

OCaml **type** 'a arbre = Nil | Noeud **of** 'a arbre * 'a * 'a arbre

Question VI.1 Écrire des fonctions :

OCaml

```
(* Calcule le nombre de noeuds de a *)
let taille (a : 'a arbre) : int
(* Calcule la hauteur de a *)
let hauteur (a : 'a arbre) : int
(* Indique si a est réduit à une feuille *)
let feuille (a : 'a arbre) : bool
```

Démonstration.

OCaml

```
let rec taille a = match a with
| Nil -> 0
| Noeud(g,x,d) -> 1 + taille g + taille d

let rec hauteur a = match a with
| Nil -> -1
| Noeud(g,x,d) ->
    1 + max (hauteur g) (hauteur d)

let feuille a = match a with
| Nil -> false
| Noeud(g, x, d) -> g = Nil && d = Nil
```

Question VI.2 Écrire une fonction sous_arbres qui renvoie la liste des sous-arbres non vides d'un arbre.

En déduire des fonctions noeuds, feuilles et noeuds_internes qui renvoient la liste des étiquettes des noeuds correspondants.

Démonstration.

OCaml

```
let rec sous_arbres a = match a with
| Nil -> []
| Noeud(g, x, d) ->
    a :: (sous_arbres g @ sous_arbres d)

let rec noeuds a = match a with
| Nil -> []
| Noeud(g, x, d) ->
    x :: (noeuds g @ noeuds d)

let rec feuilles a = match a with
| Nil -> []
| Noeud(g, x, d) ->
    if feuille a then [x]
```

```

        else feuilles g @ feuilles d

let rec noeuds_internes a = match a with
| Nil -> []
| Noeud(Nil, _, Nil) -> []
| Noeud(g, x, d) ->
    x :: (noeuds_internes g @ noeuds_internes d)

```

Pour accéder à un nœud de l'arbre, on descend en partant de la racine et en allant à gauche ou à droite. On peut donc représenter un tel chemin par une liste de déplacements :

```

OCaml type déplacement = Gauche | Droite
      type chemin = déplacement list

```

Question VI.3 Écrire une fonction `chemin_noeud : 'a arbre -> chemin -> 'a option` qui renvoie l'étiquette d'un nœud donné par son chemin. On utilise un type option en cas de chemin invalide.

Démonstration.

```

OCaml type déplacement = Gauche | Droite
      type chemin = déplacement list

let rec chemin_noeud a l =
  match a, l with
  | Nil, _ -> None
  | Noeud(g,x,d), Gauche::q -> chemin_noeud g q
  | Noeud(g,x,d), Droite::q -> chemin_noeud d q
  | Noeud(g,x,d), [] -> Some x

```

VI.2 Parcours

Sur le même modèle que l'exploration par *backtracking*, on va réaliser un parcours en profondeur d'un arbre en explorant à gauche puis à droite ses sous-arbres de manière récursives. Chaque nœud est donc vu trois fois :

- une première fois quand on le découvre avant d'explorer son sous-arbre gauche
- entre les deux explorations
- enfin quand on a fini d'explorer son sous-arbre droit.

Question VI.4 Écrire des fonctions `affiche_avant`, `affiche_milieu` et `affiche_apres` qui parcourent et affiche les étiquettes d'un `string` arbre selon les trois cas précédents. *Note : il s'agit essentiellement du même code à une ligne près.*

Démonstration.

OCaml

```

let rec affiche_avant a = match a with
| Nil -> ()
| Noeud(g,x,d) ->
    print_string x;
    affiche_avant g;
    affiche_avant d

let rec affiche_milieu a = match a with
| Nil -> ()
| Noeud(g,x,d) ->
    affiche_milieu g;
    print_string x;
    affiche_milieu d

let rec affiche_apres a = match a with
| Nil -> ()
| Noeud(g,x,d) ->
    affiche_apres g;
    affiche_apres d;
    print_string x

```

■

Question VI.5 On peut représenter une expression arithmétique comme $(2 + 3) \times 4$ en tant qu'arbre avec des opérations pour les nœuds internes et des valeurs pour les feuilles. Quitte à utiliser des string dans les deux cas, qu'obtient-on sur une telle expression avec les trois opérations précédentes?

■

Démonstration.

OCaml

```

let rec evaluate a = match a with
| Nil -> failwith "Vide"
| Noeud(Nil,x,Nil) -> int_of_string x
| Noeud(g, x, d) ->
    let eg = evaluate g in
    let ed = evaluate d in
    match x with
    | "+" -> eg + ed
    | "*" -> eg * ed
    | _ -> failwith "Ope inconnue"

```

L'appel à `affiche_avant` sur l'expression $(2 + 3) \times 4$ va afficher `*+234`.

L'appel à `affiche_milieu` sur l'expression $(2 + 3) \times 4$ va afficher `2+3*4`.

L'appel à `affiche_apres` sur l'expression $(2 + 3) \times 4$ va afficher `23+4*`.

On retrouve ainsi les notions préfixes, infixes et postfixes d'une expression arithmétique.

■

On redonne ici l'implémentation d'une pile et d'une file en OCaml :

```

OCaml
let pile_creer () = ref []
let pile_empile p x = p := x :: !p
let pile_depile p =
  match !p with
  | [] -> failwith "Pile vide"
  | x::q -> p := q; x
let pile_est_vide p = !p = []

let file_creer () = (pile_creer (), pile_creer ())
let file_bascule (pin, pout) = pout := List.rev !pin; pin := []
let file_enfile (pin, pout) x = pile_empile pin x
let file_defile (pin, pout) =
  if pile_est_vide pout
  then file_bascule (pin, pout);
  pile_depile pout
let file_est_vide (pin, pout) = pile_est_vide pin && pile_est_vide pout

```

Ces deux structures de données ont la même interface mais deux comportements différents. On va utiliser un unique type pour les représenter afin de permettre à une fonction d'utiliser soit une pile soit une file.

```

OCaml
type ('a, 'b) taches = {
  creation : unit -> 'a;
  ajouter : 'a -> 'b -> unit;
  retirer : 'a -> 'b;
  est_vide : 'a -> bool
}

let taches_pile = {
  creation = pile_creer;
  ajouter = pile_empile;
  retirer = pile_depile;
  est_vide = pile_est_vide
}

let taches_file = {
  creation = file_creer;
  ajouter = file_enfile;
  retirer = file_defile;
  est_vide = file_est_vide
}

```

■ **Remarque 10.7** Ici, on a une limitation du système de types. On aimerait paramétrer `taches` par `('a 'b, 'a)` où `'b` est un type paramétré, ainsi on aurait soit `('a pile, 'a)` où `('a file, 'a)` comme pour les deux cas. Cependant, ce n'est pas possible, les types génériques ne sont pas paramétriques. C'est pour cela qu'on a un `('a, 'b)` sans avoir de lien apparent entre `'a` et `'b`.

On pourra alors écrire une fonction prenant un gestionnaire de tâches en paramètre :

```

OCaml | let f t a =
        let a_traiter = t.creation () in
        for i = 0 to Array.length a - 1 do
            t.ajouter a_traiter a.(i)
        done;
        while not (t.est_vide a_traiter) do
            print_int (t.retirer a_traiter)
        done

```

Ainsi `f taches_pile [|1;2;3|]` va afficher 321 et `f taches_file [|1;2;3|]` va afficher 123.

Question VI.6 En déduire une fonction

```

OCaml | parcours : ('a, 'b arbre) taches -> 'b arbre -> unit

```

qui effectue un parcours de l'arbre en affichant les étiquettes des nœuds visités dans l'ordre induit par le gestionnaire de tâches passé en paramètre.

Démonstration.

```

OCaml | let parcours t a =
        let avisiter = t.creation () in
        t.ajouter avisiter a;
        while not (t.est_vide avisiter) do
            match t.retirer avisiter with
            | Nil -> ()
            | Noeud(g,x,d) ->
                print_int x;
                t.ajouter avisiter g;
                t.ajouter visiter d
        done

```

VI.3 Arbres binaires stricts

Pour représenter des arbres dont les nœuds ont tous deux fils non vides, on va utiliser le type

```

OCaml | type ('a, 'b) arbre_bin = Feuille of 'a
        | NoeudI of ('a, 'b) arbre_bin * 'b * ('a, 'b) arbre_bin

```

Ce type permet de représenter naturellement des expressions arithmétiques comme :

```

OCaml | NoeudI(NoeudI( Feuille 2, '+', Feuille 3), '*', Feuille 5)

```

Question VI.7 Écrire une fonction `evalue : (int, char) arbre_bin -> int` qui évalue une telle expression.

Démonstration.

```

OCaml | let rec evaluer a = match a with
      | Feuille n -> n
      | NoeudI(g, x, d) ->
          let eg = evaluer g in
          let ed = evaluer d in
          match x with
          | '+' -> eg + ed
          | '*' -> eg * ed
          | _ -> failwith "Opérateur inconnu"

```

Si on considère le type

```

OCaml | type ('a, 'b) etiquette = F of 'a | N of 'b

```

On peut passer d'un ('a, 'b) arbre_bin à un ('a, 'b) etiquette arbre et, dans certains cas, d'un ('a, 'b) etiquette arbre à un ('a, 'b) arbre_bin.

Question VI.8 Écrire ainsi deux fonctions de conversion :

```

OCaml | arbre_bin_vers_arbre : ('a, 'b) arbre_bin -> ('a, 'b) etiquette arbre
      | arbre_vers_arbre_bin : ('a, 'b) etiquette arbre -> ('a, 'b) arbre_bin option

```

Démonstration.

```

OCaml | let rec arbre_bin_vers_arbre a =
      match a with
      | Feuille n -> Noeud(nil, F n, nil)
      | NoeudI(g, x, d) -> Noeud(arbre_bin_vers_arbre g,
                                N x,
                                arbre_bin_vers_arbre d)

      let rec arbre_vers_arbre_bin a =
        match a with
        | Noeud(nil, F f, nil) -> Feuille f
        | Noeud(g, N x, d) -> begin
            match arbre_vers_arbre_bin g, arbre_vers_arbre_bin d with
            | Some g', Some d' -> NoeudI(g', x, d')
            | _ -> None
          end
        | _ -> None

```

VI.4 Dessin d'arbres

On va réaliser dans cette partie une fonction de dessin d'arbres avec `graphics`. L'idée est de placer la racine et de dessiner les sous-arbres gauche et droit en dessous. Pour cela, il va falloir connaître la taille de ces sous-arbres en pixels.

On pourra consulter la documentation de `graphics` ici : [Graphics](#)

Question VI.9 Écrire une fonction `pixels : int arbre -> int * int` qui renvoie un couple (`largeur`, `racine`) où `largeur` est la largeur en pixels d'un arbre dans son affichage et `racine` l'abscisse de la racine dans cet affichage.

Démonstration.

```
OCaml
let rnoeud = 5 (* Le rayon du noeud *)
let marge = 2 (* la marge autour du noeud *)

let rec pixels a = match a with
| Nil -> 0, 0
| Noeud(g, _, d) ->
    let lg, _ = pixels g in
    let ld, _ = pixels d in
    lg + ld + 2*(marge + rnoeud), lg + marge + rnoeud
```

Question VI.10 En déduire une fonction `dessine : int arbre -> int -> int -> unit` tel que `dessine a x y` dessine l'arbre `a` en plaçant la racine en (x, y) .

Démonstration.

```
OCaml
let rec dessine a x y =
  match a with
  | Nil -> ()
  | Noeud(g, e, d) ->
      (* On précalcule les tailles des sous-arbres *)
      let lg, rg = pixels g in
      let _, rd = pixels d in
      (* position de la racine gauche relativement à (x,y) *)
      let v_rg = x - lg - marge - rnoeud + rg in
      (* position de la racine droite relative (x, y) *)
      let v_rd = x + marge + rnoeud + rd in
      let dec_y = y - 2 * rnoeud - marge in

      moveto x y; lineto v_rg dec_y;
      moveto x y; lineto v_rd dec_y;
      fill_circle x y rnoeud;

      moveto (x-tw/2) (y-th/2);
      set_color white;
      draw_string e;
      set_color black;

      dessine g v_rg dec_y;
      dessine d v_rd dec_y
```

VI.5 Génération aléatoire d'arbres

Dans cette partie, ce qui nous intéresse est de générer aléatoirement des arbres pour en observer la structure. Les étiquettes ne sont pas pertinentes, on pourra donc au choix, soit considérer des `unit arbre`, soit redéfinir un type d'arbre non étiquetés.

On va réaliser ici deux modèles de génération aléatoire d'arbres. Le premier modèle consiste, pour générer un arbre de n nœuds, à choisir aléatoirement k dans $\llbracket 0, n \rrbracket$, à générer aléatoirement un arbre g à k nœuds et un arbre d à $n - k$ nœuds puis à renvoyer l'arbre `Noeud(g, (), d)`.

Question VI.11 Implémenter ce modèle avec une fonction `genere_arbre_1 : int -> unit arbre`.

Tester votre fonction, notamment avec l'affichage de la partie précédente.

Démonstration.

```
OCaml
let rec genere_arbre_1 n =
  if n = 0
  then Nil
  else if n = 1 then Noeud(Nil, (), Nil)
  else let k = Random.int (n-1) in
        Noeud(genere_arbre_1 k, (),
              genere_arbre_1 (n-1-k))
```

Pour le deuxième modèle, on va choisir un sous-arbre vide uniformément et le remplacer par un nœud.

Question VI.12 Écrire une fonction `chemins_vides : 'a arbre -> chemin list` qui étant donné un arbre renvoie la liste des chemins permettant d'aboutir à un sous-arbre vide.

Démonstration.

```
OCaml
let chemins_vide a =
  let rec aux a l =
    match a with
    | Nil -> [ List.rev l ]
    | Noeud(g, _, d) ->
        aux g (Gauche::l)
        @ aux d (Droite::l)
  in aux a []
```

Question VI.13 Écrire une fonction `remplit : 'a arbre -> chemin -> 'a arbre` qui remplace un sous-arbre vide donnée par son chemin par un nœud.

Démonstration.

OCaml

```

let rec rempli a c =
  match a, c with
  | Nil, [] -> Noeud(Nil, (), Nil)
  | Noeud(g,x,d), Gauche::q ->
      Noeud(rempli g q, x, d)
  | Noeud(g,x,d), Droite::q ->
      Noeud(g, x, rempli d q)
  | _ -> failwith "Error"

```

Question VI.14 En déduire une fonction `genere_arbres_2 : int -> unit arbre` qui prend un entier n et construit un arbre à n nœuds en partant de l'arbre vide et en réalisant n remplacement d'un arbre vide par un nœuds, le choix de l'arbre vide se faisant uniformément.

Note : on pourra utiliser `List.nth` et `Random.int` Voir par exemple `Random`

Démonstration.

OCaml

```

let rec genere_arbre_2 n =
  if n = 0 then Nil
  else
    let a = genere_arbre_2 (n-1) in
    let l = chemins_vide a in
    let k = Random.int (List.length l) in
    let ch = List.nth l k in
    rempli a ch

```