

(4) Types sommes – Arbres binaires

Programmation fonctionnelle (LU2IN019)

Licence d'informatique
2022/2023

Mathieu Jaume – Adrien Koutsos



Types sommes

Types sommes avec constructeurs constants

- regrouper un nombre fini de « *constantes symboliques* » au sein d'un même type
 - ▶ type énuméré
- *exemple : couleurs d'un feu rouge*

```
type couleur = Vert | Orange | Rouge
```

- ▶ définition du type `couleur`
- ▶ définition de 3 nouvelles constantes : `Vert`, `Orange` et `Rouge`
- ▶ `Vert`, `Orange` et `Rouge` sont des **constructeurs** constants (sans arguments) de valeurs de type `couleur`
 - ★ les constructeurs commencent par une majuscule

Filtrage (*pattern matching*)

- expression de filtrage `match ... with ...`

```
# let x = Vert in
  match x with
  | Vert -> "je passe"
  | Orange -> "il faut s'arreter"
  | Rouge -> "il faut absolument s'arreter";;
- : string = "je passe"
```

- fonction sur une valeur de type somme

```
let next (x : couleur) : couleur =
  match x with
  | Vert -> Orange
  | Orange -> Rouge
  | Rouge -> Vert
```

```
# next Vert;;
- : couleur = Orange
```

Type somme (union disjointe)

- les constructeurs d'un type somme peuvent avoir des arguments
 - ▶ réunir dans un même type des valeurs de types différents

```
type nombre = Entier of int | Flottant of float
```

- à chaque « composante » correspond un constructeur du type somme
 - ▶ le type `nombre` correspond à l'« union disjointe » des types `int` et `float`
 - ▶ les constructeurs `Entier` et `Flottant` sont les injections canoniques des types `int` et `float` dans le type `nombre`
 - ▶ construction de valeurs

```
# Entier (7 * 8);;  
- : nombre = Entier 56  
# Flottant 5.5;;  
- : nombre = Flottant 5.5
```

Type somme

```
type nombre = Entier of int | Flottant of float
```

- filtrage

```
let float_of_nombre (n : nombre) : float =  
  match n with  
  | Entier x -> float_of_int x  
  | Flottant y -> y
```

```
# float_of_nombre (Entier 2);;
```

```
- : float = 2.
```

```
# float_of_nombre (Flottant (float_of_nombre (Entier 2)));;
```

```
- : float = 2.
```

- *rappel* : dans le corps de la fonction `float_of_nombre`

- ▶ `Entier x` et `Flottant y` sont des motifs de filtrage

- ▶ `x` et `y` sont des variables de filtrage

- ★ utilisables dans les expressions qui définissent le résultat de la fonction

Type somme paramétré par un type

exemple : type prédéfini 'a option

```
type 'a option = None | Some of 'a
```

- 'a est une variable de type
- une valeur de type 'a option est :
 - ▶ soit obtenue à partir du constructeur **None** : c'est une constante
 - ▶ soit obtenue en appliquant le constructeur **Some** sur une valeur de type 'a

```
# Some 3;;  
- : int option = Some 3  
# Some false;;  
- : bool option = Some false
```

- permet d'implanter une fonction partielle

```
let nombre_to_int (n : nombre) : int option =  
  match n with  
  | Entier k -> Some k  
  | _ -> None
```

Liste d'association avec type option

- *rappel* : liste d'association : (`'a`, `'b`) `list`
 - ▶ `'a` est le type des clés
 - ▶ `'b` est le type des valeurs
- recherche de la valeur associée à une clé dans une liste d'association
 - ▶ utilisation de la fonction prédéfinie `List.assoc`
 - ▶ utilisation du type prédéfini `'a option`
`type 'a option = None | Some of 'a`
résultat de la recherche
 - ★ si `List.assoc` retourne une valeur `v` alors la recherche retourne la valeur (`Some v`)
 - ★ si `List.assoc` lève l'exception `Not_found` alors la recherche retourne la valeur `None`
 ↪ rattrapage de l'exception `Not_found`

Rattrapage d'exceptions

- rattrapage d'une exception

`try e with E -> e'`

- ▶ si l'évaluation de `e` ne lève pas d'exception, alors le résultat de l'évaluation de `try e with E -> e'` est le résultat de l'évaluation de `e`
- ▶ si l'évaluation de `e` lève l'exception `E`, alors
 - ★ l'exception `E` est dite rattrapée
 - ★ l'expression `e'` définit le résultat à retourner dans ce cas (et peut contenir des variables de filtrage si le motif de filtrage `E` correspond à une exception paramétrée)
- ▶ si l'évaluation de `e` lève une exception `E'` différente de `E`,
`try e with E -> e'` lève l'exception `E'`
 - ★ l'exception `E'` n'est pas rattrapée

- comme pour un `match ... with ...`, on peut rattraper plusieurs exceptions différentes

`try e with E1 → e1 | ... | En → en`

Liste d'association avec type option

- recherche de la valeur associée à une clé dans une liste d'association

```
let assoc_opt (k : 'a) (l : ('a*'b) list) : 'b option =  
  try Some (List.assoc k l) with  
  | Not_found -> None
```

```
# assoc_opt "b" [("c", 3); ("b", 2); ("a", 1)];;  
- : int option = Some 2  
# assoc_opt "d" [("c", 3); ("b", 2); ("a", 1)];;  
- : int option = None
```

- la construction `try ... with ...` permet de rattraper une exception

Liste d'association avec type option

- fonction qui détermine la valeur associée à une clé `k` dans une liste d'association et retourne une valeur par défaut `dflt` si aucune valeur n'est associée à la clé `k`

```
let assoc_dflt (k:'a) (dflt:'b) (l : ('a*'b) list) : 'b =  
  match assoc_opt k l with  
  | None -> dflt  
  | Some x -> x
```

```
# assoc_dflt "b" 0 [("c",3);("b",2);("a",1)];;  
- : int = 2  
# assoc_dflt "d" 0 [("c",3);("b",2);("a",1)];;  
- : int = 0
```

Arbre binaires

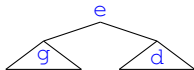
Arbres binaires

- les mêmes que dans l'UE de mathématiques discrètes

- ▶ l'ensemble `'a btree` des arbres binaires étiquetés par les éléments d'un alphabet `'a` est défini inductivement par :

(base) **Empty** (arbre vide) est un arbre binaire

(induction) si `g` et `d` sont des arbres binaires, alors (pour tout élément `e` de l'alphabet `'a`), **Node** (`e`, `g`, `d`) est un arbre binaire



- définition OCaml du type des arbres binaires :

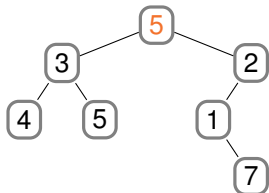
```
type 'a btree = Empty | Node of 'a * ('a btree) * ('a btree)
```

- ▶ type somme : deux constructeurs **Empty** et **Node**
- ▶ type récursif : un arbre binaire non vide est défini à partir de deux arbres binaires
- ▶ type polymorphe : paramétré par une variable de type (`'a`) désignant le type des éléments qui étiquettent les nœuds de l'arbre

Arbres binaires

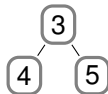
```
type 'a btree = Empty | Node of 'a * ('a btree) * ('a btree)
```

exemple : arbre t

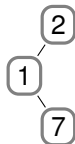


- ▶ 1,2,3,4,5,7 sont les étiquettes des nœuds de t
- ▶ 5, 3, 2, 4, 5, 1, 7 sont les nœuds de t

- ▶ la racine de t est le nœud 5
- ▶ sous-arbre gauche de t :



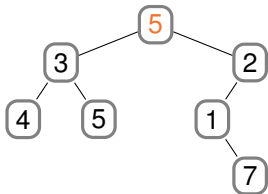
- ▶ sous-arbre droit de t :



Arbres binaires

```
type 'a btree = Empty | Node of 'a * ('a btree) * ('a btree)
```

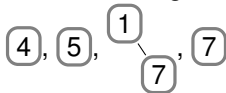
exemple : arbre t



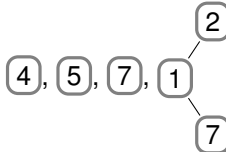
► 4, 5 et 7 sont les feuilles de t

► **Empty** est :

► le sous-arbre gauche de



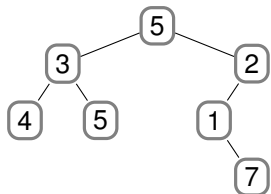
► le sous-arbre droit de



Arbres binaires

```
type 'a btree = Empty | Node of 'a * ('a btree) * ('a btree)
```

exemple : arbre t



```
#let t =  
    Node(5, Node(3,  
                Node(4, Empty, Empty),  
                Node(5, Empty, Empty)),  
        Node(2,  
            Node(1,  
                Empty,  
                Node(7, Empty, Empty)),  
            Empty));;  
val t : int btree = ...
```

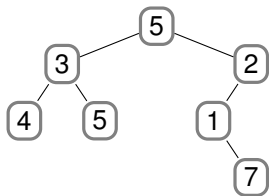

Arbres binaires : filtrage

```
type 'a btree = Empty | Node of 'a * ('a btree) * ('a btree)
```

- étiquette de la racine d'un arbre binaire non vide

```
let label_root (t : 'a btree) : 'a =  
  match t with  
  | Empty -> raise (Invalid_argument "label_root")  
  | Node (e,_,_) -> e
```

arbre t



```
# label_root t;;  
- : int = 5
```

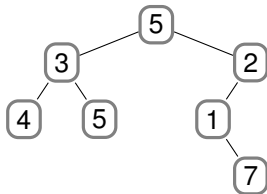
Arbres binaires : filtrage

- sous-arbres gauche et droit d'un arbre binaire non vide

```
let sag (t : 'a btree) : 'a btree =  
  match t with  
  | Empty -> raise (Invalid_argument "sag")  
  | Node (_, g, _) -> g
```

```
let sad (t : 'a btree) : 'a btree =  
  match t with  
  | Empty -> raise (Invalid_argument "sad")  
  | Node (_, _, d) -> d
```

arbre t



```
# sag t;;  
- : int btree =  
  Node (3, Node (4, Empty, Empty),  
        Node (5, Empty, Empty))  
  
# sad (sad t);;  
- : int btree = Empty
```

Hauteur d'un arbre binaire

- dans l'UE de mathématiques discrètes

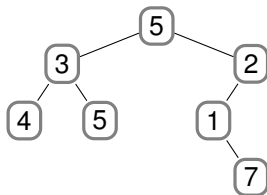
$$h(\text{Empty}) = 0$$

$$h(\text{Node}(e, g, d)) = 1 + \max(h(g), h(d))$$

- dans l'UE de programmation fonctionnelle

```
let rec height (t : 'a btree) : int =  
  match t with  
  | Empty -> 0  
  | Node(_, g, d) -> 1 + max (height g) (height d)
```

arbre t



```
# height t;;  
- : int = 4
```

Taille (nombre de nœuds) d'un arbre binaire

- dans l'UE de mathématiques discrètes

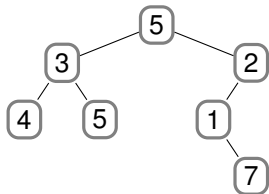
$$\mathcal{N}(\text{Empty}) = 0$$

$$\mathcal{N}(\text{Node}(e, g, d)) = 1 + \mathcal{N}(g) + \mathcal{N}(d)$$

- dans l'UE de programmation fonctionnelle

```
let rec size (t:'a btree) : int =  
  match t with  
  | Empty -> 0  
  | Node (_, g, d) -> 1 + (size g) + (size d)
```

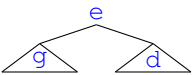
arbre t



```
# size t;;  
- : int = 7
```

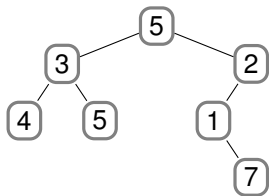
Recherche d'une étiquette dans un arbre binaire

- l'étiquette x n'apparaît pas dans l'arbre vide **Empty**

- l'étiquette x apparaît dans l'arbre  ssi :
 $e = x$ ou x apparaît dans l'arbre g ou x apparaît dans l'arbre d

```
let rec mem (x : 'a) (t : 'a btree) : bool =  
  match t with  
  | Empty -> false  
  | Node(e, g, d) -> x=e || mem x g || mem x d
```

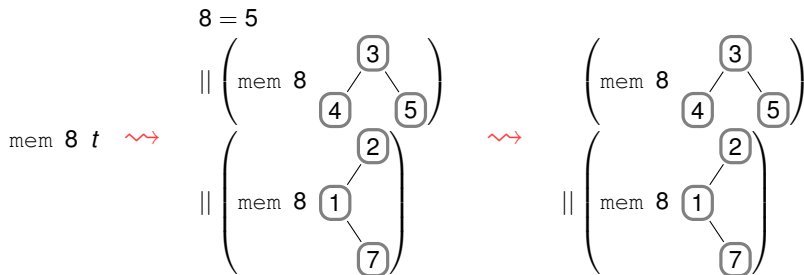
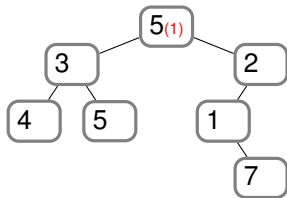
arbre t



```
# mem 3 t;;  
- : bool = true  
# mem 8 t;;  
- : bool = false
```

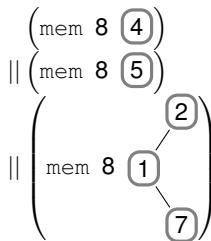
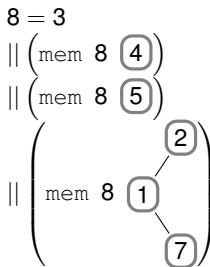
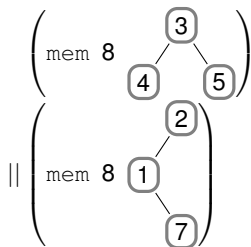
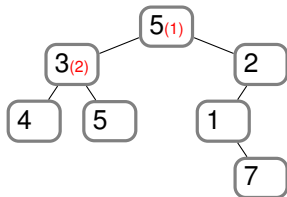
Recherche de l'étiquette 8 dans l'arbre binaire t

```
let rec mem (x : 'a) (t : 'a btree) : bool =  
  match t with  
  | Empty -> false  
  | Node (e,g,d) -> x=e || mem x g || mem x d
```



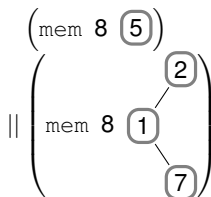
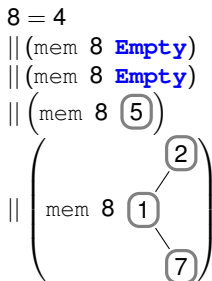
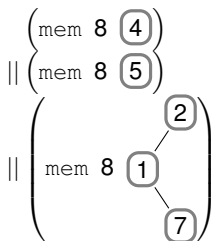
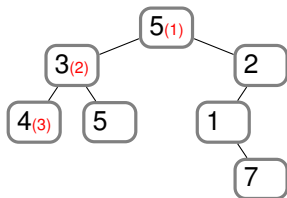
Recherche de l'étiquette 8 dans l'arbre binaire t

```
let rec mem (x : 'a) (t : 'a btree) : bool =  
  match t with  
  | Empty -> false  
  | Node(e,g,d) -> x=e || mem x g || mem x d
```



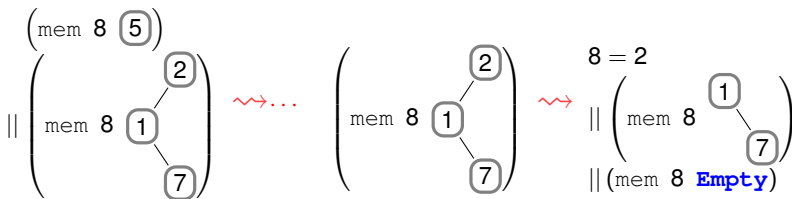
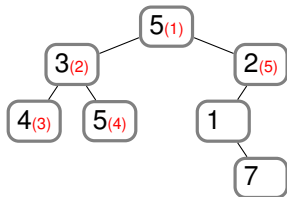
Recherche de l'étiquette 8 dans l'arbre binaire t

```
let rec mem (x : 'a) (t : 'a btree) : bool =
  match t with
  | Empty -> false
  | Node(e,g,d) -> x=e || mem x g || mem x d
```



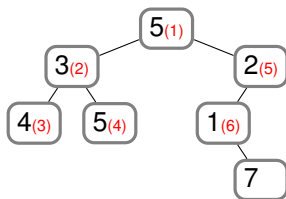
Recherche de l'étiquette 8 dans l'arbre binaire t

```
let rec mem (x : 'a) (t : 'a btree) : bool =
  match t with
  | Empty -> false
  | Node (e, g, d) -> x=e || mem x g || mem x d
```



Recherche de l'étiquette 8 dans l'arbre binaire t

```
let rec mem (x : 'a) (t : 'a btree) : bool =  
  match t with  
  | Empty -> false  
  | Node (e, g, d) -> x=e || mem x g || mem x d
```



8 = 2

$\parallel \left(\text{mem } 8 \begin{array}{c} \boxed{1} \\ \diagdown \\ \boxed{7} \end{array} \right)$
 $\parallel (\text{mem } 8 \text{ Empty})$



$\left(\text{mem } 8 \begin{array}{c} \boxed{1} \\ \diagdown \\ \boxed{7} \end{array} \right)$
 $\parallel (\text{mem } 8 \text{ Empty})$

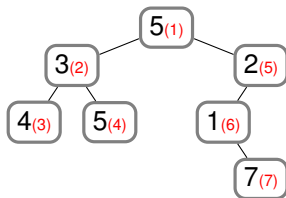


8 = 1

$\parallel (\text{mem } 8 \text{ Empty})$
 $\parallel (\text{mem } 8 \boxed{7})$
 $\parallel (\text{mem } 8 \text{ Empty})$

Recherche de l'étiquette 8 dans l'arbre binaire t

```
let rec mem (x : 'a) (t : 'a btree) : bool =  
  match t with  
  | Empty -> false  
  | Node (e,g,d) -> x=e || mem x g || mem x d
```



8 = 1

\parallel (mem 8 Empty) \rightsquigarrow (mem 8 Empty) \rightsquigarrow ... \rightsquigarrow false
 \parallel (mem 8 7) \rightsquigarrow (mem 8 7)
 \parallel (mem 8 Empty)

Parcours en profondeur d'un arbre binaire

- dans l'UE de mathématiques discrètes (parcours préfixe)

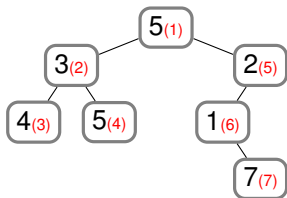
$$pre(\text{Empty}) = \varepsilon$$

$$pre(\text{Node}(e, g, d)) = e \cdot pre(g) \cdot pre(d)$$

- dans l'UE de programmation fonctionnelle

```
let rec pre (t : 'a btree) : 'a list = match t with  
| Empty -> []  
| Node(e, g, d) -> e :: ((pre g) @ (pre d))
```

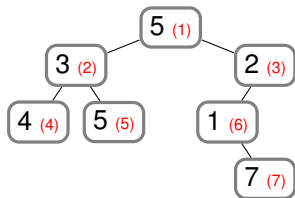
arbre t



```
# pre t;;  
- : int list = [5; 3; 4; 5; 2; 1; 7]
```

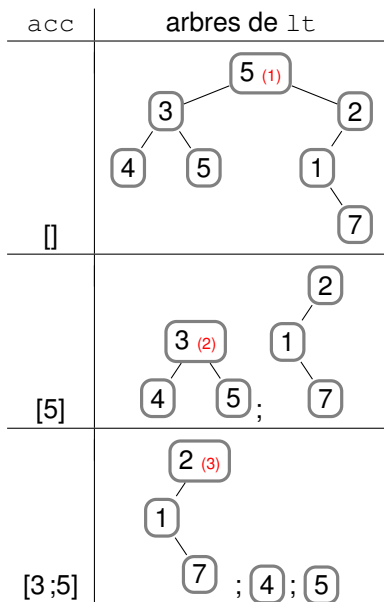
Parcours en largeur d'un arbre binaire

- la liste des arbres à traiter est utilisée comme une file (*First In First Out*)
 - les sommets sont énumérés dans l'ordre du parcours en largeur

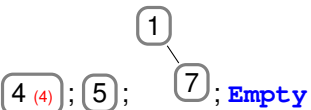
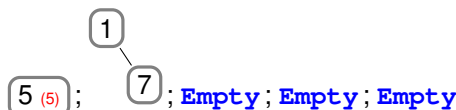
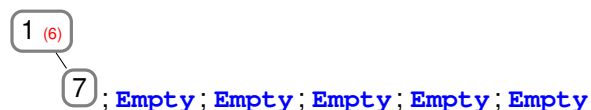


```
let breadth (t : 'a btree) : 'a list =  
  let rec aux acc lt =  
    match lt with  
    | [] -> List.rev acc  
    | Empty :: tlt -> aux acc tlt  
    | Node (e,g,d) :: tlt -> aux (e :: acc) (tlt @ [g; d])  
  in  
  aux [] [t]  
  
# breadth t;;  
- : int list = [5; 3; 2; 4; 5; 1; 7]
```

Parcours en largeur d'un arbre binaire



Parcours en largeur d'un arbre binaire

acc	arbres de lt
[2;3;5]	
[4;2;3;5]	
[5;4;2;3;5]	
[1;5;4;2;3;5]	<p>Empty; Empty; Empty; Empty; Empty; Empty; 7</p>
[1;5;4;2;3;5]	<p>⋮</p> <p>7</p>

Parcours en largeur d'un arbre binaire

acc	arbres de 1t
[1 ; 5 ; 4 ; 2 ; 3 ; 5]	7 ⁽⁷⁾
[7 ; 1 ; 5 ; 4 ; 2 ; 3 ; 5]	Empty ; Empty
[7 ; 1 ; 5 ; 4 ; 2 ; 3 ; 5]	Empty
[7 ; 1 ; 5 ; 4 ; 2 ; 3 ; 5]	

~> [5 ; 3 ; 2 ; 4 ; 5 ; 1 ; 7]

- implantation d'une file avec une liste : ajout en fin de liste avec l'opérateur de concaténation
 - ▶ coûteux en nombres d'opérations