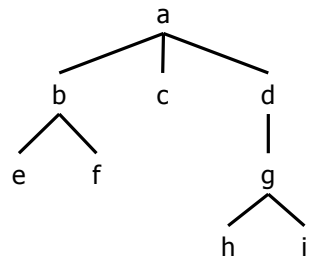
OCaml - Les arbres

L3 MI

Introduction

- Un arbre est un ensemble fini d'éléments liés par une relation dite "de parenté" telle que :
  - un élément et un seul ne possède pas de père (cet élément est appelé ***racine*** de l'arbre),
  - tout élément (à part la racine) possède exactement un père.



Exemple d'arbre

Quelques définitions (1)

- les éléments d'un arbre sont appelés ***noeuds***. Les éléments sans fils sont appelés ***noeuds terminaux*** ou ***feuilles***,
- la ***taille*** d'un arbre est le nombre de noeuds dans cet arbre,
- la ***profondeur*** d'un noeud est le nombre d'ascendants de ce noeud. La profondeur maximale des noeuds d'un arbre est la ***hauteur*** de cet arbre,
- le ***degré*** d'un noeud est son nombre de fils,

Quelques définitions (2)

- l'ensemble des descendants d'un noeud n forme un ***sous-arbre*** de racine n,
- un arbre est dit ***ordonné*** si l'ordre des branches d'un noeud n est significatif,
- un ***arbre n-aire*** est un arbre ordonné pour lequel tout noeud non-terminal possède exactement *n* branches,
- un arbre est ***étiqueté*** si ses noeuds portent une information.

Retour sur les listes

- Le type `list` prédéfini dans Caml :
  - constructeur `::`
  - arbre vide `[]`
- Notre propre type liste (utilise la récursivité de type de Caml) :

  `# type int_list = Empty_list | Elem of int * int_list;;`
- Ce qui signifie : "une liste d'entier est soit une liste vide soit la composition d'un entier (la tête de liste) et d'une liste d'entier".
- Récursivité de la définition de `int_list` : `int_list` apparaît dans sa propre définition.

Retour sur les listes (2)

- Exemple de fonctions utilisant le type `int_list` :

```
# let rec somme_int l = match l with
    Empty_list -> 0
  | Elem(tdl, reste) -> tdl + (somme_int reste);;
val somme_int : int_list -> int = <fun>
# let liste1 = Elem(1, Elem(2, Elem(3, Empty_list)));;
val liste1 : int_list = Elem (1, Elem (2, Elem (3, Empty_list)))
# somme_int liste1;;
- : int = 6
```


Arbres binaires (1)

- Dans un arbre binaire
  - Chaque noeud possède exactement deux sous-arbres (sous-arbre gauche et sous-arbre droit).
  - Pour les feuilles, les sous-arbres gauches et droits sont vides.
- Contrairement aux listes, pas de type arbre prédéfini en Caml.
- Exemple : un arbre binaire étiqueté par des entiers

```
# type int_btree =
    Empty
  | Node of int_btree * int * int_btree;;
type int_btree = Empty | Node of int_btree * int * int_btree
```

- Cette définition peut se lire de la manière suivante : un arbre binaire d'entiers est soit vide (`Empty`), soit formé d'un noeud racine (`Node`) auquel sont attachés un entier et deux sous-arbres d'entiers (le sous-arbre droit et le sous-arbre gauche).

Arbres binaires (2)

- `Empty` et `Node` sont des noms arbitraires, choisis par le programmeur. On les appelle constructeurs de types.
- Le constructeur `Empty` n'attend pas d'argument.
- Le constructeur `Node` attend - c'est la signification du mot-clé `of` - trois arguments respectivement de type `int_btree`, `int` et `int_btree`.

```
# let t1 = Empty;;
val t1 : int btree = Empty
# let t2 = Node (Node (Empty, 4, Empty), 3, Node (Empty, 5,
  Empty));;
val t2 : int btree = Node (Node (Empty, 4, Empty), 3, Node
  (Empty, 5, Empty))
# let t3 = Node (Node (Empty, 2, Empty), 1, t2 );;
val t3 : int btree =
  Node (Node (Empty, 2, Empty), 1,
    Node (Node (Empty, 4, Empty), 3, Node (Empty, 5, Empty)))
```

Arbres binaires (3)

- Les constructeurs servent aussi à accéder, via le mécanisme de filtrage, aux éléments de l'arbre.

```
# let root_label t = match t with
    Empty -> failwith "root_label : empty tree"
  | Node (l, x, r) -> x;;
val root_label : int_btree -> int = <fun>
# let left_subtree t = match t with
    Empty -> failwith "left_subtree : empty tree"
  | Node (l, x, r) -> l;;
val left_subtree : int_btree -> int_btree = <fun>
# let right_subtree t = match t with
    Empty -> failwith "right_subtree : empty tree"
  | Node (l, x, r) -> r;;
val right_subtree : int_btree -> int_btree = <fun>
# root_label t2;;
- : int = 3
# left_subtree t2;;
- : int_btree = Node (Empty, 4, Empty)
# right_subtree t3;;
- : int_btree = Node (Node (Empty, 4, Empty), 3, Node (Empty, 5,
  Empty))
```


Arbres polymorphes (1)

- `int_btree` définit des arbres étiquetés par des entiers : `int_btree` est **monomorphe**.
- Arbre binaire générique : paramètre de type dans la définition du type `btree`

```
# type 'a btree =
    Empty
  | Node of 'a btree * 'a * 'a btree;;
type 'a btree = Empty | Node of 'a btree * 'a * 'a
  btree
```

- Le type `int_btree` apparaît alors comme un cas particulier du type `'a btree` :

```
# let t2 = Node ( Node (Empty, 4, Empty), 3, Node
  (Empty, 5, Empty));;
val t2 : int btree = Node (Node (Empty, 4, Empty), 3,
  Node (Empty, 5, Empty))
```


Arbres polymorphes (2)

- On peut alors définir des arbres binaires étiquetés avec des valeurs de type quelconque :
  - chaîne de caractères :

```
# let t4 = Node (Node(Empty, "its", Empty), "hello",
  Node(Empty, "me", Empty));;
val t4 : string btree =
  Node (Node (Empty, "its", Empty), "hello", Node
  (Empty, "me", Empty))
```

  - listes :

```
# let t5 = Node (Node (Empty, [5; 6; 7], Empty), [3;
  4], Node (Empty, [8; 9], Empty));;
val t5 : int list btree =
  Node (Node (Empty, [5; 6; 7], Empty), [3; 4], Node
  (Empty, [8; 9], Empty))
```


Arbres polymorphes (3)

  - ou même fonctions :

```
# let t6 = Node (
    Node(Empty, (function x -> x * 2), Empty),
    (function x -> x ),
    Node(Empty, (function x -> x * 4), Empty));;
val t6 : (int -> int) btree =
  Node (Node (Empty, <fun>, Empty), <fun>, Node (Empty,
  <fun>, Empty))
```

[TD] La fonction *size*

- La fonction `btree_size` renvoie la taille d'un arbre binaire, c-à-d le nombre d'éléments qu'il contient :

```
# let rec btree_size t = match t with
    Empty -> 0
  | Node (l, _,r ) -> 1 + btree_size l + btree_size r;;
val btree_size : 'a btree -> int = <fun>
# btree_size t5;;
- : int = 3
```


[TD] La fonction *depth*

- La fonction `btree_depth` renvoie la profondeur d'un arbre binaire :

```
# let rec btree_depth t = match t with
    Empty -> 0
  | Node (l, _, r) -> 1 + max (btree_depth l)
  (btree_depth r);;
val btree_depth : 'a btree -> int = <fun>
# btree_depth t5;;
- : int = 2
```


[TD] La fonction *elem*

- La fonction `btree_elem` permet de tester si un élément `e` appartient à un arbre :

```
# let rec btree_elem e t = match t with
    Empty -> false
  | Node (l, x, r) -> (x = e) || (btree_elem e l) ||
  (btree_elem e r);;
val btree_elem : 'a -> 'a btree -> bool = <fun>
# btree_elem "me" t4;;
- : bool = true
# btree_elem "you" t4;;
- : bool = false
```


[TD] La fonction *mirror*

- La fonction `btree_mirror` renvoie la version "miroir" d'un arbre binaire, c-à-d un arbre pour lequel tous les sous-arbres gauche et droite ont été inter-changés :

```
# let rec btree_mirror t = match t with
    Empty -> Empty
  | Node(l, x, r) -> Node(btree_mirror r, x,
  btree_mirror l);;
val btree_mirror : 'a btree -> 'a btree = <fun>
# btree_mirror t4;;
- : string btree =
  Node (Node (Empty, "me", Empty), "hello", Node
  (Empty, "its", Empty))
```

[TD] La fonctionnelle *map*

- Comme pour les listes, il est utile de définir des **fonctionnelles** génériques opérant sur les arbres binaires. La première, `map_btree`, renvoie l'arbre obtenu en appliquant une fonction `f` à toutes les étiquettes d'un arbre :

```
# let rec map_btree f t = match t with
    Empty -> Empty
  | Node(l, x, r) -> Node(map_btree f l, f x, map_btree f r);;
val map_btree : ('a -> 'b) -> 'a btree -> 'b btree = <fun>
# map_btree String.length t4;;
- : int btree = Node (Node (Empty, 3, Empty), 5, Node (Empty, 2,
  Empty))
```


[TD] La fonctionnelle *fold*

- La seconde, `fold_btree`, est analogue à la fonctionnelle `list_fold` vue au chapitre précédent. Elle permet de définir une opération `f` ternaire et son élément neutre `z` :

```
# let rec fold_btree f z t = match t with
    Empty -> z
  | Node(l, x, r) -> f (fold_btree f z l) x (fold_btree f z r);;
val fold_btree : ('a -> 'b -> 'a -> 'a) -> 'a -> 'b btree -> 'a
  = <fun>
# fold_btree (fun l x r -> l + x + r ) 0 t2;;
- : int = 12
```