Université Paris Diderot — Programmation Fonctionnelle
L2 Math/Info, L3 Info, M1 Linguistique — Année 2012-2013

Examen

Jeudi 10 janvier 2013

Motivez bien vos réponses. On recommande de *bien lire* l'énoncé d'un exercice avant de commencer à le résoudre.

Tout document papier est autorisé. Les ordinateurs, les téléphones portables, comme tout autre moyen de communication vers l'extérieur, doivent être éteints. Le temps à disposition est de 3 heures.

Les exercices doivent être rédigés en fonctionnel pur : ni références, ni tableaux, ni boucles `for` ou `while`, pas d'enregistrements à champs mutables. Chaque fonction ci-dessous peut utiliser les fonctions prédéfinies (sauf indication contraire), et/ou les fonctions des questions précédentes.

Cet énoncé a quatre pages.

Exercice 1 (Expressions). Qu'affiche-t-il quand on entre l'une après l'autre les expressions suivantes dans le toplevel Caml ? Donner le type de chaque expression ou définition (ou le message d'erreur si l'expression est mal typée), ainsi que sa valeur et son effet de bord (ou l'exception levée) le cas échéant. Justifier si nécessaire.

1. `let x = 1 in x = 2 || not (2 <> 3) ;;`
2. `let x = 1 and y = 2 in max x y ;;`
   On rappelle la définition de la librairie standard : `val max : 'a -> 'a -> 'a`.
3. `let x = 1 and y = x in max x y ;;`
4. `x + 1 ;;`
5. `let null f = function 0 -> false | n -> f n ;;`
6. `null (fun x -> x mod 2 = 0) 4 ;;`
7. `null (fun x -> x) 4 ;;`
8. `match [1;2] with a :: b :: c -> c | _ -> raise Not_found ;;`
9. `match [1;2] with [_] -> true | _ :: _ -> false ;;`
10. `let m = max (0,1) ;;`
11. `m (2,3) ;;`
12. `let r p = let (x, y) = p in (y, x) ;;`
13. `r (false, 'a') ;;`
14. `let pr x = print_string (x ^ "\n") ;;`
15. `pr "je vaux "; 21 ;;`

Exercice 2 (Listes et sommes).

1. Écrire l'implémentation d'une fonction `is_sum_cons` qui prend en entrée un entier n et une liste d'entiers l, et qui renvoie `true` si n peut être obtenu comme la somme de deux nombres *consecutifs* dans l, et qui renvoie `false` sinon. Par exemple, pour $l =$`[2;5;3;5]`, elle renverra `true` si $n = 8$ (car $8 = 5 + 3$) ; elle renverra `false` si $n = 5$.

2. Écrire l'implémentation d'une fonction `is_sum` qui prend en entrée un entier n et une liste d'entiers l et qui renvoie `true` si n peut être obtenu comme la somme de deux nombres (pas forcément consécutifs) dans l, et qui renvoie `false` sinon.
3. Écrire l'implémentation d'une fonction `knapsack` qui prend en entrée un entier n et une liste d'entiers l, et qui renvoie `true` si n peut être obtenu comme la somme d'éléments de l, chacun compté au plus une fois, et qui renvoie `false` sinon. Par exemple, si l est `[2;5;3;5]` la réponse sera `true` si n est 12 ($= 2 + 5 + 5$) mais elle sera `false` si n est 6. Vous commencerez la récursion par le cas de la liste vide. Dans ce cas la réponse sera `true` seulement si $n = 0$.
4. Écrire l'implémentation d'une fonction `knaplist` qui prend en entrée un entier n et une liste d'entiers l, et qui renvoie une valeur de type `int list option`. La fonction doit renvoyer `None` si n ne peut pas être obtenu comme somme d'éléments de l. Sinon la fonction doit renvoyer `Some pos`, où `pos` est la liste d'entiers qui correspondent aux positions des éléments de l qui ont comme somme n (s'il y a plusieurs possibilités on en choisira une). Par exemple, si l est `[2;5;3;5]`, alors la réponse sera `Some [0;1;3]` si n est 12 ($= 2+5+5$), elle sera `None` si n est 6, elle sera `Some [1]` ou `Some [3]` ou `Some [0,2]` si n est 5. Finalement si n est 0, la reponse sera `Some []`.

Exercice 3. On considère le type suivant, permettant de représenter en OCaml des arbres binaires dont les nœuds internes sont étiquetés par des valeurs quelconques :

```
type 'a tree = F | N of 'a * 'a tree * 'a tree
```

1. Sans déclarer de fonctions auxiliaires et sans vous servir des fonctions prédéfinies, écrire :

   ```
   forall_labels : ('a -> bool) -> 'a tree -> bool
   ```

   telle que `forall_labels p a` renvoie `true` si et seulement si chaque étiquette `x` de `a` vérifie `(p x) = true`. A partir de cette seule fonction, écrire une fonction

   ```
   is_uniform : 'a -> 'a tree -> bool
   ```

   tel que `is_uniform v a` renvoit `true` si et seulement si chaque étiquette de `a` est égale à `v`. Cette fonction ne devra pas être déclarée comme récursive, mais doit déléguer entièrement la récurrence à la précédente.
2. Toujours sans fonctions auxiliaires ou prédéfinies, écrire :

   ```
   forall_subtrees : ('a -> 'a tree -> 'a tree -> bool) -> 'a tree -> bool
   ```

   telle que `forall_subtrees p a` renvoie `true` si et seulement si pour chaque sous-arbre de `a` de la forme `N(x, g, d)`, on a `p x g d = true`.
   Rappelons qu'un arbre est un *peigne droit* s'il est réduit à une feuille, ou si son fils gauche est une feuille et son fils droit est un peigne droit. A partir de la seule fonction `forall_subtrees`, écrire :

   ```
   est_peigne_droit : 'a tree -> bool
   ```

   tel que `est_peigne_droit a` renvoit `true` si et seulement si `a` est un peigne droit, et déléguant entièrement la récurrence à la fonction `forall_subtrees`.
3. On définit pour les arbres un itérateur de la manière suivante :

   ```
   let rec fold_tree fn vf a = match a with
       F -> vf
     | N(n, g, d) -> fn n (fold_tree fn vf g) (fold_tree fn vf d);;
   ```

Noter que `vf` spécifie la valeur à renvoyer si l'arbre `a` est réduit à une feuille. Si c'est un noeud interne, l'itérateur est appliqué aux sous-arbres avec les mêmes arguments, et les valeurs de retour sont combinées à la valeur étiquetant la racine via la fonction `fn`.

Quel est le type de `fold_tree` ?

4. En utilisant uniquement `fold_tree`, et en lui déléguant la récurrence, écrire :

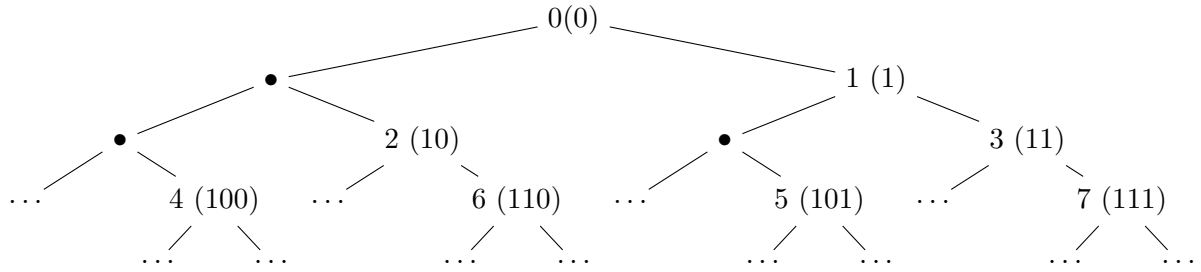
   ```
   somme_etiquettes : int tree -> int
   ```

   renvoyant la somme des étiquettes d'un arbre étiqueté par des entiers. Avec les mêmes contraintes, écrire :

   ```
   map_tree : ('a -> 'b) -> 'a tree -> 'b tree
   ```

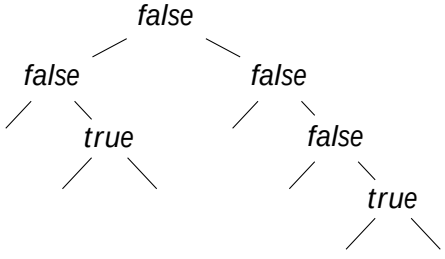
   telle que `map_tree f a` renvoie l'arbre obtenu en remplaçant chaque étiquette `x` de `a` par `(f x)`.

Exercice 4. Dans cet exercice, on se propose de coder une structure de données pour représenter les ensembles d'entiers positifs ou nuls appelée *Arbre de Patricia*. L'idée consiste en une structure arborescente basée sur l'écriture binaire de l'entier : si l'entier vaut 0 il est stocké à la racine ; sinon, on le stocke dans le sous-arbre gauche si son dernier chiffre binaire est 0 (i.e. s'il est pair) et dans le sous-arbre droit si son dernier chiffre binaire est 1 (i.e. s'il est impair), et l'on poursuit avec les chiffres suivants. À titre indicatif, voici où se retrouvent placés les huit premiers entiers (l'écriture binaire est indiquée entre parenthèses) :



Notez que les chiffres de l'écriture binaire sont donc utilisés de droite à gauche (i.e. du moins significatif au plus significatif). Notez également que certains nœuds (indiqués par le symbole •) ne sont pas utilisés car l'entier correspondant a déjà été rencontré plus haut. Ainsi le noeud • le plus à droit dans l'arbre ci-dessus correspondrait à 01, c'est-à-dire 1.

On choisit de coder les arbres de Patricia en Caml par des arbres avec des feuilles vides et des nœuds contenant un booléen indiquant la présence ou l'absence de l'entier correspondant. Les nœuds ne correspondant pas à un entier contiennent la valeur *false*. L'ensemble $\{2, 7\}$ sera donc représenté par l'arbre suivant :



Rappel En Caml, on teste si un entier `n` est pair (resp. impair) par la condition `n mod 2 = 0` (resp. `n mod 2 = 1`).

1. Donner un type Caml `pat` codant les arbres de Patricia.
2. Écrire en Caml une fonction `cherche` : `int` → `pat` → `bool` testant l'appartenance d'un entier à un ensemble (par la suite on identifie la notion d'arbre de Patricia et celle d'ensemble d'entiers).
3. Écrire en Caml une fonction `ajoute` : `int` → `pat` → `pat` ajoutant un entier à un ensemble.
4. En utilisant la fonction précédente, écrire en Caml une fonction `construit` : `int list` → `pat` construisant un ensemble à partir d'une liste d'entiers.
5. Écrire en Caml une fonction `union` : `pat` → `pat` → `pat` réalisant l'union de deux ensembles.
6. Écrire en Caml une fonction `elements` : `pat` → `int list` retournant la liste de tous les éléments d'un ensemble (l'ordre n'importe pas mais chaque élément doit apparaître une fois et une seule dans la liste).

Pour votre culture : Les arbres de Patricia sont utilisés (dans une version un peu plus compacte et efficace) dans les routeurs TCP/IP pour diriger les données entre toutes les machines constituant le réseau Internet.