Interrogation écrite 4

INF 201 — IMA4— 27/04/2023 — 20 minutes

Problème. (/21) Dans cet exercice il est **interdit** d'écrire des fonctions récursives, il faudra utiliser à la place les schémas d'ordre supérieur du module `List`. Vous êtes autorisés à appeler les fonctions en omettant le `List.`, par exemple `hd` au lieu de `List.hd`.

En mathématiques, on nomme fraction continue une expression de la forme:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cdots}}}$$

où $a_0 \in \mathbb{Z}$ et $a_{i,i>0} \in \mathbb{N}$. On peut avoir un nombre fini ou infini de a_i. Par exemple le très célèbre nombre Pi s'écrit avec une infinité de coefficients:

$$\pi = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cdots}}}$$

En pratique on écrira pour simplifier $\pi = [3, 7, 15, 1, 292, \ldots]$.

Question 1. (/1) Proposer un type `frac_cont` pour représenter une fraction continue.

```
type frac_cont = float list;;
```

Question 2. (/2) Définir en utilisant une fonction anonymé un opérateur de composition `>>` en OCaml qui permet de faire la composition de deux fonctions, c'est-à-dire tel que `g>>f` représente la fonction $g \circ f$.

```
let (>>) g f = (fun x -> g (f x));;
```

Vu en TD la semaine précédente, tout le monde devrait avoir bon.

Question 3. (/1) Préciser le profil de cet opérateur:

```
('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

Même chose ! Mais presque tout le monde tombe dans le piège:

```
('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

Question 4. (/4) À l'aide uniquement de cet opérateur et de fonctions du module `List`, implémenter une fonction `dernier` qui renvoie le dernier élément d'une liste et une fonction `sans_dernier` qui renvoie la liste privée de son dernier élément. Vous définirez comme une application partielle: **la variable qui représente la liste ne doit pas apparaître.**

Tout simplement:

```
let dernier = List.hd >> List.rev;;

let sans_dernier = List.rev >> List.tl >> List.rev;;
```

Question 5. (/4) Implémenter une fonction `fc_vers_R` qui convertit une fraction continue en un Réel. Par exemple:

```
# fc_vers_R  [3.;7.;15.;1.;292.];;

- : float = 3.14159265301190249
```

Votre fonction devra utiliser un schéma de type `fold_right` et vous pouvez utiliser `dernier` pour extraire la graine ainsi que `sans_dernier` pour appeler votre schéma.

Si le fonctionnement de `fold_right` est compris, pas de difficulté, à revoir pour les autres…

```
let fc_vers_R l =
  List.fold_right (fun x acc-> x+.1./.acc) (sans_dernier l) (dernier l);;
```

Question 6. (/3) Implémenter maintenant une fonction `couper n l` qui renvoie les `n` premiers éléments d'une liste.

Similaire à ieme vu en TD et correction envoyée par mail à tout le monde.

```
let couper n l =
  let _,rl = List.fold_left (fun (i,lacc) x -> if i<=n then (i+1, lacc@[x])
                                    else (i+1,lacc))
    (1,[]) l in rl;;
```

Question 7. (/2) Déduire des fonctions `couper` et `fc_vers_R` une fonction `approx` qui renvoie l'approximation de d'un réel en coupant le développement en fraction continue au nième terme. Vous utiliserez l'opérateur de composition défini ci-dessus et vous définirez comme une application partielle.

Encore une fois il suffit de composer deux fonctions…

```
let fc_vers_R_approx n = fc_vers_R >> (couper n);;
```

Certains réel ont un développement en fraction continue particulier, c'est le cas par exemple de la contante d'euler e.

`e = [2.0; 1.0; 2.0; 1.0; 1.0; 4.0; 1.0; 1.0; 6.0; 1.0; 1.0; 8.0; 1.0; 1.0]`, le premier terme est 2 et ensuite il s'agit d'une répétition de triplet du type $(1, 2n+1, 1)$. On rappelle que la fonction `List.init` permet de générer une liste d'éléments à partir d'une liste d'entiers, par exemple:

```
# List.init 10 (fun k -> 2*k);;

- : int list = [0; 2; 4; 6; 8; 10; 12; 14; 16; 18]
```

Question 8. (/4) À l'aide de la fonction `List.init` implémenter une fonction `e_fc` qui génére les `n` premiers termes du DFC de e, attention à bien renvoyer une liste de `float`, dans le bon sens et qui commence par 2 !

Fonction un peu plus difficile car nous n'avions pas manipulé `List.init`, il fallait aussi penser à convertir en float avec `List.map`:

```
let e_fc n =
  let terms =
    List.init n (fun k ->
      if k mod 3 = 2 then 2 * ((k+1)/3)
      else 1) in
  List.map float_of_int (2::(List.tl terms));;
```

Bilan de cette catastrophe

Moyenne:	4.26
Écart-type:	4.6
Max:	15.5
Min:	0
<10	23

Tableau.