info401 : Programmation fonctionnelle
TP 3 : tas, récursion terminale et la suite de Conway

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

Ce TP, comme les suivants, sera noté. Je ne demande aucun compte rendu à part, mais seulement un *unique* fichier Caml, commenté.

- Votre fichier doit contenir du code valide et ne doit pas provoquer d'erreur lorsqu'on l'évalue avec l'interprète Caml. (Testez avant de me l'envoyer : si votre fichier ne s'évalue pas correctement, vous perdez automatiquement 5 points sur votre note finale.)
- Je me réserve le droit d'enlever des points de manière exponentielle pour les code mal écrit.*
- Votre fichier devra contenir un commentaire contenant votre nom, prénom et filière. (Idem, si ce n'est pas le cas, vous perdez 5 points sur votre note finale.)
- Pour m'envoyer votre TP, utilisez uniquement le formulaire dont l'adresse est : (lien disponible sur le wiki)
  `http://www.lama.univ-savoie.fr/~hyvernat/envoi-TP.php`

Partie 1 : les tas (1h30)

Un *tas* est une structure de donnée permettant d'optimiser la recherche du minimum dans un ensemble de données. (Il existe une variante qui permet d'optimiser la recherche du maximum.)

Un *tas* est simplement un arbre binaire qui vérifie la Propriété inductive suivante :
- l'arbre vide est un tas,
- sinon, la racine est le plus petit élément,
- et les deux fils sont des tas.

Par rapport aux arbres binaires de recherche, cette structure ne permet pas d'optimiser la recherche d'un élément, car on ne sait pas dans quel sous-arbre il faut chercher.

Question 1. Déclarez un type `'a tas` pour les tas ; et écrivez une fonction qui teste si un élément de ce type est effectivement un tas.

Question 2. Écrivez une fonction `min_tas : 'a tas -> 'a * 'a tas` qui permettra de calculer :
- le minimun d'un tas (première composante),
- le tas sans son minimum.

Remarque : attention à gérer les cas problématiques correctement.

Comme pour les arbres binaires de recherche, la structure de tas est véritablement efficace quand l'arbre sous-jacent est équilibré, c'est à dire quand la différence entre les tailles des branches est au plus 1.

* J'enlève 2 points pour le premier TP, puis 4 points pour le second, puis 8 points pour le troisième, etc.

Question 3. Écrivez une fonction `equilibre : 'a tas -> bool` qui vérifie si un arbre est un tas équilibré.

Remarque : essayez de minimiser le nombre de parcours de votre tas.

L'insertion dans un arbre de recherche équilibré est un peu compliquée (mais pas trop). Pour les tas, on peut utiliser le fait que les fils gauche et droit peuvent être permutés pour créer des tas équilibrés !

Question 4. Écrivez la fonction `insere : 'a -> 'a tas -> 'a tas` qui insère un élément dans un tas.

Cette fonction devra toujours insérer l'élément dans le fils droit, mais elle permutera le fils droit et le fils gauche. Cela permet de mélanger le tas et de le garder équilibré.

Question 5. Écrivez la fonction `insere_liste : 'a list -> 'a tas -> 'a tas` qui insère une liste d'éléments dans un tas, et testez sur des exemples pour vérifier que vos tas sont effectivement équilibrés.

Question 6. Programmez le *tri par tas* : pour trier une liste par ordre croissant, on met tous les éléments dans un tas, puis on vide le tas pour récupérer les éléments dans l'ordre.

Comparez ce tri avec le tri fusion et le tri par insertion du TP1.

Remarque : dans un langage impératif (C, Java, Pascal), les tas sont en général implémentés de manière très différentes...

Partie 2 : exception (30 m)

Question 1. En utilisant `List.fold_left`, programmez une fonction `somme : int list -> list` et une fonction `produit : int list -> int` qui calculent respectivement la somme et le produit des éléments d'une liste.

Question 2. Idem, mais pour une fonction `produit7` qui calcule le produit dans $\mathbf{Z}/7\mathbf{Z}$ (modulo 7).

Question 3. Testez `produit7` sur des listes aléatoires d'éléments de $\mathbf{Z}/7\mathbf{Z}$ (en utilisant `Random.int 7` pour générer un tel élément) de grande taille. Que constatez-vous ? Expliquez.

Question 4. Réécrivez `produit7-bis` avec `List.fold_left`, mais en utilisant une exception pour améliorer le calcul.

Partie 3 : la suite de Conway (2h00)

La suite de Conway est la suite suivante :

```
1
11
21
1211
111221
312211
13112221
...
```

Si vous ne connaissez pas cette suite, passez quelques minutes à essayer de deviner comment on la calcule avant de passer à la suite.

Pour calculer le terme suivant, on *lit* simplement le terme précédent :
- `1` donne "un `1`", soit `11`,
- `11` donne "deux `1`", soit `21`,
- `21` donne "un `2` et un `1`", soit `1211`,
- etc.

Question 1. Programmez une petite fonction `conway : int list -> int -> int list` qui calcule le n-ème terme de la suite à partir d'une suite initiale d'entiers.

Question 2. La taille du terme de la suite semble augmenter à chaque étape. Écrivez une fonction `conway_croissance : int list -> int -> float list` qui calcule la facteur de croissance de la suite.

Par exemple, `conway_croissance [1] 7` donnera `[2. ; 1. ; 2. ; 1.5 ; 1. ; 1.33333333]` car la taille des termes était 1, 2, 2, 4, 6, 6, 8.

Que constatez-vous ?

Question 3. Écrivez une fonction de `conway_term` qui soit récursive terminale.

Vérifiez bien que vous obtenez la même chose qu'avec `conway`.

Un terme de la suite u_n peut être *scindé* en deux morceaux $u_n = g_n \cdot r_n$ si les parties g_n et r_n n'interagissent pas et que l'on peut calculer l'évolution de u_n indépendamment sur g_n et sur r_n :

$$u_{n+k} = g_{n+k} \cdot r_{n+k}$$

Si $n \geq 2$, alors on peut scinder u_n en $g_n \cdot r_n$ ssi :
- le dernier entier de g_n est supérieur ou égal à `4` et le premier entier de r_n est inférieur ou égal à `3` ;
- le dernier entier de g_n est `2`, et r_n est de la forme :
  - `[1 ;x]` ou `[1 ;x ;y ...]`
  - `[1 ;1 ;1 ...]`
  - `[3]`, `[3 ;x]`, `[3 ;x ;y ...]`, `[3 ;x ;x]` ou `[3 ;x ;x ;y ...]`
  - `[x]` ou `[x ;y ...]` avec `x` supérieur ou égal à `4` ;
- le dernier entier de g_n est `1` ou `3`, et r_n est de la forme :
  - `[2 ;2 ;1 ;x]` ou `[2 ;2 ;1 ;x ;y ...]`,
  - `[2 ;2 ;1 ;1 ;1 ...]`
  - `[2 ;2 ;3]`, `[2 ;2 ;3 ;x]`, `[2 ;2 ;3 ;x ;x]`, `[2 ;2 ;3 ;x ;y ...]`, `[2 ;2 ;3 ;x ;x ;y ...]`,
  - `[2 ;2]`, `[2 ;2 ;x]` ou `[2 ;2 ;x ;y ...]` avec `x` supérieur ou égal à `4`.

Question 4. Écrivez une fonction `scinde : int list -> int list list` qui découpe un terme de la suite en une liste d'atomes.

> *Remarque :* je vous conseille de commencer par faire une version simplifié qui ne prend en compte que quelques cas.

Question 5. Vérifiez que l'on peut calculer l'évolution d'un terme indépendemment sur ces atomes.

Expliquez ce que vous faites.

Question 6. *(Bonus)* Calculez la liste des atomes (sans doublons) apparaissant dans les termes de la suite à partir d'un état initial aléatoire.

Que constatez-vous ?

Si la suite de Conway vous intéresse, je vous conseille la lecture de :
- Conway, J. H. "The Weird and Wonderful Chemistry of Audioactive Decay." dans *Open Problems in Communications and Computation*.