TD de programmation fonctionnelle et logique

Corrigé du TD 6 : exceptions

Listes

1. Écrivez une fonction qui recherche si un élément appartient à une liste.

```
# let rec appartient x = function
    []   -> false
  | t::r -> (t = x) or (appartient x r);;
val appartient : 'a -> 'a list -> bool = <fun>
```

2. Optimisez cette fonction par l'usage d'exceptions.

```
# exception Trouvé;;
exception Trouvé

# let appartient_opt x l =
    let rec appartient_aux = function
       []  -> false
    | t::r -> if (t = x) then raise Trouvé
                         else (appartient_aux r)
    in try appartient_aux l with Trouvé -> true;;
val appartient_opt : 'a -> 'a list -> bool = <fun>
```

3. Écrivez une fonction booléenne récursive qui recherche si un élément appartient à une liste de listes.

```
# let rec appartient_liste x = function
    []   -> false
  | t::r -> (appartient x t) or (appartient_liste x r);;
val appartient_liste : 'a -> 'a list list -> bool = <fun>
```

4. Optimisez cette fonction par l'usage d'exceptions.

```
# let appartient_liste x l =
    let rec appartient_aux = function
        []   -> false
      | t::r -> if t=x then raise Trouvé
                       else appartient_aux r
    in
       let rec appartient_liste_aux = function
            []  -> false
         | t::r -> (appartient_aux t) or (appartient_liste_aux r)
       in try appartient_liste_aux l with Trouvé -> true;;
val appartient_liste : 'a -> 'a list list -> bool = <fun>
```

5. Écrivez, au moyen de fonctionnelles, une fonction booléenne qui recherche si un élément appartient à une liste de listes.

```
# let appartient_liste a l =
     List.fold_left
        (function x-> function y-> (appartient a y) or x)
```

```
        false
        l;;
val appartient_liste : 'a -> 'a list list -> bool = <fun>
```

6. Optimisez cette fonction par l'usage d'exceptions.

```
# let appartient_liste a l =
    let rec appartient_aux = function
        []   -> false
      | t::r -> if t=a then raise Trouvé
                       else appartient_aux r
    in try List.fold_left
             (function x-> function y-> (appartient a y) or x)
             false
             l
       with Trouvé -> true;;
val appartient_liste : 'a -> 'a list list -> bool = <fun>
```

7. Écrivez une fonction récursive qui recherche si un élément appartient à une liste de listes et qui renvoie la liste contenant l'élément.

```
# let rec appartient_liste x = function
    []   -> []
  | t::r -> if (appartient x t) then t else (appartient_liste x r);;
val appartient_liste : 'a -> 'a list list -> 'a list = <fun>
```

8. Optimisez cette fonction par l'usage d'exceptions (on supposera ici que l'on traite des listes de listes d'entiers).

```
# exception Liste of int list;;
exception Liste of int list
```

```
# let appartient_liste x l =
    let rec appartient_aux = function
        []   -> false
      | t::r -> if t=x then raise Trouvé
                       else appartient_aux r
    in
       let rec appartient_liste_aux = function
            []  -> []
         | t::r -> if (try (appartient_aux t) with Trouvé -> true)
                   then raise (Liste t)
                   else (appartient_liste_aux r)
       in try appartient_liste_aux l with Liste t -> t;;
val appartient_liste : int -> int list list -> int list =
<fun>
```

9. Écrivez, au moyen de fonctionnelles, une fonction qui recherche si un élément appartient à une liste de listes et qui renvoie la liste contenant l'élément.

```
# let rec appartient_liste a l =
     List.fold_left
        (function x -> function y -> if (appartient a y) then y else x)
        []
        l;;
val appartient_liste : 'a -> 'a list list -> 'a list = <fun>
```

10. Optimisez cette fonction par l'usage d'exceptions (on supposera ici que l'on traite des listes de listes d'entiers).

```
# let rec appartient_liste a l =
    let rec appartient_aux = function
```

```
        []   -> false
      | t::r -> if t=a then raise Trouvé
                       else appartient_aux r
   in try List.fold_left
               (function x -> function y ->
                 if try (appartient a y)
                    with Trouvé -> true
                 then raise (Liste y)
                 else x)
               []
               l
       with Liste z -> z;;
val appartient_liste : int -> int list list -> int li
st = <fun>
```

Arbres binaires

Un arbre binaire est soit une feuille (un entier), soit un nœud interne (composé d'un entier, la clef, et de deux sous-arbres). Nous définissons un arbre binaire par le type suivant :

```
# type arbre = Feuille of int | Noeud of (int * arbre  * arbre);;
type arbre = Feuille of int | Noeud of (int * arbre * arbre)
```

1. Écrivez une fonction `appartient` qui renvoie `true` si et seulement si l'entier argument est contenu dans l'arbre étudié.

   ```
   # let rec appartient n= function
           Feuille f   -> f = n
     | Noeud (c, fg, fd) -> c = n or
                              (appartient n fg) or
                              (appartient n fd);;
   val appartient : int -> arbre -> bool = <fun>
   ```

2. Écrivez une fonction `appartient_opt`, nouvelle version de `appartient` optimisée par l'usage d'exceptions.

   ```
   # exception Trouvé;;
   exception Trouvé
   ```

   ```
   # let appartient_opt n tree =
       let rec appartient_aux = function
          Feuille f   -> if f = n then raise Trouvé else false
        | Noeud (c, fg, fd) -> if c = n
                               then raise Trouvé
                               else (appartient_aux fg) or
                                    (appartient_aux fd)
       in try appartient_aux tree with Trouvé -> true;;
   val appartient_opt : int -> arbre -> bool = <fun>
   ```