

Programmation Fonctionnelle



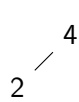
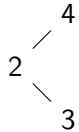
Emmanuel Chailloux


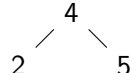
30 juin 2021

Plan du cours 5:

- ▶ Arbres de recherche (suite)
- ▶ Arbres généraux (n-aires)
- ▶ Arbres lexicaux (tries)
- ▶ Représentation mémoire (des valeurs et des appels)
- ▶ Entrées/sorties
- ▶ Conclusions et bibliographie

Insertion dans arbre de recherche

fonction insert: : insert 3 () = 

insert 5 () = 

```
1 let rec insert (key: 'a) (t: 'a btree): 'a btree =  
2   match t with  
3   | Node(g, x, d) ->  
4     if key < x then  
5       Node(insert key g, x, d)  
6     else if key > x then  
7       Node(g, x, insert key d)  
8     else t  
9   | Empty -> Node(Empty, key, Empty)
```

Création d'un arbre de recherche à partir d'une liste

▶ récursion simple

```
1 let rec from_list (xs:'a list) : 'a btree =  
2   match xs with  
3     | [] -> Empty  
4     | x::xs -> (insert x (from_list xs))
```

▶ récursion terminale

```
1 let from_list2 (xs:'a list) : 'a btree =  
2   let rec loop (xs:'a list) (r:'a btree) : 'a btree =  
3     match xs with  
4       | [] -> r  
5       | x::xs -> (loop xs (insert x r))  
6   in  
7     (loop xs Empty)
```

▶ en utilisant fold_left

```
1 let from_list3 (xs:'a list) : 'a btree =  
2   List.fold_left (fun r x -> (insert x r)) Empty xs
```

Arbres généraux (ou n-aires) (1)

Généralisation des arbres binaires: : chaque nœuds peut avoir un nombre quelconque de sous-arbres.

La liste des sous-arbres attachée à un nœud est appelée forêt. Ainsi une arbre général est :

- ▶ soit vide ;
- ▶ soit composé d'une étiquette et d'une forêt.

et une forêt est :

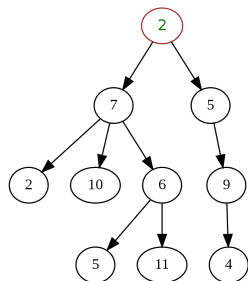
- ▶ soit vide ;
- ▶ soit composée d'un arbre et d'une forêt.

On parlera d'arbres généraux ou n-aires.

Arbres généraux (2)

Type et valeurs: :

```
1 type 'a gtree =  
2   Empty  
3   | Node of ('a * ('a gtree) list)
```



```
1 # let gt =  
2 Node(2,  
3   [ Node(7, [ Node( 2, []);  
4     Node(10, []);  
5     Node( 6, [ Node(5, []);  
6       Node(11, [])  
7       ])  
8  
9     ] ) ;  
10 Node(5, [ Node(9, [Node(4, [])  
11     ])  
12     ])  
13 ] ) ;;  
14 val gt : int gtree = ...
```

mem : recherche d'un élément (1)

Il est à noter que le parcours d'un arbre n-aire nécessite un double parcours :

- ▶ arborescent pour la structure de l'arbre
- ▶ et linéaire pour le parcours de la forêt.

```
1 let rec gtree_mem (z:'a) (gt : 'a gtree) : bool =
2   match gt with
3   | Empty -> false
4   | Node(x, gts) -> (z=x) || (forest_mem z gts)
5 and forest_mem z (gts:( 'a gtree) list) : bool =
6   match gts with
7   | [] -> false
8   | gt::gts -> (gtree_mem z gt) || (forest_mem z gts)
```

```
1 # gtree_mem 5 gt ;;
2 - : bool = true
3 # gtree_mem 3 gt ;;
4 - : bool = false
5 # forest_mem 5 [gt] ;;
6 - : bool = true
```

Utilisation d'itérateurs sur les listes (1)

mem : recherche d'un élément:

▶ List.exists

```
1 # List.exists;;  
2 - : ('a -> bool) -> 'a list -> bool = <fun>
```

```
1 # let rec gtree_mem (z:'a) (gt : 'a gtree) : bool =  
2   match gt with  
3   | Empty -> false  
4   | Node(x, gts) -> (x=z) || (List.exists (gtree_mem z) gts)      ;;  
5 val gtree_mem : 'a -> 'a gtree -> bool = <fun>  
6  
7 # gtree_mem 5 gt;;  
8 - : bool = true  
9  
10 # gtree_mem 3 gt;;  
11 - : bool = false
```


Utilisation d'itérateurs sur les listes (2)

► List.fold_left et List.map

```
1 # List.fold_left;;
2 - : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
3 # List.map;;
4 - : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
1 # let rec size (gt : 'a gtree) : int = match gt with
2   Empty -> 0
3   | Node (_,l) ->
4     1 + (List.fold_left (+) 0 (List.map size l));;
5 val size : 'a gtree -> int = <fun>
6
7 # let rec height (gt : 'a gtree) : int = match gt with
8   Empty -> 0
9   | Node (_,l) ->
10    1 + (List.fold_left (max) 0 (List.map height l));;
11 val height : 'a gtree -> int = <fun>
```

Autres exemples

- ▶ `sum` : même schéma que `size` et `height`, mais type différent

```
1 # let rec sum (gt : int gtree) : int = match gt with
2   Empty -> 0
3   | Node (e,l) ->
4     e + (List.fold_left (+) 0 (List.map sum l));;
5 val sum : int gtree -> int = <fun>
```

- ▶ autre version de `mem` :

```
1 # let rec mem (e : 'a) (gt : 'a gtree) =
2   let rec forest_mem (l : 'a gtree list) =
3     match l with
4     | [] -> false
5     | h::t -> mem e h || forest_mem t
6   in
7     match gt with
8     | Empty -> false
9     | Node (ne,l) ->
10       ne = e || forest_mem l ;;
11 val mem : 'a -> 'a gtree -> bool = <fun>
```

Parcours en largeur

En utilisant le module Fqueue :

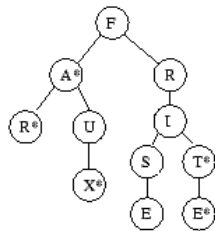
```
1 # let add_list (xs:'a list) (ys:'a Fqueue.t) : 'a Fqueue.t =
2   List.fold_left (fun r x -> add x r) ys xs
3 val add_list : 'a list -> 'a Fqueue.t -> 'a Fqueue.t = <fun>
4
5 let rec bflist (gt : 'a gtree) : 'a list =
6   let rec loop (gts : ('a gtree) Fqueue.t) : 'a list =
7     if (is_empty gts) then [] else (
8       match (top_pop gts) with
9       | Empty, gts -> (loop gts)
10      | Node(x,gts1), gts -> x::(loop (add_list gts1 gts)) )
11   in
12   (loop (add gt (create())))
13 val bflist : 'a gtree -> 'a list = <fun>
14
15 # bflist gt;;
16 - : int list = [2; 7; 5; 2; 10; 6; 9; 5; 11; 4]
```

Arbres lexicaux (ou tries) (1)

Pour la représentation de dictionnaires, on utilisera des arbres lexicaux (ou tries).

```
1 # type noeud_lex = Lettre of char * bool * arbre_lex
2   and arbre_lex = noeud_lex list;;
3 # type mot = string;;
```

La valeur booléenne du `noeud_lex` marque la fin d'un mot lorsqu'elle vaut `true`. Dans une telle structure, la suite de mots « fa, far, faux, frise, frit, frite » est stockée de la façon suivante :



```
1 # let dico =
2   Lettre ('F',false,
3     [ Lettre ('A',true,
4       [ Lettre ('R',true,[]);
5         Lettre ('U',false,[ Lettre ('X',true,[])])]);
6     Lettre ('R',false,[
7       Lettre ('I',false,
8         [ Lettre ('S',false,[Lettre ('E',true,[])]);
9           Lettre ('T',true,[Lettre ('E',true,[])]);
10        ])
11     ])) ;;
12 val dico : noeud_lex = ...
```

Fonctions sur les dictionnaires (1)

- ▶ existe qui teste si un mot appartient à un dictionnaire de type arbre_lex.

```
1 # let rec existe m d =
2   let aux sm i n =
3     match d with
4     [] -> false
5     | (Lettre (c,b,l))::q ->
6       if c = sm.[i] then (* if x = String.get i sm then *)
7         if n = 1 then b
8         else existe (String.sub sm (i+1) (n-1)) l
9       else existe sm q
10  in aux m 0 (String.length m);;
11 val existe : string -> arbre_lex -> bool = <fun>
12
13 # existe "FLAN" [dico];;
14 - : bool = false
15
16 # existe "FRIT" [dico];;
17 - : bool = true
```

Fonctions sur les dictionnaires (2)

- ▶ ajoute prend un mot et un dictionnaire et retourne un nouveau dictionnaire qui contient ce mot en plus. Si le mot existe déjà, il n'est pas nécessaire de l'ajouter.

```
1 # let rec ajoute m d =
2   let aux sm i n =
3     if n = 0 then d else
4     match d with
5     | [] -> [Lettre (sm.[i], n = 1,
6                 ajoute (String.sub sm (i+1) (n-1)) [])]
7     | (Lettre(c,b,l))::q ->
8       if c = sm.[i] then
9         if n = 1 then (Lettre(c,true,l))::q
10        else Lettre(c,b,ajoute (String.sub sm (i+1) (n-1)) l)::q
11       else (Lettre(c,b,l))::(ajoute sm q)
12   in aux m 0 (String.length m);;
13 val ajoute : string -> arbre_lex -> arbre_lex = <fun>
14
15 # existe "FLAN" (ajoute "FLAN" [dico]);;
16 - : bool = true
```

Fonctions sur les dictionnaires (3)

- ▶ `construit` prend une liste de mots et construit le dictionnaire correspondant.
- ▶ `verifie` prend une liste de mots et un dictionnaire et retourne la liste de mots n'appartenant pas à ce dictionnaire.

```
1 # let construit l =
2   let rec aux l d =
3     match l with
4       [] -> d
5       | t::q -> aux q (ajoute t d)
6   in
7     aux l [];;
8 val construit : string list -> arbre_lex = <fun>
9
10 # let verifie l d =
11   List.filter (function x -> not (existe x d)) l;;
12 val verifie : string list -> arbre_lex -> string list = <fun>
13
14 # verifie ["LE"; "FAUX"; "FAR"; "FRIT"; "NE" ; "FRISE"; "PAS"] [dico];;
15 - : string list = ["LE"; "NE"; "PAS"]
```

Valeurs : représentation uniforme

Nécessité de parcourir les valeurs :

- ▶ Fonctions primitives génériques : égalité, sérialisation, etc.
- ▶ Gestion mémoire (cf. suite du cours)
- ▶ Introspection, affichage générique, etc.

Solution logique : **uniformiser la structure des valeurs**

Question centrale : distinction entre

- ▶ Valeurs immédiates (entiers, caractères, etc.)
- ▶ Valeurs allouées (tableaux, structures, etc.)
- ▶ Différentes sortes de valeurs allouées.

En machine : **un pointeur = un entier = un mot machine**

Valeurs : solution plus avancées

Bit(s) discriminant(s) :

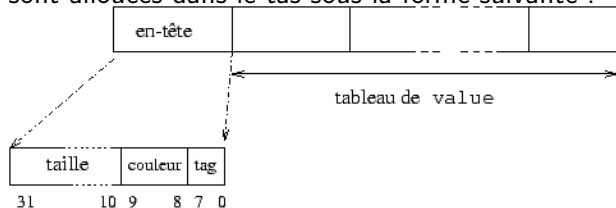
- ▶ On mange un bit sur le mot machine pour discriminer entre entier et pointeur
- ▶ Éventuellement plus de bits pour plusieurs types d'immédiats
- ▶ On utilise un système de tags comme précédemment pour les valeurs allouées
- ▶ On limite l'étendue des immédiats

En OCaml tout ce qui peut être représenté par un entier : entier, caractère, booléen, constructeur constant, est codé dans un entier et tout ce qui est plus gros qu'un entier est alloué dans le tas .

Le pointeur de la zone allouée occupe un mot mémoire, comme les entiers.

Représentation des valeurs allouées en OCaml (1)

Les constructeurs paramétrés, les enregistrements et les fermetures sont alloués dans le tas sous la forme suivante :



le tableau de valeur correspond aux champs pour un enregistrement ou un n-uplet, ou l'environnement et le code d'une fermeture. L'entête contient la taille du tableau de valeur, 2 bits pour le gestionnaire mémoire, et un tag correspondant au constructeur qui a alloué cette valeur.

Représentation des valeurs allouées en OCaml (2)

On reprend la fonction `calcule_taille` pour explorer les valeurs :

```
1 # let calcule_taille o = Obj.reachable_words (Obj.repr o) ;;
2 val calcule_taille : 'a -> int = <fun>
```

On commence par des valeurs allouées plates que l'on définit spécialement :

<pre>1 # type ty2 = Ctor2 of int * ↵ int;; 2 type ty2 = Ctor2 of int * int 3 # let v2 = Ctor2 (3,4);; 4 val v2 : ty2 = Ctor2 (3, 4) 5 # calcule_taille v2;; 6 - : int = 3</pre>	<pre>1 # type ty5 = Ctor5 of int * int * int * ↵ int * int ;; 2 type ty5 = Ctor5 of int * int * int * int↵ * int 3 # let v5 = Ctor5 (3,6,9,12,15);; 4 val v5 : ty5 = Ctor5 (3, 6, 9, 12, 15) 5 # calcule_taille v5;; 6 - : int = 6</pre>
---	--

- ▶ `v2` : a 2 valeurs : 3 et 4 dans le tableau de valeur, son entête contient cette longueur de 2 est dans l'entête ainsi que le numéro de constructeur `Ctor2` dans l'octet de tag , au total :
2 valeur + 1 entête = 3 mots
- ▶ `v5` : c'est pareil mais avec un tableau de 5 valeurs = 6 au total

Représentation des valeurs allouées en OCaml (3)

Calcul sur une structure dynamique:

- ▶ Le constructeur `::` des listes peut être considéré comme le `Ctor2` de l'exemple précédent.
- ▶ Le constructeur constant `[]`, comme tous les constructeurs constants, est codé par un entier.

```
1 # let c = 4::[];;
2 val c : int list = [4] (* taille 3 *)
3 # let d = 22::c;;
4 val d : int list = [22; 4] (* taille 6 *)
5 # let e = 77::d;;
6 val e : int list = [77; 22; 4] (* taille 9 *)
7 # List.map calculer_taille [c;d;e];;
8 - : int list = [3; 6; 9]
```

- ▶ `c` a 2 valeurs 4 et `[]` + l'entête
- ▶ `d` prend 3 mots + la taille de `c`
- ▶ `e` prend 3 Mots plus la taille de `d`

Pile ou tas

- ▶ size1 : utilise la pile des appels de fonction
- ▶ size2 : récursive terminale mais utilise une pile à la main

```
1 # let rec size1 (bt:'a btree) : int =
2     match bt with
3     | Empty -> 0
4     | Node(_, bt1, bt2) -> 1 + (size1 bt1) + (size1 bt2) ;;
5 val size1 : 'a btree -> int = <fun>
6
7 # let rec size_aux (bt:'a btree) (bts:( 'a btree) list) (r:int) =
8     match bt with
9     | Node (_,bt1,bt2) -> size_aux bt1 (bt2::bts) (r+1)
10    | Empty -> ( match bts with
11                  | [] -> r
12                  | bt::bts -> size_aux bt bts r
13                  ) ;;
14 val size_aux : 'a btree -> 'a btree list -> int -> int = <fun>
15 # let size2 (bt:'a btree) : int = size_aux bt [] 0 ;;
16 val size2 : 'a btree -> int = <fun>
```

Quid pour height? \Rightarrow lire

“Mesurer la hauteur d’un arbre” - JC Filliâtre - JFLA 2020

<https://hal.inria.fr/hal-02427360/document>

Canaux:

- ▶ types : *in_channel* et *out_channel*
- ▶ fonctions : *open_in* : *string* → *in_channel* (*close_in*)
open_out : *string* → *out_channel* (*close_out*)
- ▶ exception : *End_of_file*
- ▶ canaux prédéfinis : *stdin*, *stdout* et *stderr*
- ▶ fonctions de lecture et d'écriture sur les canaux
- ▶ organisation et accès séquentiels
- ▶ type *open_flag* pour les modes d'ouverture

Principales fonctions d'ES

<code>input</code>	: <code>in_channel</code> \rightarrow <code>bytes</code> \rightarrow <code>int</code> \rightarrow <code>int</code> \rightarrow <code>int</code>
<code>input_line</code>	: <code>in_channel</code> \rightarrow <code>string</code>
<code>output</code>	: <code>out_channel</code> \rightarrow <code>bytes</code> \rightarrow <code>int</code> \rightarrow <code>int</code> \rightarrow <code>unit</code>
<code>output_string</code>	: <code>out_channel</code> \rightarrow <code>string</code> \rightarrow <code>unit</code>
<code>output_bytes</code>	: <code>out_channel</code> \rightarrow <code>bytes</code> \rightarrow <code>unit</code>
<code>read_line</code>	: <code>unit</code> \rightarrow <code>string</code>
<code>read_int</code>	: <code>unit</code> \rightarrow <code>int</code>
<code>print_string</code>	: <code>string</code> \rightarrow <code>unit</code>
<code>print_bytes</code>	: <code>bytes</code> \rightarrow <code>unit</code>
<code>print_int</code>	: <code>int</code> \rightarrow <code>unit</code>
<code>print_newline</code>	: <code>unit</code> \rightarrow <code>unit</code>

string : chaînes immutables

bytes : chaînes mutables

unit : type ne possédant qu'une seule valeur ()

Exemple : C+/C-

Dans les déclarations locales :

```
1 let x = e1 in e2
```

l'expression e1 est évaluée avant l'expression e2,
x peut être un motif.

```
1 # let rec cpcm n =
2     let _ = print_string "taper un nombre : " in
3     let i = read_int () in
4         if i = n then print_string "BRAVO\n\n"
5         else let _ = (if i < n then print_string "C+\n"
6                     else print_string "C-\n")
7                 in cpcm n;;
8 val cpcm : int -> unit = <fun>
9
10 # cpcm 64;;
11 taper un nombre : 88
12 C-
13 taper un nombre : 44
14 C+
```


Conclusion

programmation fonctionnelle:

- ▶ grande expressivité
- ▶ modèle élégant de programmation
la concision rend lisible le code
- ▶ plus sûre (moins de risque d'erreur) :
 - ▶ travaille par copie mémoire, pas par modification physique
 - ▶ itérateurs puissants et faciles à comprendre (paramètres fonctionnels)
 - ▶ et aussi grâce (au typage statique et) à la gestion automatique de la mémoire
- ▶ tout en restant efficace :
 - ▶ en consommation mémoire (optimisation récursion terminale)
 - ▶ en optimisation des application totales
- ▶ avec des environnements de développement riches