UNIVERSITÉ D'ORLÉANS **DEUG MIAS**
IF 3.01 - Informatique **Année 2001-2002**

Feuille d'exercices n° 6

Exercice 1
Il existe plusieurs méthode pour énumérer toutes les paires issues d'une liste. En voici une :

```
# let rec paires l = let rec paires_el a m = match m with
                                  []   -> []
                                | b::r -> [(a,b);(b,a)]@(paires_el a r) in
                        match l with
                      []   -> []
                    | a::r -> (a,a)::(paires_el a r)@(paires r);;
val paires : 'a list -> ('a * 'a) list = <fun>
# paires [1;2;3];;
- : (int * int) list = [1, 1; 1, 2; 2, 1; 1, 3; 3, 1; 2, 2; 2, 3; 3, 2; 3, 3]
```

Exercice 2
Réutilisons la fonction *quicksort* définie lors d'un TD précédent :

```
# let rec separe inf a l = match l with
          []   -> ([],[])
        | b::m -> let (x,y) = separe inf a m in
                if (inf b a) then (b::x,y) else (x,b::y);;
val separe : ('a -> 'b -> bool) -> 'b -> 'a list -> 'a list * 'a list = <fun>
# let rec quicksort inf l = match l with
          []   -> []
        | b::m -> let (x,y) = separe inf b m in
                (quicksort inf x)@[b]@(quicksort inf y);;
val quicksort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

Pour l'utiliser sur des paires avec un ordre lexicographique, il suffit de définir cet ordre lexicographique à partir d'un ordre donné :

```
# let lexinf inf (a,b) (c,d) = (inf a c) || ((a=c) && (inf b d));;
val lexinf : ('a -> 'a -> bool) -> 'a * 'a -> 'a * 'a -> bool = <fun>
```

On peut maintenant écrire une nouvelle fonction `paires`, comme à l'exercice précédent, mais en imposant que la liste finale soit triée... Attention : ici, il faut prendre pour `inf` l'ordre strict, sinon l'ordre lexicographique n'est pas réellement un ordre.

```
# let inf a b = (a<b);;
val inf : 'a -> 'a -> bool = <fun>
# let paires_triees l = quicksort (lexinf inf) (paires l);;
val paires_triees : 'a list -> ('a * 'a) list = <fun>
# paires_triees [1;2;3;4];;
- : (int * int) list = [1, 1; 1, 2; 1, 3; 1, 4; 2, 1; 2, 2;
2, 3; 2, 4; 3, 1; 3, 2; 3, 3; 3, 4; 4, 1; 4, 2; 4, 3; 4, 4]
```

Exercice 3

Le type ABF est définit comme suit :

```
# type 'a abf = Nil
              | Node of ('a abf * 'a abf)
              | Leaf of 'a;;
type 'a abf = Nil | Node of ('a abf * 'a abf) | Leaf of 'a
```

L'arbre donné dans l'énoncé est alors décrit par :

```
# let arbre = Node (Node (Node (Nil,
                              (Leaf 'N')),
                         (Leaf 'O')),
                   (Node (Node ((Leaf 'I'),
                              Nil),
                         (Node ((Leaf 'U'),
                              Nil)))));;
val arbre : char abf =
  Node
   (Node (Node (Nil, Leaf 'N'), Leaf 'O'),
    Node (Node (Leaf 'I', Nil), Node (Leaf 'U', Nil)))
```

Attention : il faut bien parenthéser ce genre d'expressions pour 'eviter toute ambiguïté. Caml s'en plaindra, de toutes façons...
Nous pouvons maintenant écrire notre fonction de décodage :

```
# let rec decode_info arbre code = match code with
          [] -> (match arbre with
                   Leaf i -> i
                 | _      -> failwith "erreur")
        | a::r -> decode_info (match arbre with
                         Node (x,y) -> if a then x else y
                       | _          -> failwith "erreur" ) r;;
val decode_info : 'a abf -> bool list -> 'a = <fun>
# decode_info arbre [true; true; false];;
- : char = 'N'
# decode_info arbre [true; true];;
Uncaught exception: Failure "erreur".
```

```
# let rec aux_code_info arbre info = match arbre with
          Nil        -> ([], false)
        | Leaf i     -> if (i=info) then ([], true) else ([], false)
        | Node (x,y) -> let (c,v) = aux_code_info x info in
                                if v then (true::c,v) else
                        let (c,v) = aux_code_info y info in
                                if v then (false::c,v) else ([], v);;
val aux_code_info : 'a abf -> 'a -> bool list * bool = <fun>
# let code_info arbre info = let (c,v) = aux_code_info arbre info in
        if v then c else failwith "erreur";;
val code_info : 'a abf -> 'a -> bool list = <fun>
# code_info arbre 'N';;
- : bool list = [true; true; false]
```

Remarque : on fait ici un parcours en profondeur de l'arbre, et on retourne ainsi le bon chemin dès qu'on le trouve, sans nécessairement parcourir tout l'arbre.

Pour coder des mots, il suffit maintenant simplement de faire ces codages/décodages récursivement sur chaque lettre du mot. Un codage sera alors une liste de listes de booléens. Par ailleurs, pour rester dans un cadre assez général, un mot sera ici une liste de caractères. De cette façon, il suffira d'appliquer la fonction `map` sur la liste avec les fonctions de codage et de décodage :

```
# let rec map f l = match l with
          []   -> []
        | a::r -> (f a)::(map f r);;
# let decode_mot arbre code = map (decode_info arbre) code;;
val decode_mot : 'a abf -> bool list list -> 'a list = <fun>
# let code_mot arbre mot = map (code_info arbre) mot;;
val code_mot : 'a abf -> 'a list -> bool list list = <fun>
# code_mot arbre ['N';'O';'N'];;
- : bool list list =
[[true; true; false]; [true; false]; [true; true; false]]
# code_mot arbre ['O';'U';'I'];;
- : bool list list =
[[true; false]; [false; false; true]; [false; true; true]]
```

Exercice 4

```
# let rec ff = function   []   -> 1
                        | x::l -> 2*ff l;;
val ff : 'a list -> int = <fun>
```

Cette fonction prend en argument une liste de taille t et renvoie la valeur 2^t.

```
# ff [1;2;3];;
- : int = 8
# ff (paires [1;2;3]);;
- : int = 512
# ff (paires [1;2;3;4;5]);;
- : int = 33554432
```