Université Paris Diderot
L2 Math/Info, L3 Info, M1 Lingustique

Programmation Fonctionnelle
Année 2010-2011

Examen 2ème session

Mercredi 15 juin 2011

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.

Exercice 1 [Expressions et Types] Pour chacune des expresssions ci-dessous :

- si elle n'est pas typable, expliquez en quelques mots pourquoi,
- si elle est typable, donnez son type et sa valeur. Décrire la valeur en français (une ligne) quand il s'agit d'une valeur qui ne peut pas être affichée.

1. `let x = 1 in let x = 2 in x + 3`
2. `let x = 1 in let x = x + 2 in x + 3`
3. `let x = 1 and y = 2 in [(x, y)]`
4. `let x = [1] and y = [2] in [(x, y)]`
5. `[]@[]`
6. `[]::[]`
7. `[]::[[]]`
8. `(fun x -> (fun y -> x + y)) 1`
9. `(fun x -> ((fun y -> x + y) 1))`
10. `List.map (fun (x, y) -> (y, x)) [(2, 'a'); (0, 'c')]`
11. `List.map (fun x -> [x]) [(2, 'a'); (0, 'c')]`
12. `(fun x -> if x < 2 then (+) else (-)) 1 2 3`

Exercice 2 [Listes avec occurrences] Vous pouvez dans cet exercice définir des fonctions locales ou auxiliaires, mais ne pouvez utiliser aucune fonction prédéfinie de la bibliothèque `List`.

Toute liste commence par un certain nombre d'occurrences d'une certaine valeur, suivies d'un certain nombre d'occurrences d'une autre valeur, suivies d'un certain nombre d'occurrences d'une autre valeur, etc. Le contenu d'une liste de la forme

$$[\overbrace{\texttt{v1; ...; v1}}^{\texttt{n1 fois}}\ \overbrace{\texttt{v2; ...; v2}}^{\texttt{n2 fois}}\ \texttt{...;}\ \overbrace{\texttt{vp; ...; vp}}^{\texttt{np fois}}\ ]$$

peut en fait être décrit précisément par la liste de couples

```
[(v1, n1); ...; (vp, np)]
```

interprétable par : "la liste contient `n1` fois la valeur `v1`, suivi de `n2` fois la valeur `v2`, ..., suivi de `np` fois la valeur `vp`". Par exemple, la liste

```
['a'; 'a'; 'b'; 'c'; 'c'; 'c'; 'a'; 'a']
```

peut être décrite par `[('a', 2); ('b', 1); ('c', 3); ('a', 2)]`.

1. Écrire la fonction `reconstruire : ('a * int) list -> 'a list` reconstruisant une liste à partir de sa description sous forme de liste de couples.
2. Écrire la fonction `decrire : 'a list -> ('a * int) list` transformant une liste en sa description la plus courte sous forme de liste de couples.
3. La description d'une liste n'est pas unique : `['a'; 'a'; 'a']` est décrite par `[('a',3)]`, mais aussi par `[('a',1); ('a',2)]`, ou encore par

   ```
   [('b',0); ('a',1); ('c',0); ('a';2); ('d',0)].
   ```

   De même la liste vide peut être décrite par `[]`, mais aussi par `[(3.14,0); (0.0,0)]`, `[("abc",0)]`, etc. Une liste de couples sera dite *optimale* si elle ne contient pas de couples de la forme `(v,0)`, et si elle ne contient pas deux couples de suite mentionnant la même valeur à gauche.
   Écrire la fonction `optimiser : ('a * int) list -> ('a * int) list` transformant une liste de couples en une liste optimale, décrivant la même liste. Vous devrez écrire cette fonction sans reconstruire la liste !
4. Écrire la fonction `ajouter : 'a -> ('a * int) list -> ('a * int) list` prenant un élément x et une liste optimale l, et renvoyant une liste optimale décrivant la liste obtenue en ajoutant x en tête de la liste décrite par l. Cette fonction doit être écrite sans effectuer de reconstruction de liste.
5. (Question bonus) Écrire `supprimer : int -> ('a * int) list -> ('a * int) list` telle que `(supprimer i l)`, où l est une liste optimale, renvoie une liste optimale décrivant la liste obtenue en supprimant le i-ème élément de la liste décrite par l. Si cet élément n'existe pas, la liste renvoyée est simplement égale à l. Une fois encore, cette fonction doit être écrite sans effectuer de reconstruction de liste.

Exercice 3 [Entiers longs en OCaml]

Les entiers machines en Caml sont bornés : ceux-ci sont, sur une machine 32 bits, compris entre -2^{30} et $2^{30}-1$. Pour certaines applications, on a cependant besoin d'entiers non bornés (par exemple pour chercher de grands nombres premiers ou pour certaines applications de cryptographie).

Dans cet exercice, on appellera *entiers longs* les entiers naturels (*i.e.* les entiers positifs ou nuls) qu'on souhaite représenter (qu'on distinguera des *entiers machines*, les entiers implantés en standard en Caml).

Le type `entier_long` sera défini par

```
type entier_long = int list
```

Un entier naturel sera en effet représenté par la liste de ses chiffres : la liste $[\alpha_0;\ldots;\alpha_n]$ sera une représentation de l'entier naturel $\alpha_0 + \alpha_1 \times 1000^1 + \ldots + \alpha_n \times 1000^n$. On convient que la liste vide représente l'entier 0. Ainsi $[23;521;4]$ est une représentation de l'entier 4521023 (car $4521023 = 23+521\times1000+4\times1000^2$). Remarquez qu'un entier n'a pas une unique représentation. Ainsi 1234 peut être représenté par $[234;1]$ mais aussi par $[234;1;0]$, $[1234]$, $[1234;0]$ ou encore $[-766;2]$.

On appelle *représentation canonique* d'un entier naturel tout élément de type `entier_long` ne contenant que des entiers machines compris entre 0 et 999, de longueur minimale. On admettra

que cette représentation est unique. Ainsi, [234; 1] est la représentation canonique de 1234. [234; 1; 0] n'est pas canonique car elle n'est pas de longueur minimale. [1234] n'est pas canonique non plus car elle ne contient pas que des entiers machines compris entre 0 et 999, de même que [−766; 2]. Notez que la représentation canonique de 0 est la liste vide ([ ]).

Il peut être utile, pour répondre aux questions de cet exercice, de réutiliser des réponses à des questions précédentes.

1. Écrire une fonction `affiche : entier_long -> unit` prenant en argument une représentation canonique et affichant à l'écran sa représentation décimale (la représentation que vous utilisez tous les jours). Ainsi `affiche [67; 345; 2; 1]` affichera 1002[illegible]345067.
2. Écrire une fonction `coupe_zero : entier_long -> entier_long` qui, étant donnée une représentation d'un entier long, coupe tous les zéros qui se trouvent à la fin de cette représentation. Ainsi `coupe_zero [123; 4; 0; 0]` devra retourner `[123; 4]`.
3. Écrire une fonction `canonise : entier_long -> entier_long` qui, lorsqu'on lui donne la représentation d'un entier long ne contenant que des entiers machines positifs ou nuls, la met sous forme canonique.
4. Écrire une fonction `plus : entier_long -> entier_long -> entier_long` qui prendra en entrée les représentations canoniques de deux entiers n et m et calculera la représentation canonique de $n + m$.
5. Écrire une fonction `mult_chiffre : int -> entier_long -> entier_long` qui prendra en entrée un entier machine c compris entre 0 et 999 et une représentation canonique d'un entier n et rendra en sortie une représentation canonique de $c \times n$. NB : votre capacité à utiliser `List.map` sera pris en compte dans la notation de votre réponse (on rappelle que le type de cette fonction est `('a -> 'b) -> 'a list -> 'b list`).
6. Écrire une fonction `exp_mille : entier_long -> int -> entier_long` qui, étant donnés la représentation d'un entier long n et un entier machine k, renvoie la représentation de $n \times 1000^k$. Si l'entier long est donné sous forme canonique, le résultat devra l'être également.
7. (Question bonus) Écrire une fonction `mult : entier_long -> entier_long -> entier_long` qui prend en argument les représentations canoniques de deux entiers longs, et calcule la représentation canonique de leur produit.

Exercice 4 [Arbres binaires et Graphisme] On considère dans cet exercice les arbres binaires non vides dont les nœuds et les feuilles sont étiquetés par des chaînes de caractères. Un arbre est donc :

- soit une feuille, encapsulant une chaîne de caractères,
- soit un noeud, encapsulant une chaîne de caractères, avec deux fils.

Pour afficher un arbre à une position (x, y), on commence par afficher la chaîne s de sa racine. La chaîne doit être centrée en x, et son bord supérieur doit être placé à la hauteur y. Lorsque la racine est un nœud, on dessine ensuite ses deux fils (voir la figure 1).

Les positions verticales d'affichage des deux fils sont les mêmes, à une distance b (constante, fixée) du bord inférieur de s. Les deux fils sont séparés par un espace horizontal de taille a (constante, fixée). Les positions horizontales des deux fils sont choisies de manière à ce que la racine de l'arbre se trouve exactement au milieu du diagramme (pour cela, il faut en particulier connaître les largeurs d'affichage des deux fils).

Finalement, on dessine deux traits allant du milieu du bord inférieur de s jusqu'aux positions d'affichage des deux fils.

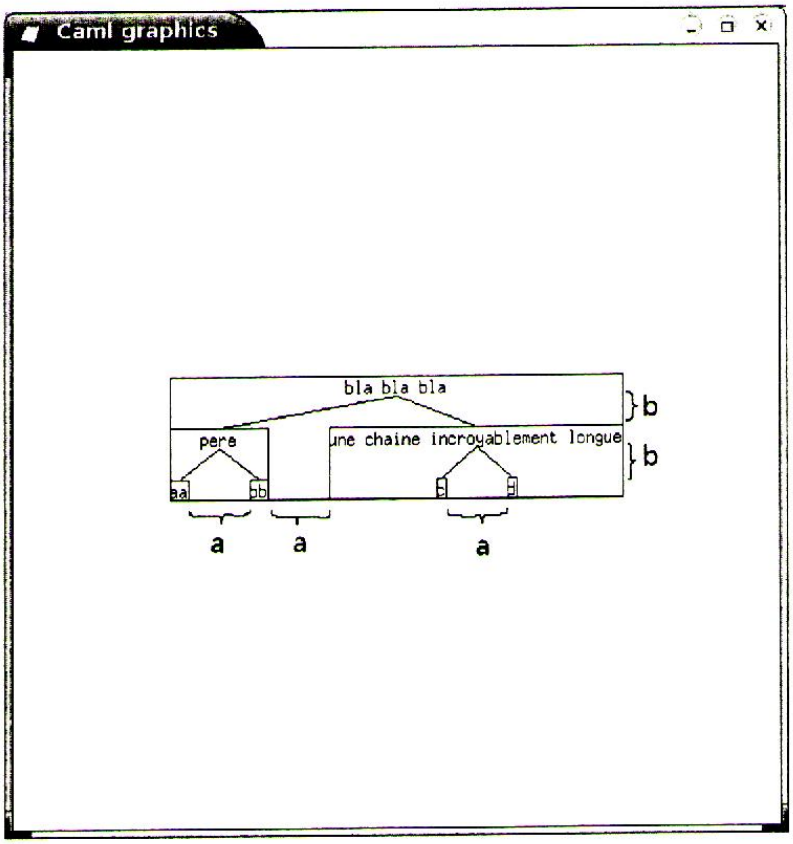

FIGURE 1 – Affichage d'un arbre. Les boites rectangulaires ne sont pas à afficher par la fonction demandée ; elles sont indiquées ici seulement afin d'illustrer l'algorithme d'affichage.

Questions :

1. Donner la définition OCaml d'un type `arbre` pour la représentation des arbres.
2. Ecrire une fonction `taille : arbre -> (int * int)` prenant en argument un arbre et renvoyant une paire consistant en la largueur et la hauteur de son affichage suivant les contraintes ci-dessus.

   On supposera les constantes `a` et `b` déjà définies (de type `int`). La fonction prédéfinie `text_size : string -> (int * int)` permet d'obtenir les dimensions (largeur, hauteur) de l'affichage d'une chaîne de caractères.
3. Ecrire une fonction `afficher : arbre -> unit` prenant en argument un arbre et affichant cet arbre à l'écran à l'aide des fonctions du module `Graphics`. La taille de la fenêtre graphique sera supposée stockée dans deux constantes entières `xsize` et `ysize`. L'arbre devra être centré dans cette fenêtre. Servez-vous de la fonction `taille` de la question précédente. Déclenchez une exception si l'arbre dépasse les dimensions de la fenêtre graphique.