info421 : Programmation fonctionnelle
TD 2 : tuples et listes, un peu de filtrage

Responsables : Pierre Hyvernat et Krzysztof Worytkiewicz
Laboratoire de mathématiques de l'université de Savoie
email : Pierre.Hyvernat@... / Krzysztof.Worytkiewicz@... (...@univ-savoie.fr)
http://lama.univ-savoie.fr/~hyvernat/
http://lama.univ-savoie.fr/~worytkiewicz/

Exercice 1 : tuples.

Question 1. Écrivez une fonction `f : ('a * 'b) * 'c -> 'a * ('b * 'c)` de deux manieres différentes.

Question 2. Écrivez plusieurs fonctions différentes de type `(int -> int) -> int`.

Même question pour `(int -> int) -> int -> int`

Question 3. Les fonctions Caml suivantes sont valides. Quel sont leurs types, et que calculent-elles ?

```
let f x = match x with
  (x,y) -> fst x
let g x y = match x with
  y -> snd y
let h x y = match (x,y) with
  z -> fst z
```

Question 4. Caml possède une fonction `max` de type `int -> int -> int` qui permet de trouver la plus grande de deux valeurs.

- Écrivez une fonction `max3valeurs` dont le type est `int->int->int->int`
- Écrivez une fonction `maxPaire` dont le type est `int*int -> int` sans utiliser les opérateurs de comparaison.

Question 5. On peut transformer une fonction à deux arguments de type `int` en une fonction à un seul argument, de type `int * int` ; et vice et versa. Écrivez les deux fonctions génériques de transformation et donnez leurs types. (La seconde fonction s'appelle `curry` et la première `uncurry`).

Discutez des avantages de la première (plusieurs arguments) et la seconde forme (un seul argument) pour une fonction.

Question 6. Soit la fonction récursive (Fibonacci)

$$f(0) = 0 \qquad f(1) = 1 \qquad f(n+2) = f(n+1) + f(n)$$

Comment pourriez vous compter le nombre d'appels recursifs nécessaires pour calculer $f(n)$?

Exercice 2 : listes.

Question 1. Écrivez une fonction `longueur : 'a list -> int` qui calcule la longueur d'une liste.

Question 2. Écrivez une fonction `applique` qui prend une fonction et une liste en arguments, et qui applique la fonction à tous les éléments de la liste.

Quel est le type de cette fonction ?

Question 3. Écrivez une fonction `concatene : 'a list -> 'a list -> 'a list` qui concatène deux listes ensemble.

Question 4. Écrivez une fonction `renverse : 'a list -> 'a list` qui renverse une liste.

Que pensez-vous de la complexité de votre fonction ?

Question 5. Toutes les fonctions précédentes* suivent le schéma suivant :
- si la liste est vide, on renvoie une valeur donnée "`o`"
- sinon, on fait un appel récursif sur la fin de la liste, et on applique une fonction donnée "`f`" avec comme arguments :
  . le premier élément de la liste,
  . le résultat de l'appel récursif.

Une version générique de cette fonction est la fonction :

```
let rec reduit o f l =
  match l with
    [] -> o
  | x::xs -> let r = (reduit o f xs)
             in f x r
```

Rajoutez des annotations de type sur la déclaration de cette fonction et donnez son type complet tels que Caml l'afficherait.

Question 6. Reprogrammez les fonctions précédentes uniquement à partir de la fonction `reduit`.

Question 7. Toutes ces fonctions sont définies dans la librairie `List` :
- `longueur` s'appelle `length`,
- `applique` s'appelle `map`,
- `concatene` s'appelle `@` et on l'utilise de maniere infixe,
- `renverse` s'appelle `rev`,
- `reduit` s'appelle `fold_right`.

L'interprète Caml nous donne le type suivant pour la fonction `fold_right` :

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Que pouvez-vous en déduire sur l'ordre des arguments de `fold_right` par rapport à ceux de notre fonction `reduit` ?

Question 8. La fonction `fold_right` permet de calculer

```
f a1 (f a2 (f a3 ( ... (f an o))))
```

à partir de la liste `[a1 ; a2 ; a3 ; ... ; an]`.

La librairie `List` possède une fonction `fold_left`, dont le type est :

```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Essayez de reprogrammer la fonction `fold_left`.

* sauf peut-être la fonction `renverse`, suivant la manière dont vous l'avez programmée