puis à évaluer $e_2$ en accédant à la case mémoire $m$ à chaque utilisation de $x$.

On cherche à produire, pour chaque expression $E$, du code de la machine à pile introduite en cours dont l'exécution a pour effet de calculer la valeur de l'expression $E$ au sommet de la pile. On suppose dans la suite que dans l'expression $E$, toutes les variables locales ont un nom différent. On choisit de représenter chaque variable locale à un emplacement fixé à partir de la base de la pile.

1. On suppose donnée une fonction `index` qui à chaque variable locale $x$ de $E$, associe un entier $n$ qui représente le décalage par rapport à la base de la pile à laquelle la variable $x$ est stockée. On suppose que le nombre de cases mémoire différentes à utiliser pour stocker les variables locales de l'expression $E$ est $N$. On suppose également que pour chaque fonction prédéfinie $f$, il y a un label `label`$(f)$ auquel sont associées les instructions machines correspondant au calcul de $f$. Ces fonctions prennent en argument deux entiers et renvoient un entier, cet entier est stocké à la place du premier argument.

   Proposer une fonction `code` qui produit pour chaque expression $E$, les instructions de la machine à pile nécéssaires à l'évaluation de cette expression.

2. Soit l'expression

   ```
   let x = 1 in
   let t = let u = 2 in plus(u,3) in
   f(let y = h(2,t) in g(y,x), let z=4 in g(z,z))
   ```

On suppose que $x$ est stocké au décalage 0, $t$ et $u$ en 1 et $y$ et $z$ en 2. Donner le code produit par votre programme pour cette expression.

3. On se propose maintenant de calculer une fonction `index` correcte. On adopte la stratégie suivante : si en un point de programme, les variables visibles sont $x_0, x_1, \ldots, x_n$ alors chaque $x_i$ aura pour décalage $i$, c'est-à-dire que l'on aura $\texttt{index}(x_i) = i$.

   Proposer une fonction `analyse` qui étant donnée une expression $E$, attribue à chaque variable locale de $E$ un décalage. On pourra supposer donnée une fonction `set_index` telle que (`set_index` $n$ $x$) remplit une table utilisée par la fonction `index` en associant le décalage $n$ à la variable $x$.

4. Proposer une fonction `maxvar` qui calcule pour chaque expression $E$ le nombre de cases mémoire à réserver pour les variables locales en utilisant la stratégie précédente. Combien de cases mémoire doit-on utiliser pour l'expression de la question 2 ?.

5. Proposer un décalage pour les variables de l'expression de la question 2 qui n'utilise que deux cases mémoire en supposant que dans l'expression $f(e_1, e_2)$, l'expression $e_1$ est évaluée avant $e_2$.

6. On se propose de mettre en œuvre une analyse plus fine pour le calcul de `index` qui prenne en compte l'utilisation effective des variables. Pour réaliser cette analyse, on transforme l'expression en une suite d'affectations élémentaires de la forme $x := t$ avec $x$ une variable, et $t$ un terme simple qui peut être une constante $c$, une variable $y$ ou bien $f(y, z)$ l'application d'une fonction $f$ à deux variables $y$ et $z$.

   Pour compiler une expression $e$ on se donne une variable $r$ dans laquelle on souhaite stocker la valeur de $e$. Si $e$ est une constante ou une variable, on utilise l'affectation correspondante. Si $e$ est de la forme `let` $x = e_1$ `in` $e_2$ on engendre les affectations élémentaires pour calculer la valeur de l'expression $e_1$ dans la variable $x$ puis les affectations pour calculer $e_2$ dans la variable $r$. Si $e$ est de la forme $f(e_1, e_2)$, on crée des nouvelles variables $r_1$ et $r_2$, on engendre les affectations pour calculer $e_1$ dans $r_1$ puis $e_2$ dans $r_2$ et on ajoute l'affectation $r := f(r_1, r_2)$, si $e_1$ (resp. $e_2$) est déjà une variable, on peut l'utiliser directement.

   (a) Donner la suite d'affectations correspondant à l'expression de la question 2.

   (b) Fournir pour chaque point du programme ainsi obtenu l'ensemble des variables vivantes. Construire le graphe d'interférence et en déduire une manière d'allouer chacune des variables.

   (c) Proposer une fonction `transforme` qui étant données une variable $r$ et une expression $e$, calcule la suite d'affectations équivalente pour le calcul de la valeur de $e$ dans la variable $r$.

7. On se donne une suite d'affectations $p$, proposer une fonction `vivantes` qui calcule l'ensemble des variables vivantes au début de $p$ (qui seront donc utilisées dans $p$ avant d'être redéfinies). On pourra utiliser les notations ensemblistes usuelles : l'ensemble vide $\emptyset$, le singleton $\{x\}$, l'union de deux ensembles $e_1 \cup e_2$ et la différence entre deux ensemble $e_1 \backslash e_2$.

# 2 Filtrage (12 points)

On étudie un langage permettant la définition d'expressions par filtrage.
**Exemple** Dans un tel langage, on peut par exemple introduire un nouveau type *tree* pour représenter les arbres, avec deux constructeurs *Leaf* et *Node*.

$$\texttt{type}\ \mathit{tree}\ =\ \mathit{Leaf}\ \texttt{of int} \mid \mathit{Node}\ \texttt{of int} * (\mathit{tree} * \mathit{tree})$$

L'expression suivante, définie par filtrage, calcule la valeur à la racine de l'arbre $e$ :

$$\begin{array}{lll} \texttt{match}\ e\ \texttt{with} & \mathit{Node}(x,(y,z)) & \texttt{->}\ x \\ & \mid \mathit{Leaf}(x) & \texttt{->}\ x \end{array}$$

On peut également former des motifs plus complexes comme dans l'expression suivante notée $E_0$ dans la suite de l'exercice :

$$\begin{array}{lll} E_0\ \equiv\ \texttt{match}\ e\ \texttt{with} & \mathit{Node}(x,(y,\mathit{Leaf}(\_))) & \texttt{->}\ \texttt{plus}(x,1) \\ & \mid \mathit{Node}(1,(y,z)) & \texttt{->}\ 0 \\ & \mid \mathit{Leaf}(x) & \texttt{->}\ x \\ & \mid \_ & \texttt{->}\ 3 \end{array}$$

**Grammaire** Ce langage est décrit par la grammaire suivante dans laquelle `cte` est un token correspondant à une valeur entière, `ident` est un token correspondant à un identificateur représentant soit une variable, soit une fonction prédéfinie, soit un constructeur et finalement `let`, `in`, `match`, `with` sont des mots clés et `=`, `(`, `,`, `->`, `|`, `_` et `)` sont des symboles. $E$, $P$ et $B$ sont des non-terminaux respectivement pour les expressions, les motifs et les branches de filtrage.

$$\begin{array}{lll} E & ::= & \texttt{cte} \mid \texttt{ident} \mid \texttt{ident(}E\texttt{)} \mid \texttt{(}E\texttt{)} \mid \\ E & ::= & \texttt{let ident = } E\ \texttt{in}\ E \mid E\texttt{,}E \mid \texttt{match}\ E\ \texttt{with}\ B \\ P & ::= & \texttt{cte} \mid \texttt{ident} \mid \texttt{ident(}P\texttt{)} \mid \_ \mid P,P \mid \texttt{(}P\texttt{)} \\ B & ::= & P\texttt{->}E \\ B & ::= & P\texttt{->}E\ \texttt{|}\ B \end{array}$$

**Types** Les expressions de ce langage sont typées. Les types sont soit le type primitif `int`, soit un type produit $\tau_1 * \tau_2$ soit un identificateur correspondant à un type défini par l'utilisateur. Un tel type est introduit par une déclaration de type de la forme :

$$\texttt{type}\ t = C_1\ \texttt{of}\ \tau_1 \mid \ldots \mid C_n\ \texttt{of}\ \tau_n$$

où $C_1, \ldots, C_n$ sont les constructeurs de $t$ et $\tau_i$ est le type de l'argument de $C_i$. On suppose données deux fonctions :

- `def` associe à un identificateur de type $t$ la liste de ses constructeurs avec le type de leur argument : $\texttt{def}(t) = [(C_1, \tau_1)\ ;\ \ldots\ ;\ (C_n, \tau_n)]$
- `profil` prend en argument un identificateur $F$ (correspondant à une fonction ou un constructeur) et renvoie le profil de cet identificateur sous la forme d'un couple $(\tau, \tau')$ avec $\tau$ le type de l'argument attendu par $F$ et $\tau'$ le type du résultat de l'application de $F$ à une expression $e$ de type $\tau$. Pour le constructeur $C_i$ du type $t$ on a $\texttt{profil}(C_i) = (\tau_i, t)$. Si `plus` correspond à l'addition sur les entiers, on aura $\texttt{profil}(\texttt{plus}) = (\texttt{int} * \texttt{int}, \texttt{int})$.

**Environnement** Un environnement associe des types à des variables. On note $[x_1 : \tau_1; \ldots; x_n : \tau_n]$ l'environnement qui associe à chaque variable $x_i$ le type $\tau_i$. Soit $\Gamma$ cet environnement, on définit $\texttt{Var}(\Gamma)$ l'ensemble $\{x_1, \ldots, x_n\}$ des variables de l'environnement.

On note $\Gamma(x_i)$ le type $\tau_i$ associé à $x_i$ dans $\Gamma$. La concaténation de l'environnement $\Gamma' = [y_1 : \sigma_1; \ldots; y_m : \sigma_m]$ à $\Gamma$ est un environemment noté $\Gamma; \Gamma'$ défini comme $[x_1 : \tau_1; \ldots; x_n : \tau_n; y_1 : \sigma_1; \ldots; y_m : \sigma_m]$.

**Évaluation du filtrage** Une valeur $v$ est soit une constante $c$, soit une valeur construite $C(v')$ avec $C$ un constructeur, soit la paire $(v_1, v_2)$ de deux valeurs $v_1$ et $v_2$.
Une valeur $v$ filtre un motif $p$ dont les variables sont $\{x_1, \ldots, x_n\}$ s'il existe des valeurs $v_1, \ldots, v_n$ telles que l'expression `let` $x_1 = v_1$ `in`... `let` $x_n = v_n$ `in` $p$ s'évalue en la valeur $v$. La liste des couples $[(x_1, v_1); \ldots; (x_n, v_n)]$ est la solution du filtrage. On peut réaliser une fonction de filtrage `filtre` d'une valeur $v$ par un motif $p$ de la manière suivante :

| | | |
|---|---|---|
| $\texttt{filtre}(v, x)$ | $= [(x, v)]$ | si $x$ est une variable |
| $\texttt{filtre}(v, \_)$ | $= []$ | |
| $\texttt{filtre}(c, c)$ | $= []$ | si $c$ est une constante |
| $\texttt{filtre}(C(v), C(p))$ | $= \texttt{filtre}(v, p)$ | $C$ est un constructeur |
| $\texttt{filtre}((v_1, v_2), (p_1, p_2))$ | $= \texttt{filtre}(v_1, p_1)@\texttt{filtre}(v_2, p_2)$ | |
| $\texttt{filtre}(v, p)$ | $= \texttt{erreur}$ | dans tous les autres cas |

L'objectif de cet exercice est d'étudier différents aspects de la compilation d'un langage avec filtrage (analyse syntaxique, typage et génération de code).

1. **Analyse syntaxique** La grammaire suivante est une version simplifiée de la grammaire des motifs avec `atom` un terminal.

$$P ::= \texttt{atom} \mid P, P \mid \texttt{(}P\texttt{)}$$

   (a) La grammaire précédente est ambiguë, expliquer pourquoi.

   (b) Construire l'automate SLR(1) de cette grammaire, dans quel état a lieu le conflit, quelle est la nature de ce conflit ?

   (c) On souhaite que l'expression $p_1, p_2, p_3$ soit interprétée comme $p_1, (p_2, p_3)$, donner les transitions à effectuer à partir de l'état conflictuel pour obtenir ce comportement.

2. **Typage des motifs** Un motif $p$ est bien formé si on peut lui associer un type $\tau$ et un environnement $l = [x_1 : \tau_1; \ldots; x_n : \tau_n]$ tel que
   – $\texttt{Var}(l) = \{x_1, \ldots, x_n\}$ sont les variables introduites par le motif $p$;
   – chaque variable $x_i$ apparaît exactement une fois dans le motif $p$;
   – dans l'environnement $l$ le motif $p$ est bien formé de type $\tau$.
   On note cette relation $l \vdash p : \tau$. On la définit formellement par les règles suivantes :

$$\frac{c \text{ est une constante entière}}{[] \vdash c : \texttt{int}} \qquad \frac{}{[] \vdash \_ : \tau} \qquad \frac{}{[x : \tau] \vdash x : \tau}$$

$$\frac{l \vdash p : \tau' \quad C \text{ constructeur de profil } (\tau', \tau)}{l \vdash C(p) : \tau} \quad \frac{l_1 \vdash p_1 : \tau_1 \quad l_2 \vdash p_2 : \tau_2 \quad \texttt{Var}(l_1) \cap \texttt{Var}(l_2) = \emptyset}{l_1; l_2 \vdash p_1, p_2 : \tau_1 * \tau_2}$$

   (a) On appelle $p_1, p_2, p_3, p_4$ les motifs de l'expression $E_0$ introduite au début de l'exercice. Donner les environnements $l_1, l_2, l_3, l_4$ tels que $l_i \vdash p_i : tree$

   (b) Proposer une fonction qui étant donnés un motif $p$ et un type $t$ calcule $l$ tel que $l \vdash p : t$ si le motif $p$ est bien formé de type $t$ et échoue sinon.

3. **Typage des expressions** Les règles de typage du langage définissent sous quelles conditions une expression $e$ est bien typée de type $\tau$ dans un environnement $\Gamma$. On note cette relation $\Gamma \vdash e : \tau$.

$$\frac{c \text{ est une constante entière}}{\Gamma \vdash c : \texttt{int}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau' \quad \texttt{profil}(F) = (\tau', \tau)}{\Gamma \vdash F(e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma; x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1, e_2 : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash e : t \qquad l_1 \vdash p_1 : t \quad \Gamma; l_1 \vdash e_1 : \tau \ \ldots\ l_n \vdash p_n : t \quad \Gamma; l_n \vdash e_n : \tau}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1 \mid \ldots \mid p_n\ \texttt{->}\ e_n\ : \tau}$$

Proposer une fonction `typeof` qui étant donnés un environnement $\Gamma$ et une expression $e$ calcule le type $\tau$ tel que $\Gamma \vdash e : \tau$ si un tel type existe et échoue sinon.

4. **Élimination du filtrage** Le filtrage permet de faire des constructions de programme par cas (comme la conditionnelle) et d'accéder aisément aux sous-expressions d'un terme contenant des paires et des constructeurs. On se propose de transformer une expression contenant du filtrage en une expression sans filtrage mais utilisant les constructions supplémentaires suivantes :
   - des conditionnelles booléennes `if` $E$ `then` $E$ `else` $E$;
   - des fonctions `fst` et `snd` d'accès aux composantes d'une paire : si $e$ est une expression de type $\tau_1 * \tau_2$ qui s'évalue en la paire $(v_1, v_2)$ alors `fst`$(e)$ (resp. `snd`$(e)$) est une expression de type $\tau_1$ (resp. $\tau_2$) qui s'évalue en la valeur $v_1$ (resp. $v_2$);
   - Une fonction de test d'égalité sur les entiers notée $e_1 = e_2$.
   - Une fonction `test` qui étant donnés une expression $E$ et un constructeur $C$, renvoie vrai si l'expression $E$ s'évalue en une valeur $C(v)$ et faux sinon;
   - Pour chaque constructeur $C$ de profil $(\tau', \tau)$, une fonction $\texttt{proj}_C$ de profil $(\tau, \tau')$ telle que $\texttt{proj}_C(E)$ s'évalue en la valeur $v$ si l'expression $E$ s'évalue en une valeur $C(v)$ et échoue sinon.

   (a) Proposer une fonction `transforme` qui prend en argument une expression $e$, un motif $p$, une expression *succès* pouvant mentionner les variables du motif $p$ et une expression *échec*, et qui calcule une expression ne comportant pas d'expression `match` mais pouvant contenir les fonctions et constructions données ci-dessus, dont le comportement est le même que celui de l'expression : `match` $e$ `with` $p$ `->` `succès` | `_` `->` `échec`.
   C'est-à-dire que si $e$ s'évalue en une valeur $v$, si `filtre`$(v, p)$ renvoie la séquence $[(x_1, v_1), \ldots, (x_n, v_n)]$ alors le résultat attendu s'évalue comme `let` $x_1 = v_1$ `in`$\ldots$ `let` $x_n = v_n$ `in` *succès* sinon le résultat attendu est *échec*.
   On construira la fonction `transforme` par cas suivant le motif $p$. Les cas du caractère _ et d'une constante $c$ peuvent se définir de la manière suivante :

   $\texttt{transforme}(e, \_, \textit{succès}, \textit{échec}) \ = \textit{succès}$
   $\texttt{transforme}(e, c, \textit{succès}, \textit{échec}) \ = \texttt{if eq}(e, c)\ \texttt{then}\ \textit{succès}\ \texttt{else}\ \textit{échec}$

   (b) En déduire une fonction `élimine` qui étant donnée une expression $e$ remplace les constructions `match` par des conditionnelles simples et des fonctions de test et de projection. On supposera que dans les constructions `match` de $e$, les filtrages sont exhaustifs c'est-à-dire qu'il y a toujours un motif pour lequel le filtrage réussit.

   (c) Donner la traduction de l'expression `match` $e$ `with` $\textit{Node}(x, \_, \_)$`->` $x \mid \textit{Leaf}(x)$`->` $x$.

5. **Génération de code** On se propose maintenant de compiler le langage sans filtrage dans le code de la machine à pile vue en cours. Pour cela on décide de représenter les valeurs entières par des entiers, et les valeurs paires $(v_1, v_2)$ ou constructeur $C(v)$ par des adresses dans le tas. Cette adresse pointe sur un bloc de deux cases mémoire. Dans le cas d'une paire, la première case contient la valeur $v_1$ et la seconde la valeur $v_2$ ($v_1$ et $v_2$ peuvent être une adresse ou un entier) . Dans le cas d'une valeur constructeur $C(v)$, la première case contient un entier représentant l'étiquette du constructeur $C$ (attribuée de manière statique) et la seconde contient la valeur $v$.

On suppose qu'une fonction `tag` fournit pour chaque constructeur $C$, le numéro de l'étiquette de $C$.

Soit `code` la fonction qui étant donnée une expression sans filtrage produit le code pour la machine à pile permettant de calculer la valeur de cette expression. Expliciter la définition de `code` dans les cas suivants :
Application $C(e)$ et fonction de test $\mathtt{test}(C, e)$ avec $C$ un constructeur et $e$ une expression; paire $e_1, e_2$ de deux expressions $e_1$ et $e_2$; projection $\mathtt{fst}(e)$ de l'expression $e$ sur la première composante.

## Machine à pile

Les instructions de cette machine à pile sont un sous-ensemble de celles décrites dans le cours.

| | |
|---|---|
| `PUSHI` $n$ | empile la constante entière $n$. |
| `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 $gp$. |
| `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 $a + n$. |
| `ALLOC` $n$ | alloue un bloc de taille $n$ dans le tas et empile l'adresse de ce bloc. |
| `PUSHN` $n$ | empile $n$ valeurs nulles sur la pile. |
| `POPN` $n$ | 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` $f$ | 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 `LABEL` $f$. |