Feuille de révisions n°3 - Programmation en OCaml : manipulation d'arbres

Dans toute cette feuille de révision on suppose définis les types OCaml suivants, permettant respectivement la représentation d'arbres binaires non vides, d'arbres généraux non vides.

```
type 'a btree =
  | E                              (* arbre vide *)
  | N of 'a * 'a btree * 'a btree (* noeud *)

type 'a gtree =
  | GN of 'a * 'a gtree list
```

1 Pied à l'étrier

R. 3-1 Définir une fonction permettant de tester si un arbre binaire est vide.

Solution

```
let est_vide (b: 'a btree): bool =
  match b with
  | N(_, _, _) -> false
  | E          -> true
```

R. 3-2 Définir une fonction calculant la hauteur d'un arbre binaire. On rappelle que la hauteur de l'arbre vide est -1.

Solution

```
let rec hauteur (b: 'a btree): int =
  match b with
  | N(x, g, d) -> 1 + (max (hauteur g) (hauteur d))
  | E          -> -1
```

R. 3-3 Définir une fonction calculant la taille (*i.e.* le nombre de nœuds) d'un arbre général. On itérera sur l'arbre général au moyen de deux fonctions mutuellement récursives.

Solution

```
let rec taille_ag (a: 'a gtree): int =
  (* calcule la taille de l'arbre a *)
  match a with
  | GN(x, l) -> 1 + (taille_ag_list l)
and taille_ag_list (l: 'a gtree list): int =
  (* calcule la somme des tailles des arbres dans l *)
  match l with
  | []     -> 0
```

```
    | x :: xl -> (taille_ag x) + (taille_ag_list xl)
```

R. 3-4 Définir une fonction calculant la taille *i.e.* (le nombre de nœuds) d'un arbre général. On itérera dans l'arbre général au moyen d'une unique fonction récursive et d'un `List.fold_left`.

Solution

```
let rec taille_ag_bis (a: 'a gtree): int =
  let GN(_, l) = a in
  List.fold_left (fun accu elem -> accu + (taille_ag_bis elem)) 1 l
```

R. 3-5 Définir une fonction calculant la taille (*i.e.* le nombre de nœuds) d'un arbre général. On fournira une implémentation récursive terminale.

Solution

```
let taille_ag (a: 'a gtree): int =
  let rec aux (todo : 'a gtree list) (res: int): int =
    match todo with
    | [] -> res
    | GN(x, gl) :: todo' -> aux (List.rev_append gl todo') (1 + res)
    (*On utilise rev_append qui est réc. term. contrairement à @.*)
```

R. 3-6 Définir une fonction permettant de tester l'appartenance d'une étiquette à un arbre binaire.

Solution

```
let rec mem (x: 'a) (b: 'a btree): bool =
  match b with
  | E          -> false
  | N(y, g, d) -> (x = y) || (mem x g) || (mem x d)
```

R. 3-7 Définir une fonction prenant en argument un arbre binaire étiqueté par des entiers et retournant `None` si l'arbre est vide et `Some(mi, ma)` où `mi` est l'étiquette minimale de l'arbre et `ma` l'étiquette maximale sinon. Cette fonction sera récursive et ne devra pas faire usage d'une fonction récursive auxiliaire.

Solution

```
let rec min_max (b: int btree): (int * int) option =
  match b with
  | E          -> None
  | N(y, g, d) ->
    begin
      match min_max g, min_max d with
      | None, None -> Some(y, y)
      | None, Some(mi, ma) -> Some (min mi y, max ma y)
      | Some(mi, ma), None -> Some (min mi y, max ma y)
      | Some(mi1, ma1), Some(mi2, ma2) ->
                      Some(min mi1 (min y mi2), max ma1 (max y ma2))
    end
```

R. 3-8 Définir une fonction prenant en argument un arbre binaire étiqueté par des entiers et retournant `None` si l'arbre est vide et `Some(mi, ma)` où `mi` est l'étiquette minimale de l'arbre et `ma` l'étiquette maximale sinon. Cette fonction ne sera pas récursive, elle devra faire appel à une fonction récursive auxiliaire qui sera récursive terminale.

Solution

```
let min_max_bis (b: 'a btree): ('a * 'a) option =
  let rec aux (todos: 'a btree list) (mi: 'a) (ma: 'a): 'a * 'a =
    match todos with
    | []                  -> (mi, ma)
    | N(y, g, d)::todos2 -> aux (g::d::todos2) (min y mi) (max y ma)
    | E :: todos2         -> aux todos2 mi ma
  in
  match b with
  | E -> None
  | N(y, g, d) -> let mi, ma = aux [g; d] y y in Some(mi, ma)
```

R. 3-9 Définir une fonction prenant en arguments deux arbres binaires et testant si ceux-ci ont la même forme, c'est-à-dire si les deux arbres sont égaux lorsqu'on omet les valeurs des étiquettes.

Solution

```
let rec meme_forme (b1: 'a btree) (b2: 'a btree): bool =
  match b1, b2 with
  | E, E                          -> true
  | N(_, g1, d1), N(_, g2, d2) -> (meme_forme g1 g2) && (meme_forme d1 d2)
  | _                             -> false
```

R. 3-10 Définir une fonction prenant en argument un arbre binaire étiqueté par des entiers et retournant `Some(p)` où `p` est la profondeur d'un nœud d'étiquette paire s'il en existe et `None` sinon. On s'efforcera d'interrompre la recherche dès que cela est possible, en utilisant un mécanisme d'exception.

Solution

```
let recherche_profondeur_etiquette_paire (b: int btree): int option =
  let exception Found of int in
  let rec aux (bb: int btree) (p: int): unit =
    match bb with
    | N(x, _, _) when x mod 2 = 0 -> raise (Found(p))
    | N(x, g, d)                  -> (aux g (p+1); aux d (p+1))
    | E                           -> ()
  in
  try (aux b 0; None) with
  | Found(p) -> Some(p)

```

2 Parcours d'arbres

R. 3-11 Définir une fonction calculant le parcours infixe d'un arbre binaire. On pourra fournir une implémentation de complexité quadratique.