TP de programmation fonctionnelle et logique
Corrigé du TP 3 : arbres n-aires et exceptions

Arbres n-aires

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

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

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
  | Noeud (c, [])
      -> c = n
  | Noeud (c, fils)
      -> c = n or
          (List.fold_right
             (function x -> function y-> (appartient n x) or y)
             fils
             false);;
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
       Noeud (c, [])   -> if c = n then raise Trouvé else false
     | Noeud (c, fils) ->
         if c = n
         then raise Trouvé
         else let l = (List.map appartient_aux fils)
              in false
    in try appartient_aux tree with Trouvé -> true;;
val appartient_opt : int -> arbre -> bool = <fun>
```

Exceptions

On se place de nouveau dans le cadre des arbres binaires définis par le type suivant :

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

– Écrivez une fonction `appartient` qui renvoie `true` si et seulement si l'entier argument est contenu dans l'arbre étudié ; et qui se termine dès que l'élément recherché a été trouvé.

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

```
# let appartient 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 : int -> arbre -> bool = <fun>
```

– Écrivez une fonction `appartient_prof` ayant le même comportement que la fonction `appartient`, mais renvoyant en plus la profondeur du nœud contenant l'élément recherché (on renverra 0 si l'élément recherché est absent de l'arbre).

```
# exception Trouvé_prof of int;;
exception Trouvé_prof of int
```

```
# let appartient_prof n tree=
   let rec appartient_prof_aux prof = function
   Feuille f
      -> if (f = n) then raise (Trouvé_prof prof) else false
   | Noeud (c, fg, fd)
      -> if c = n
         then  raise (Trouvé_prof prof)
         else (appartient_prof_aux (prof+1) fg) or
              (appartient_prof_aux (prof+1) fd)
   in
   try (appartient_prof_aux 1 tree,0) with Trouvé_prof n -> (true,n);;
val appartient_prof : int -> arbre -> bool * int = <fun>
```

– Écrivez une fonction `appartient_arbre` ayant le même comportement que la fonction `appartient`, mais renvoyant en plus le sous-arbre dont la racine est l'élément recherché (on renverra l'arbre de départ si l'élément recherché en est absent).

```
# exception Trouvé_arbre of arbre;;
exception Trouvé_arbre of arbre
```

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

– Écrivez une fonction `appartient_arbre_père` qui recherche un élément dans un arbre et qui renvoie le sous-arbre dont un des fils a l'élément recherché pour racine (si celui existe).

```
# exception Trouvé_arbre;;
exception Trouvé_arbre
```

```
# exception Trouvé_père of arbre;;
exception Trouvé_père of arbre

# let appartient_arbre_père n tree=
   let rec appartient_arbre_père_aux = function
   Feuille f
      -> if (f = n) then raise Trouvé_arbre else false
   | (Noeud (c, fg, fd)) as t
      -> if c = n
         then  raise Trouvé_arbre
         else
            try
              (appartient_arbre_père_aux fg) or
              (appartient_arbre_père_aux fd)
            with Trouvé_arbre -> raise (Trouvé_père t)
   in
   try
      if (appartient_arbre_père_aux tree)
      then failwith "Cas impossible"
      else failwith "Élément absent de l'arbre"
   with    Trouvé_père t -> t
         | Trouvé_arbre  -> failwith "Élément à la racine";;
val appartient_arbre_père : int -> arbre ->
arbre = <fun>
```