# Programmation – examen

*mercredi 18 janvier 2006, 9h-12h*

*ne sont autorisées que les notes de cours et de TP*

# 1 Pour démarrer

## 1.1 Égalités

1. Soient les deux définitions suivantes:

   `let f x y = x = y = 3`

   `let f x y = 3 = x = y`

   Il y en a forcément une qui plante et l'autre pas: de quoi cela dépend-il? (NB: il n'est pas demandé de savoir *laquelle* plante parmi ces deux-là)

2. On suppose que c'est la première fonction qui plante dans la question précédente. On saisit la définition:

   `let g x y = x = y = true = false = true`

   (a) Quel est le type de `g`?

   (b) Que renvoie `g 32 52`?

## 1.2 Questions de cours: prospectus

Le prospectus qui est distribué avec cet énoncé propose quelque chose de très alléchant, mais qui est loin d'être totalement conforme à la réalité du systèmes de types de Caml.

Pour chacun des sept points qui sont mentionnés dans le prospectus, confirmez (et dans ce cas commentez en indiquant à quelle propriété du système de types de Caml il est fait référence) ou infirmez (et dans ce cas expliquez pourquoi c'est de la pubilicité mensongère, en illustrant au besoin par un exemple) ce qui est annoncé.

Pour éviter de recopier chaque point, répondez dans l'ordre en numérotant vos réponses de 1 à 7 (quitte à laisser certaines réponses vides si vous ne savez pas).

# 2 Arbres binaires

Dans toute cette partie, on va travailler avec le type des arbres binaires contenant des entiers aux nœuds, défini par:

```
type t = E | N of t*int*t
```

Le constructeur `E` (pour *Empty*) correspond aux feuilles, et `N` sert à représenter les nœuds (avec un fils gauche, une valeur entière et un fils droit).

## 2.1 Parcours d'arbres et *backtracking*

Nous allons utiliser l'interface suivante:

```
module type Parcours = sig
  type t0
  type enum
  val start :  t0 -> enum
  val step :  enum -> int*enum
  exception Exit
end
```

en étudiant différents types de parcours d'arbre. Le type `enum` sert à énumérer les éléments stockés dans les nœuds d'un arbre. La fonction `start` met en place l'énumération: un premier élément est prêt à être fourni. Un appel à la fonction `step` renvoie l'élément courant de l'énumération, et se positionne sur l'élément suivant. L'exception `Exit` est levée lorsque `step` échoue (il n'y a plus d'élément dans l'énumération).

**Parcours infixe** Voici comment est codé le parcours infixe d'un arbre de type `t`.

```
module type ParcoursArbre = Parcours with type t0 = t

module Infixe :  ParcoursArbre = struct
  type t0 = t
  type enum = End | More of int*t*enum

  let rec left t e = match t with
   | E -> e
   | N (l, x, r) -> left l (More (x,r,e))
  let start t = left t End

  let step = function
  | End -> raise Exit
  | More (x, r, e) -> x, left r e

  exception Exit
end
```

1. Définir, en s'inspirant de ce qui est fait pour le parcours infixe, un module de signature `ParcoursArbre` pour le parcours préfixe, où l'on examine la valeur située au nœud avant de poursuivre dans le sous-arbre gauche puis le sous-arbre droit.
2. Même question avec le parcours postfixe, où l'on examine l'élément au nœud après avoir examiné les deux fils.
3. Définir un foncteur (ou module paramétré) prenant en argument un module de type `Parcours` et définissant une fonction `s52 :  t -> bool` qui cherche s'il existe un sous-ensemble de l'ensemble des entiers stockés dans un arbre (de type `t`) dont la somme des éléments vaut `52`. Le code de `s52` ne devra pas s'appuyer sur la manière dont le type `t` est implémenté, mais utilisera les valeurs définies dans le module de type `Parcours`.

   NB: le but dans cette question est de programmer une solution élégante, quitte à ce qu'elle soit inefficace.

## 2.2 La fermeture éclair

Le *zipper* est une structure de données pour la programmation fonctionnelle proposée par G. Huet. On peut en illustrer le fonctionnement en étudiant comment programmer la modification d'un arbre binaire qui consiste à remplacer la valeur stockée en un nœud par une autre valeur (au passage, on ne souhaite pas garder l'arbre originel, tel qu'il était avant la modification).

1. Expliquer en quoi la solution naturelle en C, qui consiste à représenter l'arbre à l'aide de structures et de pointeurs, est a priori préférable pour réaliser cette opération à la solution sans `ref` de Caml, où l'on s'appuie sur le type `t` défini plus haut.

   Donner les définitions C qui permettent de représenter l'arbre correspondant à la valeur

   `a = N(E,32,N(E,52,E))` (définir des types pour les arbres, et indiquer la définition de `a`).

   Pour tâcher de remédier à ce problème, on va travailler avec les types de données suivants:

```
type path = Top | Left of path*int*t | Right of t*int*path

type location = t*path
```

Une valeur de type `location` de la forme `(a,p)` représente un endroit dans l'arbre: `a` est le sous-arbre situé en cet endroit, et `p` représente le chemin parcouru depuis la racine de l'arbre jusqu'au nœud courant.

Au départ, on se positionne à la racine de l'arbre:

```
let start :  t -> location = fun a -> a,Top
```

On définit les deux fonctions suivantes:

```
 let go_down_left = function
| E, _ -> failwith "raté"
| N (l,x,r), p -> l, Left (p,x,r)
```

```
 let go_down_right = function
| E, _ -> failwith "raté"
| N (l,x,r), p -> r, Right (l,x,p)
```

2. Dans quel sens est stocké le chemin reliant la racine de l'arbre au nœud courant dans la seconde composante d'un élément de type `location`?

3. Définir trois autres fonctions de navigation dans l'arbre `go_left`, `go_right` et `go_up`, toutes trois de type `location->location`, permettant de se déplacer respectivement vers le frère gauche, vers le frère droit et vers le père (lorsque ceux-ci existent).

4. Écrire une fonction `modif :  location -> int -> t` telle que `modif l k` renvoie l'arbre (de type `t`) obtenu en remplaçant la valeur stockée dans le nœud courant (représenté par `l`) par `k`.

   NB: on veut récupérer l'arbre originel, et non simplement le sous-arbre, modifié.

5. En quoi l'approche suivie est-elle intéressante du point de vue de la complexité, vis-à-vis des remarques faites initialement sur la comparaison avec la solution "C+pointeurs"?

# 3 Fonctions et efficacité en espace

Dans cette partie, on va étudier Camelot, qui est un langage prototype ayant été proposé récemment. Camelot peut être vu comme une variante de Caml dans laquelle on fournit au programmeur des moyens de mieux contrôler la gestion de l'espace en mémoire.

On va s'intéresser au type suivant pour les listes d'entiers:

```
type intlist = Nil | Cons of int * intlist
```

La fonction qui met une liste à l'envers peut s'écrire en Camelot ainsi:

```
 let rec rev l acc = match l with
   | Nil -> acc
   | (Cons (h,t))@d -> rev t (Cons (h,acc)@d)

let reverse l = rev l Nil
```

Dans la définition de `rev`, les annotations "`@d`" servent à indiquer que le filtrage est *destructif*, et que l'espace dans lequel est initialement stocké `Cons (h,t)` va être réutilisé pour stocker `Cons (h,acc)`.

1. Quelle place dans le tas va nécessiter l'exécution d'un appel à la fonction `reverse` (il peut être utile de lire la question 2 avant de répondre à celle-ci)?

2. Le programmeur de Camelot averti définit en réalité la fonction `rev` en remplaçant la deuxième ligne ci-dessus par `| Nil@_ -> acc` Deviner pourquoi et expliquer.

3. La définition correspondante en OCaml est exactement la même, en omettant simplement les “`@d`”.

   Expliquer *précisément* quelles sont les différences entre ce qui se passe en OCaml et ce qui se passe en Camelot du point de vue de la représentation des listes dans le tas.

4. Définir une fonction Camelot `subst : int -> int -> intlist -> intlist` telle que `(subst k m l)` renvoie la liste `l` dans laquelle tous les `k` ont été remplacés par des `m`, les modifications ayant lieu *en place*.

5. Commençons par préciser qu'en Camelot, si on n'utilise pas l'opérateur “`@`”, le comportement est le même que celui d'OCaml.

   Donner un exemple (n'utilisant que des listes d'entiers, et pas des `ref`) montrant que la possibilité de programmer avec des `@` met en péril la propriété de *transparence référentielle.*

6. Le système de types de Camelot permet de garantir que certains programmes utilisant les `@` ne sont pas nocifs du point de vue de la transparence référentielle. Il s'appuie pour cela (entre autres) sur ce qu'on appelle un critère de *linéarité* dans l'utilisation des variables: dans `let x = e1 in e2`, on veut dans certains cas s'assurer que `x` est utilisé au plus une fois dans `e2`.

   Comment comprenez-vous cette condition, avec le peu qui vous en est dit? Proposez un exemple de programme où cette condition est violée et où cela conduit à la perte de la transparence référentielle.

7. On définit la fonction `max` ainsi:

   ```
   let rec max l m = match l with
     | [] -> m
     | x::xs -> max xs (if x>m then x else m)
   ```

   Le critère de linéarité susmentionné fait que Camelot rejette le code suivant

   ```
   let l = [2;3;4] in (subst (max l 0) 0 l)
   ```

   (a) Pourquoi ce rejet est-il regrettable?

   (b) Proposez un raffinement du système de types permettant de ne pas rejeter le code ci-dessus. Il ne vous est pas demandé de fournir une description formelle précise du système de types, mais essayez d'expliquer aussi clairement que possible comment procède l'analyse, et dans quel cas celle-ci peut améliorer le système de types de Camelot.

## 4 Engendrer des fonctions à partir de types

L'inférence de types permet à l'utilisateur d'entrer un programme non typé et de profiter du fait que le système calcule pour lui un type. Dans cette partie, on s'intéresse au mécanisme inverse: on donne un type et on veut trouver des programmes ayant ce type. Ainsi, à “`int -> bool -> int`” on veut pouvoir répondre “`fun k -> fun b -> k`”, ou “`fun k -> fun b -> 32`” – par exemple: il y a bien entendu une infinité d'autres réponses possibles. On appelle cela le problème de l'*habitation des types* (on cherche à savoir par qui un type est habité).

On veut programmer tout cela en Caml. On va s'intéresser à des types définis par:

```
type ty = I | B | F of ty*ty
```

Un élément du type `ty` représente soit les entiers (`I`), soit les booléens (`B`), soit, s'il est de la forme `F(t1,t2)`, représente le type habituellement noté `t1 -> t2`. Ainsi, le type `int -> bool -> int` sera représenté par `F(I,F(B,I))`.

Pour programmer le problème de l'habitation de type, on va utiliser les définitions de types suivantes:

```
type co = Var of string | App of co*co | CI of int | CB of bool;;
type fo = string list * co;;
```

Le type `fo` sert à représenter les fonctions: la liste contient les noms des arguments de la fonction (liste vide signifie qu'on n'a pas une fonction mais une constante), et la seconde composante est le corps de la fonction. Le corps de la fonction est soit une variable, soit une constante entière (`CI`), soit une constante booléenne (`CB`), soit une application, de la forme `App(c1,c2)`, où `c1` et `c2` sont de type `co`.

Ainsi, les deux fonctions données en exemple plus haut seront représentées respecivement par `(["k";"b"], Var("k"))` et `(["k";"b"], CI(32))`. Et `(["x";"y"], App(Var("y"),Var("x")))` représentera la fonction `fun x -> fun y -> y x`. On peut également remarquer que l'on ne peut représenter de fonction récursive, puisque les fonctions n'ont pas de nom dans la représentation adoptée.

1. Donner trois éléments de type `ty` auxquels peut correspondre la réponse

   `(["x";"y"], App(Var("y"),Var("x")))`.

2. Une valeur de type `fo`, qui est de la forme `(l,t)`, n'est pas toujours bien formée, du fait de la possibilité d'avoir dans `t` des "`Var(m)`" avec `m` qui n'apparaît pas dans `l`.

   Donner le code d'une fonction `bonne_formation : fo -> bool` qui teste la bonne formation.

3. Une solution immédiate à la question de l'habitation des types consiste à ne répondre qu'avec des fonctions constantes (qui n'utilisent pas leurs arguments).

   Donner le code d'une fonction `constante : ty -> fo` qui fournit pour chaque élément de type `ty` l'une de ces réponses peu intéressantes.

4. On considère maintenant la définition suivante:

   ```
   type ty' = AA | BB | FF of ty'*ty'
   ```

   Une valeur de type `ty'` représente un type polymorphe ayant au plus deux variables de type (si vous voulez, `AA` représente un `'a` de Caml et `BB` représente `'b`).

   On se pose la même question que précédemment (en répondant toujours à l'aide de valeurs de type `fo`).

   (a) Le problème correspondant à `FF(AA,AA)` admet-il une solution? Si oui, exhibez une solution.

   (b) Même question avec `FF(AA,BB)`.

   (c) Proposez une manière de calculer des réponses à un problème représenté par une valeur de type `ty'`. Votre réponse devra prendre la forme d'un algorithme (décrit de manière *précise*), éventuellement accompagné de code si vous arrivez à pondre quelque chose de clair et concis. Bien entendu, fournir du code est un plus qui rapportera des points supplémentaires, mais un algorithme clair et bien expliqué constituera une réponse satisfaisante.

   Dans cette question, la voie est ouverte, il n'y a pas de solution absolue. Ne vous embarquez pas dans des choses trop raffinées, et ne passez pas une heure à réfléchir. À vous de trouver une réponse raisonnablement maline sans sacrifier la lisibilité. Vous pouvez au passage indiquer des améliorations possibles qui ne figurent pas dans la solution que vous proposez.

5. Comment une solution au problème posé avec des valeurs de type `ty'` peut-elle être utilisée pour le problème posé avec des `ty`? Écrivez le code faisant la liaison entre les deux problèmes, quitte à supposer l'existence d'une fonction (dont vous indiquerez le type) répondant à la question 4.