info401 : Programmation fonctionnelle
TP 1 : programmes récursifs, listes

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.)
- 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 : Quelques compléments sur les programmes récursifs *(45 min)*

Le programme

```
let rec fib n =
  if n<2
  then n
  else fib (n-1) + fib (n-2)
```

permet de calculer la suite de Fibonacci mais n'est pas très efficace. Pour essayer de comprendre ce qu'il se passe, on peut demander à Caml de *tracer* la fonction `fib` : après avoir défini la fonction, tapez "`#trace fib ;;`" dans l'interprète Caml. (N'oubliez pas le "`#`".)

Ceci permet d'afficher tous les appels intermédiaires à la fonction `fib`. Caml affiche :
- "`fib <--` n" pour dire que la fonction `fib` reçoit l'argument n,
- "`fib -->` n" pour dire que la fonction `fib` renvoie le résultat n

Pour arrêter de tracer la fonction `fib`, il suffit d'utiliser la commande "`#untrace fib ;;`".

Question 1. Tracez la fonction `fib` pour le calcul de `fib 5`. Qu'en pensez-vous ?

Remarque : il n'est pas nécessaire de mettre le code correspondant dans le fichier que vous m'envoyez : mettez seulement vos remarques en commentaire.

Question 2. Écrivez une fonction `fib2` de type "`int -> int*int`" qui pour l'argument n renvoie deux valeurs :
- la valeur de la fonction de Fibonacci pour n,
- la valeur de la fonction de Fibonacci pour *n-1*.

Bien entendu, vous ne devez pas utiliser la fonction `fib` pour définir `fib2`.

Question 3. En traçant la fonction `fib2`, comparer les calculs fait lors du calcul de `fib 7` et celui de `fib2 7`.

Comment pouvez-vous vous servir de cette fonction pour calculer la fonction de Fibonacci de manière plus efficace ?

Question 4. (Bonus) Écrivez une fonction `fib3` de type "`int -> int -> int -> int`" qui fait la chose suivante :

- son premier argument est l'entier n pour lequel on veut calculer la fonction de Fibonacci.
- ses deux autres arguments sont des *accumulateurs* qui permettent de conserver des valeurs successives (pour *i-1* et *i*) de la fonction de Fibonacci. Lorsque n diminue, ces deux arguments augmenteront...

> *Remarque :* cela veut dire que pour calculer la fonction de Fibonacci sur l'entier n, il faudra appeler "`fib3 0 1` n".

Tracez cette fonction lors du calcul de "`fib3 15`". Que constatez-vous ?

À votre avis, quelle est la meilleur méthode pour calculer la fonction de Fibonacci : en utilisant `fib2` ou en utilisant `fib3` ?

Partie 2 : Manipulation des listes *(30 min)*

Question 1. (facultatif) Reprenez quelques fonctions du TD sur les listes et programmez les en utilisant

- une définition directe,
- la fonction `List.fold_right`.

Question 2. Programmez la fonction `renverse` comme nous l'avons fait en TD, puis comparer l'efficacité de votre fonction avec celle de la fonction `List.rev`, qui fait la même chose.

Expliquez la démarche que vous suivez pour faire cette comparaison en mettant des commentaires dans votre fichier Caml.

Partie 3 : Deux algorithmes de tri pour les listes *(1h15min)*

Le *tri par insertion* et le *tri fusion* sont deux algorithmes de tri naturellement récursifs.

Le tri par insertion fonctionne de la manière suivante :

- la liste vide est triée,
- pour trier la liste "`a::l`", on commence par trier (récursivement) la liste `l`, puis on *insère* l'élément `a` dans le résultat.

Le tri fusion fonctionne de la manière suivante :

- la liste vide est triée,
- pour trier une liste non-vide, on sépare la liste en deux parties de longueur égale (à un élément prêt), on trie (récursivement) ces deux listes, puis on *fusionne* les deux résultats.

Question 1. Programmez une fonction `insertion` qui permet d'insérer un élément à sa place dans une liste triée.

Question 2. Programmez ensuite le tri par insertion : `tri_insertion`.

Question 3. Programmez une fonction `fusionne` qui permet de fusionner deux listes triées en une seule liste triée.

Question 4. Programmez une fonction `separe` qui permet de séparer une liste quelconque en deux listes de taille équivalente.

Question 5. Programmez maintenant le tri fusion : `tri_fusion`.

Question 6. Comparez l'efficacité de vos deux algorithmes. Qu'en pensez-vous ?

Expliquez la démarche que vous suivez pour faire cette comparaison en mettant des commentaires dans votre fichier Caml.

Question 7. Imaginez la situation suivante : vous avez une liste de points dans le plan, c'est à dire une liste de paires de flottants (type "`float*float list`").

Vous aimeriez trier cette liste selon la hauteur de points, c'est à dire suivant leur seconde coordonnée.

Comment pourrait-on profiter des capacités de Caml à manipuler des fonctions pour réécrire les algorithmes de tri de manière à les faire fonctionner dans ce cadre là.

Qu'en pensez-vous ?

Question 8. *(Bonus)* Un algorithme de tri est dit *stable* quand il ne modifie pas l'ordre relatifs des éléments "égaux". (Par exemple, lorsque vous triez vos fichiers par taille croissante, vous aimeriez que pour les fichiers de même taille, l'ordre soit toujours l'ordre alphabétique.)

Est-ce que vos programmes de tri par insertion et de tri fusion sont stables ?

Sinon, pouvez-vous les rendre stables ?

Partie 4 : Quelques fractales *(1h15)*

La courbe du dragon du TP0 peut facilement se généraliser de la manière suivante : on se donne une fonction qui transforme un segment de droite en une liste de segments de droites, et on itère cette transformation en partant d'un segment initial.

Un point du plan est facilement représenté par une paire de flottants : par exemple, l'origine est simplement `(0. , 0.)`.

Un segment est donc simplement une paire de points, c'est à dire une paire de paires de flottants : le vecteur unité horizontal est donc simplement `( (0.,0.) , (1.,0.) )`.

Vous pouvez déclarer des types correspondants avec :

```
type point = float * float
type segment = point * point
```

Question 1. Programmez une fonction `trans : point -> segment -> segment` qui translate un segment (second argument) pour mettre son origine sur un point donné (premier argument).

Question 2. Programmez une fonction `zoom : float -> segment -> segment` qui multiplie la longueur d'un segment (second argument) par un facteur flottant (premier argument). Le résultat devra avoir la même origine que le segment de départ.

Question 3. Programmez une fonction `rot : float -> segment -> segment` qui fait une rotation autour de l'origine d'un segment (second argument). L'angle de la rotation est donné par le premier argument de la fonction.

> *Remarque :* les fonctions trigonométriques sont disponibles dans la bibliothèque standard avec les noms usuels : `sin` et `cos`. Par contre, ces fonctions utilisent des angles en radians.
>
> La rotation d'angle α autour de l'origine transforme le point de coordonnées (x, y) en $\big(x\cos(\alpha) - y\sin(\alpha), x\sin(\alpha) + y\cos(\alpha)\big)$.

Question 4. Programmez une fonction `applique : (segment -> segment list) -> segment list -> segment list` qui permet d'appliquer une transformation de segment sur tous les segments d'une liste, et de récupérer tous les segments résultats.

Programmez ensuite une fonction `applique_iter : int -> segment -> (segment -> segment list) -> segment list` qui permet d'itérer la fonction précédente en partant d'un unique segment initial.

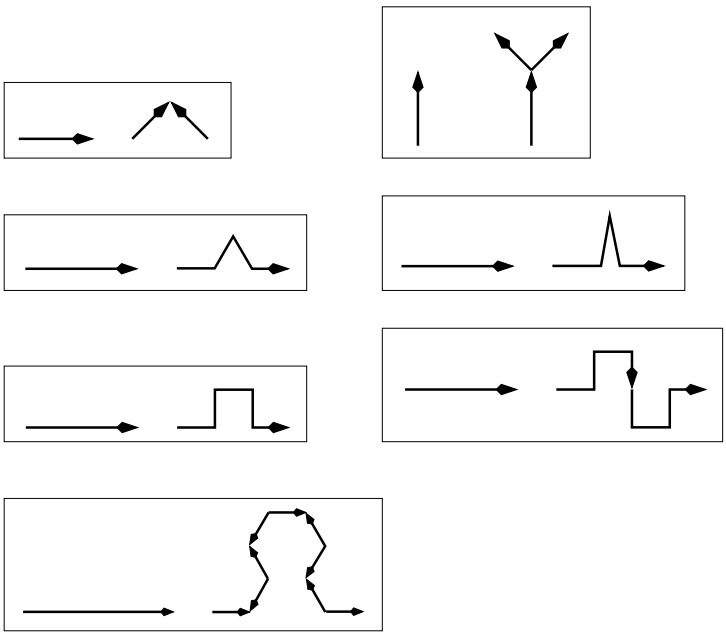
Question 5. Programmez une fonction `affiche : segment list -> unit` pour afficher tous les segments d'une liste. (Reportez-vous au TP précédent pour les fonctions d'affichage.)

Question 6. Programmez la fonction finale `dessine_fractale : int -> segment -> (segment -> segment list) -> unit`. Cette fonction prend comme arguments :

- un entier n représentant le nombre d'itérations,
- un segment initial,
- une transformation de segments,

et elle dessine le résultat.

Question 7. Essayez votre fonction sur différentes transformations. Vous pouvez utiliser par exemple (le dessin de droite montre comment on transforme le segment de gauche) :



Remarque : pour simplifier l'écriture des transformation de segments, vous pouvez utilisez des fonctions supplémentaires :

- `inv` qui renverse un segment,
- `avance` qui avance un segment pour que son origine se trouve sur l'arrivée,
- `(>>=)` qui permet de composer de composer des fonctions.

Par exemple, voici le code de la transformation du dragon telle que je l'ai écrite :

```
let fdragon (u:segment) : segment list =
  let a = 1. /. (sqrt 2.) in  (* facteur de réduction *)
  (* on racourcit, puis on tourne de 45 degré *)
  let v1 = ( rot 45. >>= zoom a ) u in
  (* on avance le segment, on tourne et on renverse *)
  let v2 =  ( inv >>= rot (-.90.) >>= avance ) v1 in
    [ v1 ; v2 ]
```

La fonction `(>>=)` est définie de la manière suivante :

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

Question 8. (Bonus) Pour améliorer votre fonction `dessine_fractale`, vous pouvez, avant d'afficher, calculer la taille du morceau de plan dont vous avez besoin, et appliquer une transformation pour faire une utilisation optimale de votre fenêtre graphique...