

Devoir surveillé de Programmation Fonctionnelle

22 décembre 2007

durée 1h30 - documents autorisés

Exercice 1. *Typage***Question 1.** Donnez le type des expressions suivantes :

1. **let** rec **f** **x y z** = **if** **x = y** **then** **z** **else** **1. +. (f (x+1) y z)**;;
2. **let** **g** **x y z** = **(x y) z** ;;
3. **let** **h** **x y z** = **(x y) (y z)** ;;

Question 2. Donnez des expressions qui possèdent les types suivants :

1. **('a -> 'b -> 'c) -> 'a -> 'b -> 'c**;
2. **('a -> 'b -> 'c) -> 'a -> ('a -> 'b) -> 'c**.

Exercice 2. *Approximation de racines carrées*

On souhaite dans cet exercice calculer la racine carrée d'un réel positif a sans utiliser la fonction prédéfinie **sqrt**. Le réel \sqrt{a} peut être approché à l'aide de la suite u_n définie par :

$$u_0 = 1 \quad \text{et pour } n \geq 0 \quad u_{n+1} = \frac{u_n + a/u_n}{2}$$

Question 1. Réalisez une fonction **approx_sqrt a n** de type **float -> int -> float** qui calcule la valeur du terme u_n .

À titre d'exemples, voici quelques appels à cette fonction.

```
# approx_sqrt 2. 0;;
- : float = 1.
# approx_sqrt 2. 1;;
- : float = 1.5
# approx_sqrt 2. 2;;
- : float = 1.416666
# approx_sqrt 2. 3;;
- : float = 1.414215
```

($\sqrt{2} \approx 1.41421356$)

Question 2. Sans définir de nouvelles variables, construisez la liste des approximations successives de $\sqrt{5}$ pour n allant jusqu'à 4.

Exercice 3. *Accouplements*

Question 1. Réalisez la fonction **applique** qui, à partir d'une liste de fonctions de type **'a -> 'b** et d'un élément de type **'a**, calcule la liste de type **'b list** des images de cet élément par chacune des fonctions. À titre d'illustration, voici un exemple d'appel à cette fonction.

```
# applique [sqrt; log; exp] 1. ;;
- : float list = [1.; 0.; 2.71828182845904509]
```

Question 2. Réalisez maintenant une fonction **accouple** de type **'a list -> ('a * 'a) list** qui à partir d'une liste construit la liste de tous les couples d'éléments qu'il est possible de construire. Par exemple

```
# accouple [1;2;3] ;;
- : (int * int) list =
[(1, 1); (2, 1); (3, 1); (1, 2); (2, 2); (3, 2); (1, 3); (2, 3); (3, 3)]
```

(l'ordre des couples n'est pas imposé).

Il existe une solution élégante qui utilise les fonctions **map** et **flatten** du module **List**, ainsi que la fonction **applique**.

Exercice 4. Codage de Huffman

Le codage de Huffman est un codage binaire fréquemment utilisé en compression des données. On va s'intéresser ici au décodage, c'est-à-dire à la décompression.

Un codage de Huffman peut être représenté par un arbre binaire dont les feuilles sont étiquetées par les caractères codés. Le mot binaire associé à chaque caractère est obtenu en parcourant l'arbre de la racine jusqu'à la feuille correspondant au caractère et en notant 0 une descente à gauche, et 1 une descente à droite.

Par exemple, l'arbre de la figure 1 représente le codage de Huffman de trois caractères **b**, **t** et **e**

t	00
b	01
e	1

et le mot **011001** correspond au texte **bete**.

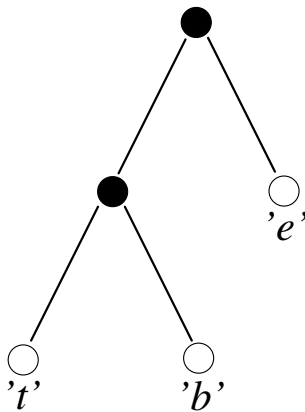


FIG. 1 – Un codage de Huffman

Dans cet exercice, les mots du texte sont représentés par des listes de caractères, et les mots binaires par des listes d'entiers 0 ou 1.

Question 1. Donnez en CAML une déclaration du type **arbre** afin de représenter les arbre binaires avec des caractères aux feuilles.

Question 2. Définissez une variable dont la valeur représente l'arbre de la figure 1.

Question 3. Réalisez une fonction **string_of_char_list** de type **char list -> string** afin de convertir un texte donné sous forme de liste en *string*. Vous pouvez utiliser la fonction **escaped : char -> string** du module **Char** afin de convertir un caractère en une chaîne.

Question 4. Réalisez la fonction **decomresse** qui, à partir d'un arbre, décomresse le mot binaire et retourne le texte obtenu sous forme de liste de caractères.

Aide : vous pouvez d'abord créer une fonction récursive de type **:int list -> arbre -> char * int list** nommée **decode** qui calcule le décodage d'un seul caractère et retourne *en même temps* le reste de la liste à décoder.

Solutions

Exercice 1

Q 1 .

Solution

```
val f : int -> int -> float -> float = <fun>
val g : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
val h : (('a -> 'b) -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

Q 2 .

Solution

```
fun a b c -> a b c ;;
fun a b c -> a b (c b);;
```

Exercice 2

Q 1 .

Solution

```
let rec approx_sqrt a = function
| 0 -> 1.
| n -> let un = approx_sqrt a (n - 1)
      in
      (un +. a/.un)/.2. ;;
```

Q 2 .

Solution

```
# List.map (approx_sqrt 5.) [0;1;2;3;4];;
- : float list =
[1.; 3.; 2.33333333333333348; 2.23809523809523814; 2.23606889564336342]
```

Exercice 3

Q 1 .

Solution

```
let rec applique l x =
  match l with
  | [] -> []
  | f::l -> (f x)::(applique l x) ;;
```

Q 2 .

Solution

```
let accouple l =
  List.flatten (List.map (applique (List.map (fun x y -> x,y) l)) l) ;;
```

Exercice 4

Q 1 .

Solution

```
type arbre = Feuille of char
           | Noeud of arbre*arbre;;
```

Q 2 .

Solution `let monarbre = Noeud(Noeud(Feuille('t'), Feuille('b')), Feuille('e'));;`

Q 3 .

Solution

```
let rec string_of_char_list = function
| [] -> ""
| h::t -> (Char.escaped h)^(string_of_char_list t);;
```

Q 4 .

Solution

```
(* decode : renvoie le premier caratere qui est encode
* par code ainsi le reste du code
*)
```

```
let rec decode code = function
| Feuille (c) -> (c,code)
| Noeud (g,d) -> if (List.hd code) = 1
                  then decode (List.tl code) d
                  else decode (List.tl code) g;;
```

```
(* decompresse : realise la decompression
* de la liste de booleans
*)
```

```
let rec decompresse code arbre =
  if code = []
  then []
  else
    let (caractere,restecode) = decode code arbre
    in
      let listecaracteres = decompresse restecode arbre
      in caractere::listecaracteres;;
```