▶ **HX4 & HX1 – Option Informatique**
Second TP Caml
Vendredi 21 janvier 2011

Listes

1 Utilisation du filtrage

1.1 filtrage simple

▶ **Question 1** *Programmez les fonctions :*

```
tete : 'a list −> 'a
queue : 'a list −> 'a list
```

qui doivent renvoyer l'élément de tête et la queue de la liste ou échouer avec une exception (utiliser le mot clé failwith*).*

1.2 filtrage et récursivité

Programmez les fonctions :

▶ **Question 2**

```
taille : 'a list −> int
```

renvoyant la taille d'une liste (c'est à dire le nombre d'éléments de la liste).

▶ **Question 3**

```
appartient : 'a −> 'a list −> bool
```

renvoyant true *si x apparait dans la liste*

▶ **Question 4**

```
applique : ('a −> 'b) −> 'a list −> 'b list
```

qui étant données une fonction f et une liste $[a_1 ; ... ; a_n]$ doit renvoyer $[fa_1 ; ... ; fa_n]$.

▶ **Question 5**

```
pour_tout : ('a −> bool) −> 'a list −> bool
```

qui prend un prédicat (c'est à dire une fonction renvoyant un booléen) et doit renvoyer true si tous les éléments de la liste vérifient le prédicat ($f(e) =$ true pour tout élément e de la liste).

2 it_list

La fonction

```
it_list : ('a −> 'b −> 'a) −> 'a −> 'b list −> 'a
```

prend en argument une fonction f, un élément a_0 et une liste $[b_1; b_2; ...; b_n]$. Elle renvoie a_n où $a_i = f\ a_{i-1}\ b_i$.

▶ **Question 6** *Explicitez f et a_0 tels que $a_n = \sum_{i=1}^{n} b_i$.*

En déduire une fonction

```
somme : int list −> int
```

utilisant it_list, qui doit renvoyer la somme des éléments d'une liste.

▶ **Question 7** *De la même manière, programmez, d'abord, en utilisant it_list, puis, sans,*

```
retourne : 'a list −> 'a list
```

qui prend en argument $[a_1; ...; a_n]$ et qui renvoie $[a_n; ...; a_1]$.

▶ **Question 8** *Programmez votre version de la fonction it_list.*

3 Tris

3.1 tri par insertion

▶ **Question 9** *Écrivez une fonction*

```
insere : 'a −> 'a list −> 'a list
```

qui étant donnés un élément x et une liste triée $[a_1; ...; a_n]$ avec $a_1 \leq a_2 \leq ... \leq a_n$ doit renvoyer $[a_1; ...; a_i; x; a_{i+1}; ...; a_n]$ avec $a_i \leq x \leq a_{i+1}$. En déduire une fonction

```
tri : 'a list −> 'a list
```

qui doit renvoyer la liste triée. On pourra utiliser it_list ou faire une fonction à deux arguments : le premier est un bout de liste déjà trié, le second est ce qu'il reste à trier.

3.2 tri fusion

▶ **Question 10** *Écrivez les fonctions :*

```
partition : 'a list −> 'a list * 'a list
```

qui, d'une liste, fabrique une bi-partition la plus équilibrée possible ;

```
reunit : 'a list * 'a list −> 'a list
```

qui fabrique une liste triée à partir de deux listes triées. En déduire la fonction :

```
tri_fusion : 'a list −> 'a list
```

▶ **Question 11** *En utilisant*

```
random__int : int −> int
```

générez des listes aléatoires et vérifiez que vos tris trient bien.

▶ **Question 12 *** *Quel est le meilleur des deux algorithmes ? Essayez d'évaluer le nombre d'opérations que va effectuer chacun des algorithmes. Mettez le sous la forme $O(f(n))$, pour de bons f.*

3.3 Tri par sélection

▶ **Question 13** *Le tri par sélection fonctionne de la manière suivante : on extrait le minimum de la liste que l'on veut trier, on trie la liste restante, on ajoute le minimum au début du reste trié.*

4 Quelques exercices difficiles

▶ **Question 14 *** ***Type ENS : Back to the future***
Après avoir voyagé dans le futur puis être revenu à votre époque, vous disposez d'un tableau qui vous dit ce que vaut, chaque mois, le cours de l'or. Comme vous voulez à tout prix vous enrichir, mais en faisant le moins d'efforts possible, vous décidez de regarder quel est le profit maximal que vous puissiez faire. Le cours de l'or est présenté sous la forme d'une liste d'entiers et vous devez donner le gain maximal réalisable. Vous ne pouvez faire qu'un achat/vente car sinon vous modifieriez trop le cours du temps et donc cela changerait les cours (vous pourriez y perdre) mais cela risquerait aussi de modifier le continuum espace/temps lui même !

▶ **Question 15 *** ***Type ENS : Majorité***
Vous possédez les résultats d'une élection, présenté sous forme de liste. Par flemme, vous ne voulez faire qu'un dépouillement, c'est à dire qu'un parcours de liste. En plus, vous ne disposez que d'une petite mémoire. Vous ne voyez aucune technique pour trouver le candidat qui a le plus de voix ... Vous vous contentez donc d'assurer de donner le bon nom (celui qui a le plus de voix), seulement s'il obtient une stricte majorité. Comment faire ?

▶ **Question 16 *** ***Jeu : Snake***
Vous connaissez tous le jeu du snake, présent sur la plupart des téléphones portables. On va le programmer. Tout d'abord il faut récupérer l'interface graphique sur le site du TP. Ensuite vous devez écrire une fonction

```
enleve_queue : 'a list -> 'a list
```

qui retire le dernier élément d'une liste.
Ensuite vous devez compléter la fonction maj. L'interface donnée se charge du graphisme et de la gestion du clavier. Cette fonction maj doit faire évoluer le serpent, gérer si la pomme est avalée, si la tête du serpent croise le corps, si le serpent sort du périmètre ... toutes les règles que vous souhaitez.

▶ **Question 17 *** ***Équilibrage***
Étant donné quatre listes d'entiers $[a_1, ..., a_n]$, $[b_1, ..., b_m]$, $[c_1, ..., c_p]$, $[d_1, ..., d_q]$, pouvez vous dire s'il existe i, j, k, l tels que $a_i + b_j + c_k + d_l = 0$.

▶ **HX4 & HX1 – Option Informatique**
Second TP Caml
Vendredi 21 janvier 2011

Listes

Un corrigé

▶ Question 1

```
let tete = function
  | [] -> failwith "liste vide"
  | a::q -> a

let queue = function
  | [] -> failwith "liste vide"
  | a::q -> q
```

▶ Question 2

```
let rec taille = function
  | [] -> 0
  | a::q -> 1+taille q
```

▶ Question 3

```
let rec appartient x = function
  | [] -> false
  | a::q -> a=x || appartient x q
```

▶ Question 4

```
let rec applique f = function
  | [] -> []
  | a::q -> f a::applique f q
```

▶ Question 5

```
let rec pour_tout pred = function
  | [] -> true
  | a::q -> pred a && pour_tout q
```

▶ Question 6

```
let somme l =
  let add x y = x+y in
  it_list add 0 l
```

▶ Question 7

```
let retourne l =
  let f ac el = el::ac in
  it_list f [] l
```

▶ Question 8

```
let rec it_list f a0 = function
  | [] -> a0
  | b::q -> it_list f (f a0 b) q
```

▶ Question 9

```
let rec insere x = function
  | [] -> [x]
  | a::q ->
    if a < x
    then a:: insere x q
    else x::a::q

let rec complete_tri l = function
  | [] -> l
  | a::q -> complete_tri (insere a l) q
;;
let tri l = complete_tri [] l

let tri2 l = it_list (fun ac el -> insere el ac) [] l
```

▶ Question 10

```
let rec partition = function
  | [] -> ([],[])
  | a::q ->
    let (g,d) = partition q in
    (a::d,g)

let rec reunit = function
  | (a::q,x::t) ->
    if x > a
    then a::reunit (q,x::t)
    else x::reunit (a::q,t)
  | (a,b) -> a@b

let rec tri_fusion = function
  | [] -> []
  | [a] -> [a]
  | l ->
    let (p1,p2) = partition l in
    reunit (tri_fusion p1,tri_fusion p2)
```

▶ **Question 11**

```
let rec aleatoire = function
  | 0 -> []
  | n -> random__int 100::aleatoire (n-1)

let a = aleatoire 100
let b = tri_fusion a
let c = tri a
let d = c = b
```

▶ **Question 12** Pour le premier algorithme on fait n fois : insérer un élément dans une liste. Dans le pire des cas on a donc :

$$\sum_{i=1}^{n} i = O(n^2)$$

Pour le second, on a que, quand $2^k \leq n \leq 2^{k+1}$, on peut minorer et majorer par les temps mis pour 2^k et pour 2^{k+1}. Quand $n = 2^k$ on montre alors par récurrence que $T(k) = 2^k + 2 * T(k-1) + 2^k$ donc $\frac{T(k)}{2^k} = 1 + \frac{T(k-1)}{2^{k-1}}$.
On a donc $T(k) = k * 2^k$ et donc l'algorithme est en $O(n * ln(n))$. Par ailleurs, on ne peut pas avoir une meilleure complexité.
Bien évidemment, le premier algorithme est plus lent, en pire cas, que le second.

▶ **Question 13**

```
let rec extrait_min = function
  | [] -> failwith "liste vide"
  | [a] -> (a,[])
  | a::q ->
    let (mini,reste) = extrait_min q in
    if a < mini
    then (a,q)
    else (mini,reste)

let rec tri_insertion = function
  | [] -> []
  | l ->
    let (mini,reste) = extrait_min l in
    mini::(tri_insertion reste)
```

▶ **Question 14** On remarque que le gain maximal réalisé en un point est donné par la différence entre la valeur de ce point et du minimum sur la partie de la liste avant cet élément. Il suffit donc de maintenir le minimum rencontré et à chaque fois de voir combien on gagne en vendant.

```
let backtothefuture = function
  | [] -> failwith "pas d'infos"
  | a::q ->
    let rec foo min_vu = function
      | [] -> 0
      | a::q -> max (a-min_vu) (foo (min a min_vu) q)
    in
    foo a q
```

▶ **Question 15** Si on était vraiment cette personne on devrait, dès que l'on a deux bulletins pour des candidats différents, les déchirer et les bulletins qui nous reste en main à la fin ne peuvent être que ceux de celui qui a la majorité stricte, si majorité stricte il existe.

```
let majorite = function
  | [] -> failwith "impossible"
  | a::q ->
    let rec foo candidat vote = function
      | [] -> candidat
      | a::q ->
        let nvote = vote + ( if a = candidat then 1 else -1 ) in
        if nvote < 0
        then foo a 1 q
        else foo candidat nvote q
    in
    foo a 1 q
```

▶ **Question 16**

```
(* Un snake minimaliste *)
and regulier y =
  if y < 0 || y >= 30 then failwith "en dehors"

and maj serpent pomme grandir ax ay =

  let (tetex,tetey) = tete(serpent) in
  regulier (tetex+ax) ;
  regulier (tetey+ay) ;
  let ntete = (tetex+ax,tetey+ay) in
  let nserpent = ntete::(if grandir > 0 then serpent else enleve_queue serpent) in
  if appartient ntete serpent then failwith "superposition" ;
  let (npomme,ngrandir) = (if pomme = ntete then ((random__int 30,random__
    (nserpent,npomme,ngrandir)
```

▶ **Question 17** Pour avoir une bonne solution (en $O(n^2)$ et non en $O(n^2 ln(n))$) il faut un outil que nous verrons prochainement : les tables de hachages. Sans : On crée la liste des sommes de un élément de a et d'un de b d'un côté et celle de l'inverse des sommes d'un élément de c et un de d. On cherche alors une intersection entre ces deux listes. Pour trouver les intersections on trie et on fait un code similaire à reunit.

```
let somme l l' =
  let rec foo acc = function
    | (_,[]) -> acc
    | ([],_) -> acc
    | ([a],x::t) -> foo (a+x::acc) ([a],t)
    | (x::t,q) -> foo (foo acc ([x],q)) (t,q)
  in
  tri_fusion (foo [] (l,l'))

let equilibre a b c d =
  let rec inter = function
    | ([],_) -> false
    | (_,[]) -> false
    | (a::q,x::t) ->
      if a=x then true
      else
        if a < x
        then inter (q,x::t)
        else inter (a::q,t)
  in
  let rec nega = function
    | [] -> []
    | a::q -> -a::nega q
  in
  inter (somme a b , somme (nega c) (nega d))
```