Licence STS, semestre 3 — 2008–2009
Algorithmique et approche fonctionnelle (Info 211) — Durée : 2h
4 pages

Examen du 29 Octobre 2008

Les notes de cours, de TD et de TP manuscrites ainsi que les supports de cours distribués cette année sont les seuls documents autorisés.

Veuillez lire attentivement les questions. Veuillez rédiger proprement, clairement et de manière concise et rigoureuse. Le barème est indicatif.

1 Typage (4 points)

Les fonctions suivantes sont-elles bien typées ? Si oui, donner leur type, sinon préciser pourquoi.

```
let f1 x y = x ^ y

let f2 x y = if x then 0 else y +. 1.0

let rec f3 x = x :: f3 (x + 1)

let f4 x y z = [ x ; y @ z ]

let f5 g h x = g (h (x + 1)) + h (x + 2)

let rec f6 x = f6 (not x)
```

2 Évaluation (4 points)

1. Étant donnée la fonction `mystere1: int -> int -> int * int` suivante :

   ```
   let rec mystere1 a  b =
     if a < b then (0, a) else
       let (x, y) = mystere1 (a - b) b in (x + 1, y)
   ```

   Donner les résultats des évaluations de `mystere1 10 2` et `mystere1 9 4`.

2. Étant donnée la fonction `mystere2 : 'a list -> 'a list` suivante :

   ```
   let rec mystere2 l =
     match l with
         []  -> []
       | [x] -> []
       | x::y::l2 -> x :: mystere2 l2
   ```

Donner les résultats des évaluations de `mystere2 [1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7]` et `mystere2 [1 ; 2 ; 3]`

3. Étant donnée la fonction `mystere3 : int list -> int list` suivante :

```
let mystere3 l =
  let rec mystint l s =
    match l with
        []  -> []
      | x::l2 -> (s + x) :: mystint l2 (s + x)
  in
  mystint l 0
```

Donner le résultat de l'évaluation de `mystere3 [12; 3; 8; 9]`.

3 Problème : Comment rendre la monnaie ? (12 points)

Le problème que nous souhaitons résoudre est celui qui consiste à programmer un appareil pour rendre la monnaie. Dans un but de simplification, on ne manipulera que des centimes et donc uniquement des entiers.

Ce problème est en fait un application directe du problème de la *partition d'entiers* : étant donné un entier positif n, une *partition* de n est une liste $[n_1; n_2; \ldots; n_s]$ d'entiers tels que $n_1 \geq \cdots \geq n_s > 0$ et $n_1 + \cdots + n_s = n$. Par exemple, les partitions de l'entier 4 sont les listes $[4]$, $[3; 1]$, $[2; 2]$, $[2; 1; 1]$ et $[1; 1; 1; 1]$. Par la suite, on dira que chaque entier n_i est une *part* de n.

Ainsi, si l'on suppose, dans un premier temps, que l'appareil possède une *infinité* de pièces de 2€, 1€, 0.50€, 0.20€ et 0.10€, rendre une monnaie de n euros, c'est trouver une partition de l'entier $100 \times n$ dont les parts sont dans l'ensemble $\{200, 100, 50, 20, 10\}$. Nous verrons à la fin de ce problème comment calculer la monnaie à rendre quand l'appareil possède un nombre fini de pièces.

Questions préliminaires (6 points). On représentera les partitions par des listes d'entiers.

1. En utilisant les fonctions `print_int` et `print_newline`, qui permettent d'afficher respectivement un entier et un retour à la ligne, écrire une fonction `affiche : int list -> unit` qui affiche une liste d'entiers, séparés deux à deux par un point virgule, et suivie d'un retour à la ligne, *en respectant l'ordre des éléments dans la liste.*
2. Écrire une fonction `affiche_rev: int list -> unit` qui affiche une liste d'entiers de la même manière, *mais dans l'ordre inverse.* La liste ne devra être parcourue qu'*une seule fois* (et donc on ne pourra pas utiliser la fonction précédente combinée avec une fonction pour renverser la liste !).
3. Écrire une fonction `verification: int -> int list -> bool` qui prend en argument un entier `n` et une liste `l` et qui vérifie que `l` est bien une partition de `n`.

4. Écrire une fonction *récursive terminale* `generation: int -> int list` qui prend en argument un entier `n` et qui retourne la liste d'entiers `[n;n-1;...;1]`. Vous pourrez vous aider pour cela d'une fonction *interne* `gen_rec : int list -> int -> int list`.

Énumération des partitions (3 points). On souhaite maintenant définir un algorithme pour énumérer l'ensemble des partitions d'un entier `n` dont les parts sont dans une liste d'entiers `parts`. On représentera l'ensemble des partitions par une liste de listes d'entiers.

Le principe de l'algorithme est le suivant :

- si $n = 0$ alors la seule partition est la partition vide, représentée par la liste vide `[]`, et l'algorithme renvoie la liste `[[]]` ;
- sinon, on distingue les deux cas suivants :
  - si `parts` est vide (il n'y a plus de parts possibles pour construire la partition), on renvoie la liste vide ;
  - sinon, `parts` est de la forme `x::s` et l'algorithme retourne l'union des deux ensembles calculés de la manière suivante :
    a) si $x \leq n$ alors on construit récursivement l'ensemble des partitions de l'entier $n - x$ avec des parts *toujours* dans `parts` et on ajoute (en tête) x à toutes ces partitions ; si $x > n$ on retourne simplement l'ensemble vide ;
    b) on construit l'ensemble des partitions de `n` où les parts sont dans l'ensemble `s`.

1. Écrire une fonction `ajout : 'a -> 'a list list -> 'a list list` qui prend en arguments une valeur `v` et une liste de listes `l` et qui ajoute `v` en tête de tous les éléments de `l`.
2. En utilisant la fonction précédente, écrire une fonction `enumeration : int -> int list -> int list list` qui prend en arguments un entier `n` et une liste d'entiers `parts` et qui implémente l'algorithme d'énumération décrit ci-dessus.

Rendre la monnaie (3 points). Afin de rendre la monnaie avec un appareil ne possédant qu'un nombre fini de pièces, on va modifier l'algorithme d'énumération décrit dans le paragraphe précédent de manière à prendre en compte la quantité limitée de chaque pièce pouvant être utilisée pour rendre la monnaie. Pour cela, une pièce sera représentée par un couple (p, i) où p représente la valeur de la pièce et i, un entier *strictement positif*, la quantité encore disponible de cette pièce.

Par exemple, un appareil possédant 4 pièces de 2€, 3 pièces de 1€, 5 pièces de 50 centimes, et 1 pièce de 10 centimes sera représenté par la liste suivante :

```
[ (200, 4); (100, 3); (50, 5); (10,1) ]
```

1. En vous inspirant de l'algorithme d'énumération décrit dans le paragraphe précédent, écrire une fonction `rendre_monnaie : int -> (int * int) list -> int list list` qui prend en arguments un entier `n` et une liste de pièces disponibles `pieces` et qui retourne toutes les façons de rendre `n` centimes d'euros en fonction des pièces disponibles dans `pieces`.

2. Que retourne l'expression

```
rendre_monnaie 1000 [ (200, 3) ; (100, 3) ; (50, 5) ] ?
```

3. Étant données une liste de pièces `pieces` et `m` l'une des solutions renvoyées par `rendre_monnaie n pieces`, écrire une fonction `nouveau_monnayeur : int list -> (int * int) list -> (int * int) list` telle que `nouveau_monnayeur m pieces` retourne la nouvelle liste de pièces de l'appareil, c'est-à-dire qui enlève de la liste `pieces` les pièces de `m`. On prendra soin de pas laisser des couples de la forme $(p, 0)$ dans le résultat.