info401 : Programmation fonctionnelle
TD 5 : récursion terminale, exceptions

Pierre Hyvernat
Laboratoire de mathématiques de l'université de Savoie
bâtiment Chablais, bureau 22, poste : 94 22
email : Pierre.Hyvernat@univ-savoie.fr
www : http://www.lama.univ-savoie.fr/~hyvernat/
wiki : http://www.lama.univ-savoie.fr/wiki

Exercice 1 : récursion terminale

Il est souvent possible de transformer une fonction récursive en fonction récursive terminale en utilisant un *accumulateur* qui permet de faire passer les résultat entre les appels récursifs. Ceci permet d'économiser de la mémoire car Caml ne sauvegarde pas le contexte lors des appels récursifs terminaux.

Question 1. Programmez la fonction `somme : int -> int` qui calcule la somme $1+2+...+n$ de manière récursive terminale.

Question 2. Programmer la fonction `factorielle` de manière récursive terminale.

Question 3. On peut écrire une fonction `iter` qui itère une fonction un certain nombre de fois : `iter : ('a -> 'a) -> int -> ('a -> 'a)`.

Une version de cette fonction est donnée par :

```
let rec iter f n x =
  if n = 0
  then x
  else f (iter f (n-1) x)
```

Programmez une version récursive terminale pour cette fonction.

Question 4. La fonction `fold_right` sur les listes est définie comme suit :

```
let rec fold_right f l o =
  match l with
   [] -> o
 | a::l -> f a (fold_right f l o)
```

et son type est `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`.

Écrivez une fonction *similaire* récursive terminale : `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

Exercice 2 : calcul de la puissance d'un nombre

Question 1. Écrivez une fonction `puissance : float -> int -> float` qui permet de calculer x^n...

Combien d'opérations sont nécessaires pour faire ce calcul ?

Quelle quantité de mémoire est nécessaire pour faire ce calcul ?

Question 2. Une version un peu plus efficace (pour la mémoire) est d'utiliser un *accumulateur*. Programmez une fonction `puissance_acc x n acc : float -> int -> float -> float` qui pour les arguments x, n et a calcule $a \times x^n$.

Pour éviter de garder des valeurs en mémoire entre les appels récursifs, débrouillez-vous pour faire des appels récursifs *terminaux*.

Combien d'opérations sont nécessaires pour faire ce calcul ?

En sachant que Caml ne sauvegarde pas le contexte lors des appels récursifs terminaux, quelle quantité de mémoire est nécessaire pour faire ce calcul ?

Servez-vous de cette fonction pour redéfinir une fonction puissance.

Question 3. Pour limiter le nombre de calculs lors du calcul de la puissance, on va maintenant utiliser l'astuce suivante :

$$\begin{cases} x^0 = 1 \\ x^{2n} = (x^n)^2 \\ x^{2n+1} = x(x^n)^2 \end{cases}$$

Écrivez une fonction `puissance_chinoise : float -> int -> float` pour utiliser cette remarque. (Pas besoin de faire une version récursive terminale pour le moment.)

Question 4. Essayer maintenant de faire une fonction qui calcule la puissance avec :
- une complexité en temps similaire la celle de la question 3,
- une complexité en espace similaire à celle de la question 2.

Exercice 3 : exceptions

Question 1. On reprend le même type pour les dictionnaires que dans le TD4 :
`type ('c,'v) dict = ('c * 'v) list`

Réécrivez la fonction `supprime : 'a -> ('a,'b) dict -> ('a,'b) dict` en utilisant une exception lorsque que la case que l'on supprime n'existe pas. (N'oubliez pas de déclarer l'exception.)

Redéfinissez l'ancienne version de `supprime` (qui ne lève pas d'exception) en utilisant la nouvelle version.

Question 2. Même question pour la fonction `recherche_cle`.

Question 3. En sachant que Caml possède une exception `Stack_overflow`, reprogrammez la fonction `sous_ensembles` du TD4 / TP2 en rajoutant que si Caml n'a pas assez de mémoire, la fonction renvoie `[]`.

Question 4. Programmez une fonction `element : int -> 'a list -> 'a` qui renvoie le n-ème éléments d'une liste.

Question 5. On peut programmer la fonction `apparait : 'a -> 'a list -> bool` avec un `fold_right` ou un `fold_left` :

```
let apparait a l = List.fold_right (fun x b -> x=a || b) l false
let apparait_bis a l = List.fold_left (fun b x -> x=a || b) false l
```

À votre avis, quel est la meilleure version ?

Comment ces fonctions se comparent-elles avec la version à la main suivante :

```
let rec apparait_tris a l =
  match l with
    [] -> false
  | b::l -> if a=b then true else apparait_tris a l
```

Corrigez le problème en utilisant une exception pour définir `apparait` avec `fold_right` ou `fold_left`.