Exercice 1 : Profondeur d'une étiquette

Dans cet exercice, on considère les arbres binaires, définis en OCaml au moyen du type ci-dessous.

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Q. 1 Définir une fonction `profondeur_etiquette : 'a btree -> 'a -> int option` prenant en paramètres un arbre binaire et une étiquette x et retournant la profondeur du nœud le plus profond contenant l'étiquette x. Si un tel nœud n'existe pas, on retournera `None`.

Solution

```
let rec profondeur_etiquette (a: 'a btree) (x: 'a): int option =
  match a with
  | V           -> None
  | N(y, g, d) ->
     match profondeur_etiquette g x, profondeur_etiquette d x with
      | Some(hg), Some(hd)            -> Some ((max hg hd) + 1)
      | None, Some(h) | Some(h), None -> Some h
      | None, None                    -> if y = x then Some 0 else None
```

On considère à présent les arbres généraux, définis ci-dessous.

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

Q. 2 Définir une fonction `profondeur_etiquette : 'a gtree -> 'a -> int option` prenant en paramètres un arbre général et une étiquette x et retournant la profondeur du nœud le plus profond contenant l'étiquette x. Si un tel nœud n'existe pas, on retournera `None`.

Solution

```
let rec profondeur_etiquette (a: 'a gtree) (x: 'a): int option =
  let GN(y, l) = a in
  match profondeur_etiquette_liste l x with
  | Some(h) -> Some(h)
  | _        -> if y = x then Some(0) else None
and profondeur_etiquette_liste (al: 'a gtree list) (x: 'a): int option =
  match al with
  | []         -> None
  | a :: al' ->
     match (profondeur_etiquette a x), (profondeur_etiquette_liste al' x) with
      | Some(hg), Some(hd) -> Some ((max (1 + hg) hd))
      | None, Some(h)      -> Some(h)
      | Some(h), None      -> Some (h+1)
      | None, None         -> None
```

Exercice 2 : Sous-arbres

Dans cet exercice, on considère les arbres binaires, définis en OCaml au moyen du type ci-dessous.

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Q. 1 Définir une fonction `egaux : 'a btree -> 'a btree -> bool` prenant en paramètres deux arbres et testant s'ils sont égaux.

Solution

```
let egaux (a: 'a btree) (b: 'a btree) = a = b
```

On dit d'un arbre a que c'est un sous-arbre d'un arbre b dès lors que l'arbre a apparaît dans l'arbre b. Formellement, on peut définir l'ensemble des sous-arbres inductivement de la manière suivante :

$$\begin{aligned} \mathsf{sous_arbres}(\mathsf{E}) &= \{\mathsf{E}\} \\ \mathsf{sous_arbres}(\mathsf{N}(x,g,d)) &= \{\mathsf{N}(x,g,d)\} \cup \{\mathsf{sous_arbres}(g)\} \cup \{\mathsf{sous_arbres}(d)\}. \end{aligned}$$

Q. 2 Définir une fonction OCaml `sous_arbres : 'a btree -> 'a btree -> bool` permettant de tester si un arbre a est un sous-arbre d'un arbre b.

Solution

```
let rec sous_arbres (a: 'a btree) (b: 'a btree) =
  (egaux a b) ||
  (match b with
   | V           -> false
   | N(x, g, d) -> (sous_arbres a g) || (sous_arbres a d))
```

On considère à présent les arbres généraux, définis ci-dessous.

```
type 'a gtree = N of 'a * 'a gtree list
```

Q. 3 Définir une fonction OCaml `sous_arbres : 'a gtree -> 'a gtree -> bool` permettant de tester si un arbre général a est un sous-arbre d'un arbre général b. *Indication :* On rappelle l'existence de la fonctionnelle `List.exists : ('a -> bool) -> 'a list -> bool` prenant en paramètres un prédicat p et une liste l et testant s'il existe, dans l, un élément x tel que $p(x)$ est vrai.

Solution

```
let rec sous_arbres (a: 'a gtree) (b: 'a gtree) =
  (a = b) ||
  (let N(_, l) = b in
   List.exists (fun x -> sous_arbres a x) l)
```

Exercice 3 : Meilleure branche

Dans cet exercice, on considère les arbres binaires, définis en OCAML au moyen du type ci-dessous.

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

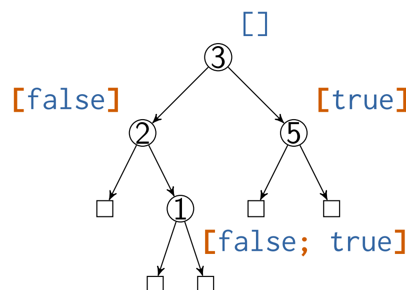
Une *branche* dans un arbre binaire est un chemin entre la racine et une feuille de l'arbre. On définit la *valeur* d'une branche $(n_1, n_2, \ldots, n_p)$ comme la somme des étiquettes des nœuds des branches.

Q. 1 Définir une fonction OCAML `val_pg_branche: int btree -> int` calculant la valeur de la plus grande branche d'un arbre binaire étiqueté par des entiers.

Solution

```
let rec val_pg_branche (b: int btree): int =
  match b with
  | V            -> 0
  | N(x, g, d) -> x + max (val_pg_branche g) (val_pg_branche d)
```

On représente une branche, dans un arbre binaire, en OCAML, au moyen d'une liste de booléens, comme indiqué dans la figure ci-dessous.



Q. 2 Définir une fonction OCAML `pg_branche: int btree -> (int * (bool list))` calculant à la fois la valeur d'une plus grande branche et une plus grande branche d'un arbre binaire étiqueté par des entiers.

Solution

```
let rec pg_branche (b: int btree): int * (bool list) =
  match b with
  | V -> (0, [])
  | N(x, g, d) ->
    let vg, bg = pg_branche g in
    let vd, bd = pg_branche d in
    if vg > vd then (x + vg, false :: bg)
    else            (x + vd, true  :: bd)
```

On considère à présent les arbres généraux, définis ci-dessous.

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

On représente une branche, dans un arbre binaire, en OCAML, au moyen d'une liste d'entiers : le premier fils est indiqué par 0, le second par 1, le troisième par 2, ...

Q. 3 Définir une fonction OCAML `pg_branche_g: int gtree -> (int * (intr list))` calculant à la fois la valeur d'une plus grande branche et une plus grande branche d'un arbre général étiqueté par des entiers.

Solution

```
let rec pg_branche_g (g: int gtree): int * (int list) =
  match g with
  | GN(x, []) -> (x, [])
  | GN(x, gs) ->
    let (v, b), idx = pg_branche_g_aux gs in
    (v + x, idx :: b)

(* Parmi la liste d'arbres généraux gs, retourne la valeur de la branche de
   celui ayant la plus grande branche, et la branche, ainsi que l'indice,
   dans la liste, de l'arbre dont elle provient. *)
and pg_branche_g_aux (gs: int gtree list): (int * (int list)) * int =
  match gs with
  | []        -> failwith "[pg_branche_g_aux] liste vide"
  | g :: []  -> (pg_branche_g g, 0)
  | g :: gs' ->
    let v, b           = pg_branche_g g in
    let (v', b'), idx = pg_branche_g_aux gs' in
    if v' > v then (v', b'), idx + 1
    else (v, b), 0
```