Programmation fonctionnelle

Xavier Urbain

2024/2025

Programmation fonctionnelle

Aspects principaux ici :

- Tout est valeur (expressions, fonctions de première classe)
- Persistance (JAMAIS de modification)
  - ⇝ Simplification du code et donc correction
  - ⇝ Efficacité (undo gratuit)

Programmation similaire à raisonnement mathématique (modulo scories)
⇝ Abstraction, programmation de haut niveau (*pourquoi* ≠ *comment*)

Bonus modèle formel : λ-calcul $x \quad (M_1\ M_2) \quad \lambda x.M \quad \underset{\beta}{\rightarrow}$
capture tout programme

⇝ Preuve de correction aisée et naturelle induction/récurrence

Programmation fonctionnelle langages

Haskell, Scheme, Lisp, ML, Erlang, Gallina...

OCAML développé à l'INRIA `http://caml.inria.fr`

- Fonctionnel + traits impératifs ici : noyau fonctionnel et exceptions
- Fortement typé les types sont nos amis !
- Évaluation stricte arguments toujours évalués en premier

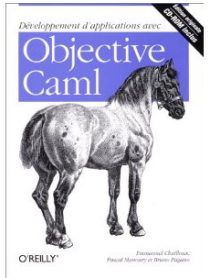
- Compilé ⇝ vers natif, efficace
- Interprété ⇝ vers bytecode, générique
- Boucle d'interaction ⇝ pratique

Persistance $\Longrightarrow$ gestion mémoire efficace, très complexe, cachée

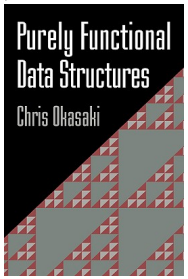
CE COURS N'EST PAS UN COURS D'OCAML

mais on va quand même apprendre la syntaxe

Biblio



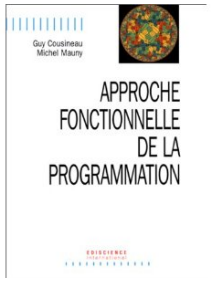
- Développement d'applications avec OCaml
- Chailloux, Manoury, Pagano
- O'Reilly
- `http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/`



- Purely Functional Data Structures
- Okasaki
- Cambridge University Press

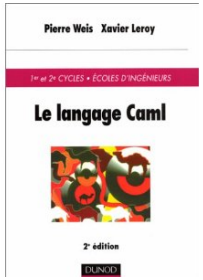
SML !

Biblio



- Approche fonctionnelle de la programmation
- Cousineau, Mauny
- Édiscience/Dunod

Camllight !



- Le langage Caml
- Leroy, Weis
- Dunod

Biblio

- Apprendre à programmer avec OCaml
- Conchon, Filliâtre
- Eyrolles

Également impératif. . .

Noyau fonctionnel

exploration

Programme = suite finie

(éval. dans l'ordre)

- Expressions → valeurs
- Déclarations → noms

Séparateur (expr) `;;`

Valeurs basiques. . .

Comparaisons, opérations logiques. . .

Arithmétique. . .

Mémoire... rappels

En gros :

...
1101010100010111100101010110111111001101010000 1110
0010101011011111000101010001011111000110101000 101
1000010101000101010101101000010111100010101000101
0000000000000000000000010110111101000101010000111
...

00101010110111111001101010000 ?

Mémoire... rappels

En gros :

...
1101010100010111100101010110111111001101010000 1110
0010101011011111000101010001011111000110101000 101
1000010101000101010101101000010111100010101000101
0000000000000000000000010110111101000101010000111
...

00101010110111111001101010000 ?

Segmentation ?

Noyau fonctionnel exploration

Programme = suite finie

- Expressions → valeurs
- Déclarations → noms

Séparateur (expr) `;;`

Valeurs basiques... typées

Comparaisons, opérations logiques... sur même type

Arithmétique... sur même type, sans implicite

Fortement typé : jamais de problèmes de confusion (ni segfault)

Noyau fonctionnel expressions

Branchement

```
if c then value1 else value2
```

Type de `c` bool et même type pour `value1` et `value2`

Égal à `value1` si `c` vaut **`true`**
Égal à `value2` si `c` vaut **`false`**

```
if (2.718 > 3.141) then 666 else 42 ;;     (* vaut 42 *)
```

Remarque : branchement ≠ fonction logique

```
if   condition  then true else false ;;
```

Noyau fonctionnel — expressions

Branchement

```
if c then value1 else value2
```

Type de `c` bool et même type pour `value1` et `value2`

Égal à `value1` si `c` vaut **`true`**
Égal à `value2` si `c` vaut **`false`**

```
if (2.718 > 3.141) then 666 else 42 ;;     (* vaut 42 *)
```

Remarque : branchement ≠ fonction logique

```
condition  ;;
```


Noyau fonctionnel — expressions

Encore des expressions !

```
(0x2ff_df_aa = 50_323_370) or (3 < 666) ;;

3 + (if 42 <= 666 then 1 else -272) ;;
```

- Manipulation d'expressions (tout au même niveau)
- Bon typage nécessaire :
  - Mêmes types des deux côtés dans branchement
  - Mêmes types pour comparaisons, etc.
- Pas de conversion implicite

Noyau fonctionnel — déclarations

Donner un nom à une expression globalement ou localement

```
let x = expr          (* x : nom de la valeur d'expr *)
```

```
let toto = 40 + 2                                  {(toto,42),...}
```

Jamais modifié car persistance

Identificateur : expression

```
let titi=toto                                  {(titi,42),(toto,42),...}
let toto = 666                 {(toto,666),(titi,42),(toto,42),...}
```

`titi` ? ⇝ 42

Noyau fonctionnel — expressions avec déclarations

Donner un nom à une expression globalement ou localement

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

```
                                                   {(toto,42),...}
let titi = 111 * 6 in                  {(titi,666),(toto,42),...}
  titi * 666 * titi                    {(titi,666),(toto,42),...}
                                                   {(toto,42),...}
```

Identificateur viable dans expression interne — portée

Noyau fonctionnel expressions avec déclarations

Donner un nom à une expression globalement ou localement

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

```
                                          {(toto,42),...}
let titi = 111 * 6 in                     {(titi,666),(toto,42),...}
  titi * 666 * toto                       {(titi,666),(toto,42),...}
                                          {(toto,42),...}
```

Identificateur viable dans expression interne portée

Noyau fonctionnel expressions avec déclarations

Donner un nom à une expression globalement ou localement

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

```
                                          {(toto,42),...}
let toto = 111 * 6 in                     {(toto,666),(toto,42),...}
  toto * 666 * toto                       {(toto,666),(toto,42),...}
                                          {(toto,42),...}
```

ici : capture

Noyau fonctionnel déclarations

Enchaîner les déclarations

```
let x = e1 in
  let y = e2 in
    ...
```

Déclarations simultanées

```
let x = e1
and y = e2 in
    ...
```

Donner un nom à une expression ou pas

```
let _ = e1 in e2
```


Noyau fonctionnel fonctions

Objet comme les autres

```
  fun x -> M
```
$\lambda x.M$

Un paramètre formel, ici `x`, un corps `M` (expression)
Type : type paramètre → type de la valeur du corps obtenue

Évaluation de l'application :

1. Remplacement dans le corps du paramètre par une valeur
2. Évaluation du corps

Syntaxe application :

```
(my_function my_param)
```
(MN)

Objet comme les autres

```
fun x -> M
```
$\lambda x.M$

Un paramètre formel, ici `x`, un corps `M` (expression)
Type : si `M` : β quand `x`:α alors type de fonction : $\alpha \to \beta$

Évaluation de l'application :

1. Remplacement dans le corps du paramètre par une valeur
2. Évaluation du corps

Syntaxe application :
`(my_function my_param)` (MN)

Objet comme les autres

Nommer... (et localement)

```
let f = fun x -> M
```

Prendre en paramètre...

```
let f = fun x -> (x something)
```

Retourner comme valeur...

```
let f = fun x -> (fun y -> M1)
```


Prendre en paramètre... (ordre supérieur)

```
let f = fun x -> (⋯(x something)⋯)
```
(M au-dessus de la parenthèse)

Type : $(\alpha \to \beta) \to \gamma$ lorsque :
`something` : α, `x` : $\alpha \to \beta$ et `M` : γ sachant ça

Retourner comme valeur...

```
let f = fun x -> (fun y -> M1)
```

Type : $\alpha \to (\beta \to \gamma)$ lorsque `M1` : γ sachant `x` : α et `y` : β

$$(\alpha \to \beta) \to \gamma \neq \alpha \to (\beta \to \gamma)$$


```
let f = fun x1 -> ⋯ -> fun xn -> M
```

Donc de type (au pire) : $\alpha_1 \to \cdots \to \alpha_n \to \beta$

Remarque. — application partielle (tant que → dans type) `(f e1 ⋯ ek)` avec $k < n$

⇝ Nouvelles fonctions dépendant de l'environnement à la création
Procédé : clôture

Remarque. — Notation équivalente :

```
let f x1 ... xn = M
```

Remarque. — Pas d'évaluation « sous les λ »

```
let f1 = fun x -> fun y -> x * x + y * y

let f2 = fun x ->
  let x2 = x * x in fun y -> x2 + y * y
```

Pas équivalentes pour application partielle

```
f1 42   ⇝ fun y -> 42 * 42 + y * y

f2 42   ⇝ fun y -> 1764 + y * y
```


Pas que des types de base...

t_1 et t_2 types : $t_1 \times t_2$ type des couples (x_1, x_2) avec $x_1 : t_1$ et $x_2 : t_2$

(, , , ,)

$$\alpha \times (\beta \times \gamma) \to \delta \neq \alpha \times \beta \times \gamma \to \delta \neq \alpha \to \beta \to \gamma \to \delta$$

Types différents, taille fixe

```
let triplet = (42,true,"Cooper");;
```

Déstructuration

```
let (t1,t2,t3) = triplet;;
```

⇝ Nommages respectifs de `42`, **`true`** et `"Cooper"`

Cas particulier des paires

Fonctions d'accès :

- `fst` : $\alpha \times \beta \to \alpha$
  Élement de gauche
- `snd` : $\alpha \times \beta \to \beta$
  Élément de droite

Créer ses types...

Déclaration : **`type`** `t =` *type*

Affirmation : `(` *ident* : *type* `)`

```
type quadruplet = int * int * int * int;;

let t1 = 1,2,3,4;;
let t2 : quadruplet = 4,3,2,1;;
```

déclaration de types

Créer ses types...

Types produits avec noms de champs : enregistrements

```
type cplx = { real : float ; im : float };;

let i = { real = 0.0 ; im = 1.0 }
```

Accès : *ident•field_name*

```
i.real;;
```


constructeurs

Définir la structure d'une valeur.

```
type couleur = | Pique | Coeur          (* MAJUSCULE *)
               | Carreau | Trèfle
type rang = | Roi | Dame | Valet
            | Ordinaire of int      (* Étiquette int *)
```

Différencier par structure : filtrage

```
match valeur with          (* Tout ceci : expression *)
| motif1 -> e1
| motif2 -> e2
| ···                                (* Vus DANS L'ORDRE *)
```

Motif : constructeurs + variables Filtrer : ∃? remplacement idoine

constructeurs

Motif : constructeurs + variables Filtrer : ∃? remplacement idoine

Pas de nom de valeur dans motifs

```
let x =  ···

match x with
| Roi -> print_string "Wow"
| Ordinaire toto ->  print_string "Oooh"
```


constructeurs

Motif : constructeurs + variables Filtrer : ∃? remplacement idoine

Pas de nom de valeur dans motifs

```
let x =  ···
let y = 2
match x with
| Roi -> print_string "Wow"
| Ordinaire y ->  print_string "Oooh"
```

Noyau fonctionnel — constructeurs

Motif : constructeurs + variables — Filtrer : ∃? remplacement idoine

Pas de nom de valeur dans motifs

```
let x = ···

match x with
| Roi -> print_string "Wow"
| Ordinaire toto ->  print_string "Oooh"
| Dame | Valet -> print_string "OK"    (* Même valeur *)
```


Noyau fonctionnel — constructeurs

Motif : constructeurs + variables — Filtrer : ∃? remplacement idoine

Pas de nom de valeur dans motifs

```
let x = ···

match x with
| Roi -> print_string "Wow"
| Ordinaire toto ->  print_string "Oooh"
| _ -> print_string "OK"    (* Ordre donc en dernier *)
```


Noyau fonctionnel — filtrage

Expression donc utilisable partout

```
let f = fun x -> match x with  ···

let f = fun  motif  ->  ···          (* Déstructuration *)

let f = function motifS                  (* Filtrage *)
```


Noyau fonctionnel — filtrage

Expression donc utilisable partout

```
let f = fun x -> match x with  ···

let f = fun  motif  ->  ···          (* Déstructuration *)

let f = function motifS                  (* Filtrage *)
```

Structure profonde :

```
type carte = C of (couleur * rang)

match ident with
| C (Coeur, Dame) -> print_string "Qu'on␣lui␣coupe␣la␣tête␣!"
| _ -> ···
```

Noyau fonctionnel — inductifs

Le constructeur construit. . .

Type : description d'un plus petit ensemble clôt par construction

En général :

- Cas de base
- Construction du suivant

```
type tige = Base | Seg of tige

match ident with
| Base -> ···                    (* UN  CAS  PAR *)
| Seg x -> ···                   (* CONSTRUCTEUR *)
```


Noyau fonctionnel — récursion

Fonction récursive : cas de base, hypothèse de récurrence

```
let rec f = ··· f ···
```

Exemple : factorielle

- Cas de base : $0! = 1$
- Hyp. $n! = v$

$\rightsquigarrow (n+1)! = (n+1) \times v$

```
let rec fact = fun n ->
  if n = 0 then 1 else
    let hyp = (fact (n-1)) in
      n * hyp
```


Noyau fonctionnel — récursion

Fonction récursive : cas de base, hypothèse de récurrence

```
let rec f = ··· f ···
```

Exemple : factorielle

- Cas de base : $0! = 1$
- Hyp. $n! = v$

$\rightsquigarrow (n+1)! = (n+1) \times v$

```
let rec fact = fun n ->
  if n = 0 then 1 else
    n * (fact (n-1))
```

Récursion non terminale : empilement non borné de $*$ $\rightsquigarrow$ risques

Noyau fonctionnel — récursion

Exemple : factorielle

Propriété plus générale : $\forall a, \forall n, (f\ a\ n) = a \times n!$

- Cas de base : $\forall a, (f\ a\ 0) = a$
- Hyp. $\forall a, (f\ a\ n) = a \times n!$

$\rightsquigarrow (f\ a\ (n+1)) = (f(a \times (n+1))\ n) = a \times (n+1) \times n! = a \times (n+1)!$

```
let rec fact_gen = fun acc -> fun n ->
  if n = 0 then acc else
    fact_gen (acc *  n) (n - 1)
```

Cas particulier

```
let rec fact = fact_gen 1;;
```

Récursion terminale : jamais d'empilement trop gros

Noyau fonctionnel — induction

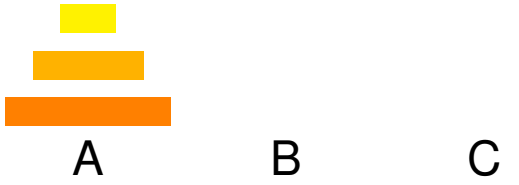
- Cas de base = constructeurs de base
- Suivant = Constructeur sur hyp. rec.

Hypothèse : on sait faire sur un constituant structurel

Distinction des cas : filtrage

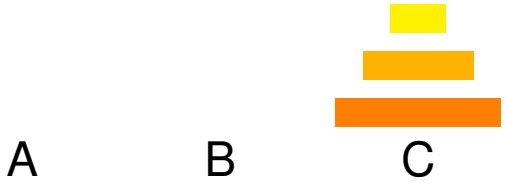
Noyau fonctionnel — illustration

Situation de départ



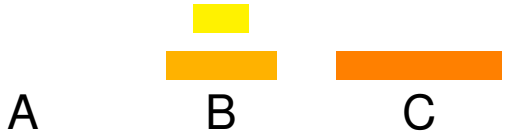
Noyau fonctionnel — illustration

Situation d'arrivée, un disque à la fois



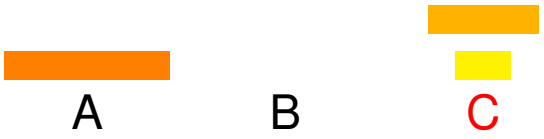
Noyau fonctionnel — illustration

Jamais de gros sur un petit : OK



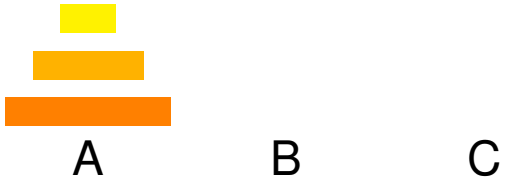
Noyau fonctionnel — illustration

Jamais de gros sur un petit : interdit



Noyau fonctionnel — illustration

Solution pour n disques ?



Noyau fonctionnel — principe de l'inférence

```
let rec f = fun g -> fun n -> fun x ->
  if n = 0 then x
  else f g (n-1) (g x)
```

Type de `f` ?

Listes — premiers pas

Structure dynamique — Mémoire

Soit vide, soit élément et liste qui suit $\rightsquigarrow$ passage au suivant

Structure — On veut persistante

- Vide
- Construction e en tête de l

Fonctions

- Ajout (en tête ou pas) — Retrait
- Appartenance
- Concaténation — Le mal

Listes — premiers pas

Structure dynamique — Mémoire

+ Ajout en tête en temps constant — Bien !
+ Ordre d'entrée conservé (FILO)
- Appartenance linéaire — Mal...
- Concaténation linéaire

Listes de quoi ? ⇝ polymorphisme

Listes — premiers pas

Type `'a list` natif dans Ocaml :

- Base `[]`
- Constructeur `::`

Utilisation directe dans filtrage

Notation alternative de la valeur :

`[a;b;⋯;k]`

Dans ce cours `a::b::⋯::k::[]`

Listes — itérateurs

Parcours de structure, construction de résultat portant sur structure

`map` liste d'applications de fonction par élément

`map` f $(a_1$`::`a_2`::⋯::`a_n`::[])` $\rightsquigarrow (f\ a_1)$`::`$(f\ a_2)$`::⋯::`$(f\ a_n)$`::[]`

Type : $(\alpha \to \beta) \to \alpha$`list` $\to \beta$`list`

Listes — itérateurs

Parcours de structure, construction de résultat portant sur structure

`fold` composition d'applications de fonction par élément

`fold_left` $f\ acc$ $(a_1$`::`a_2`::⋯::`a_n`::[])` $\rightsquigarrow f(\cdots(f(f\ acc\ a_1)\ a_2)\cdots)\ a_n$

Type : $(\alpha \to \beta \to \alpha) \to \alpha \to \beta$`list` $\to \alpha$ — Terminal

`fold_right` f $(a_1$`::`a_2`::⋯::`a_n`::[])` $acc \rightsquigarrow$
$f\ a_1\ (f\ a_2\ (\cdots(f\ a_n\ acc)\cdots))$

Type : $(\alpha \to \beta \to \beta) \to \alpha$`list` $\to \beta \to \beta$ — Non terminal

Noyau encore — exceptions

Opérations partielles : valeur exceptionnelle en cas d'erreur

```
#  1/0;;
Exception: Division_by_zero
```

. . . en cas de domaine de définition dépassé

```
Exception: Invalid_argument
```

. . . pour interrompre une évaluation

éventuellement la reprendre sur une autre expression !

Noyau encore — exceptions

Définition :

```
exception Toto

exception Terrible_erreur of string
```

(mais pas polymorphe)

Levée d'une exception :

```
raise Toto

raise (Terrible_erreur "subjonctif␣après␣«␣après␣que␣»")
```


Noyau encore — exceptions

Rattrapage: construction **`try ... with`**

`try` e_1 **`with`** `Motif_exc -> ` e_2

1. Évaluation de e_1, sans exception : calcul terminé — valeur e_1
2. Levée dans e_1 :
   - Filtrée par `Motif_exc` alors calcul poursuivi — valeur e_2
   - Non filtrée : exception propagée — ni e_1 ni e_2 évalués

Listes — travailler localement

Persistance : copies inutiles ?

⇝ accroche de la structure à la position de travail

Point de vue local ⇝ zipper (sur liste)

```
type 'a zipper = {
  left : 'a list ;
  pos : int ;
  right : 'a list ; }
```

Fonctions :

```
go_left
go_right
value
```

Listes — travailler localement

Persistance : copies inutiles ?

⇝ accroche de la structure à la position de travail

Point de vue local ⇝ zipper (sur liste)

```
type 'a zipper = {
  left : 'a list ;
  pos : int ;
  right : 'a list ; }
```

Fonctions :

```
go_left : 'a zipper -> 'a zipper
go_right : 'a zipper -> 'a zipper
value : 'a zipper -> 'a     pos : 'a zipper -> int
```


Listes — travailler localement

Persistance : copies inutiles ?

⇝ accroche de la structure à la position de travail

Point de vue local ⇝ zipper (sur liste)

```
type 'a zipper = Zip of ('a list * int * 'a list)
```

Organisation ⇝ constructeur futé

Fonctions :

```
go_left : 'a zipper -> 'a zipper
go_right : 'a zipper -> 'a zipper
value : 'a zipper -> 'a     pos : 'a zipper -> int
```


Ensembles — ABR

Structure dynamique encore

Aspects recherche et opérations ensemblistes

Organisation : relation d'ordre total — Savoir où chercher

Fonctions :

- Comparaison `compare : 'a -> 'a -> int`
- Ajout
- Appartenance
- $\#, \cup, \cap$

Ensembles — ABR

Comparaison x y :

- Positive si x strictement plus grand que y
- Nulle x est égal à y
- Négative si x strictement plus petit que y

Organisation :

- D'un côté : les plus petits
- De l'autre : les autres

⇝ Arbres binaires $\{21; 3; 42; 666; 73; 12; 37\}$?

Ensembles — ABR

Arbres d'int :

```
type abr = Empty | Node of (abr * int * abr)
```

Opérations de recherche

Opérations de construction

Coûts ?

Ensembles — ABR

Arbres de 'a :

```
compare : 'a → 'a → int
```

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

Opérations de recherche ⇝ bornées par profondeur — mieux que listes

Opérations de construction ⇝ fonction de la profondeur — moins bien

Coûts ? Réduire = réduire profondeur

À partir de maintenant :

```
let f x y z =···
```

pour

```
let f = fun x -> fun y -> fun z ->···
```


Ensembles — AVL

Réduire profondeur ⇝ équilibrage

Invariant supplémentaire : prof. à droite = prof. à gauche ±1

Information supplémentaire : profondeur

AVL de 'a : [Adelson-Velsky, Landis 62]

```
compare : 'a → 'a → int
```

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

```
height : 'a avl → int
```

Réduire profondeur $\rightsquigarrow$ équilibrage

Invariant supplémentaire : prof. à droite = prof. à gauche ± 1

Information supplémentaire : profondeur

Constructeur futé pour maintenir invariant

```
node : 'a avl → 'a → 'a avl → 'a avl
```

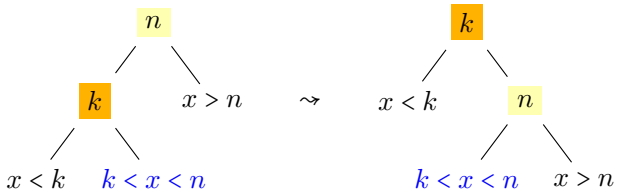

- Minimum ?
- Recherche ? bon marché !

$\rightsquigarrow$ comme ABR

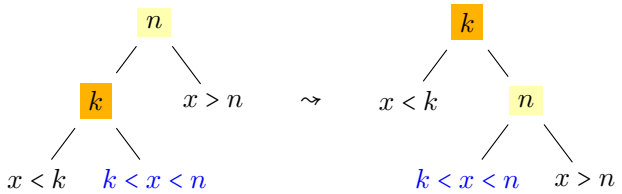
- Ajout ? (déséq. de 1)

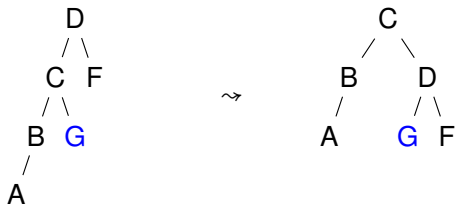
Invariant $\rightsquigarrow$ savoir équilibrer

Rotation droite : ⇜ Rotation gauche

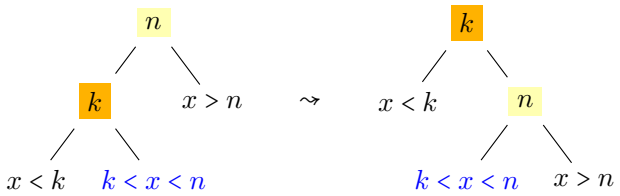


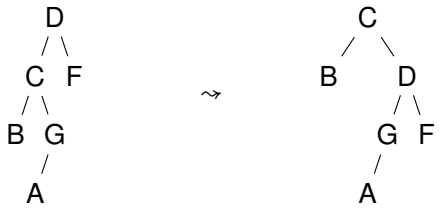
Rotation droite : ⇜ Rotation gauche





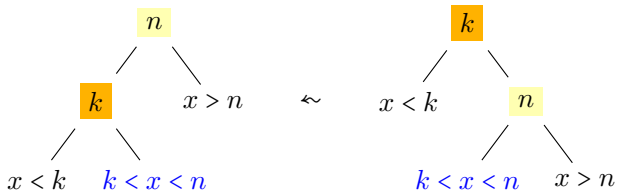
Rotation droite : ⇜ Rotation gauche





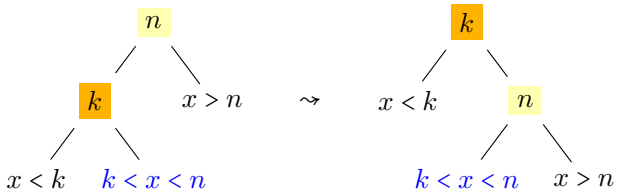
Échec

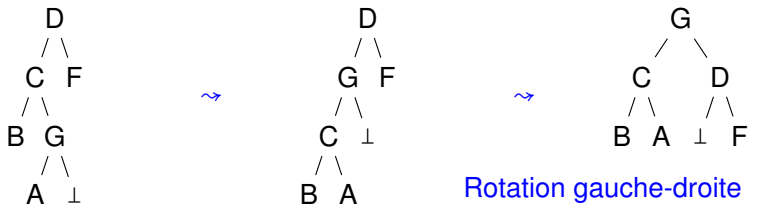
Rotation droite : ⇜ Rotation gauche



Échec

Rotation droite : ⇜ Rotation gauche

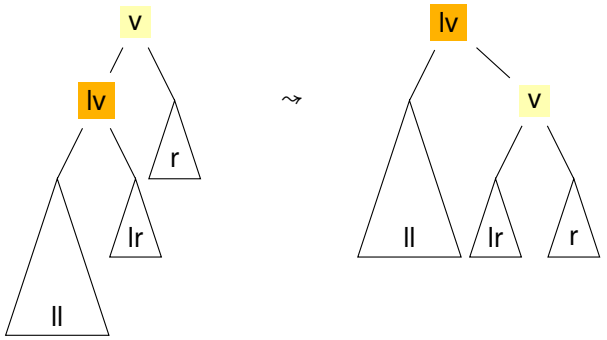




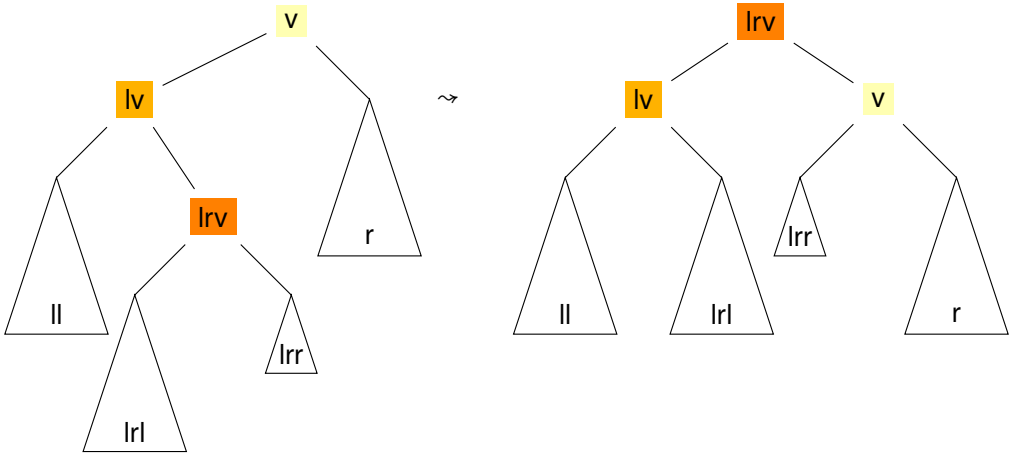
OK

Rotation gauche-droite

Simple : droite



Double : gauche-droite



Rotations simples et rotations doubles suffisantes

Garantir équilibrage à la construction : constructeur (encore plus) futé

```
balance : 'a avl → 'a → 'a avl → 'a avl
```

Par cas suivant le déséquilibre


```
let balance l v r =
  let hl = height l and hr = height r in
  if hl > hr + 1 then ( ··· )
```

`ll` plus grand que `lr` ⇝ Rot. droite suffit


```
let balance l v r =
  let hl = height l and hr = height r in
  if hl > hr + 1 then (match l with
    | Node(ll,lv,lr,_) →
      if (height ll ≥ height lr) then node ll lv (node lr v r)
```

`ll` plus grand que `lr` ⇝ Rot. droite suffit

```
let balance l v r =
  let hl = height l and hr = height r in
  if hl > hr + 1 then (match l with
    | Node(ll,lv,lr,_) →
      if (height ll ≥ height lr) then node ll lv (node lr v r)
```

`ll` plus petit que `lr` ⇝ gauche-droite


```
let balance l v r =
  let hl = height l and hr = height r in
  if hl > hr + 1 then (match l with
    | Node(ll,lv,lr,_) →
      if (height ll ≥ height lr) then node ll lv (node lr v r)
      else (match lr with
        | Node(lrl,lrv,lrr,_) →
            node (node ll lv lrl) lrv (node lrr v r)
```

`ll` plus petit que `lr` ⇝ gauche-droite


```
let balance l v r =
  let hl = height l and hr = height r in
  if hl > hr + 1 then (match l with
    | Node(ll,lv,lr,_) →
      if (height ll ≥ height lr) then node ll lv (node lr v r)
      else (match lr with
        | Node(lrl,lrv,lrr,_) →
            node (node ll lv lrl) lrv (node lrr v r)
```

Pas d'autre cas pour ce déséquilibre !


```
let balance l v r =
  let hl = height l and hr = height r in
  if hl > hr + 1 then (match l with
    | Node(ll,lv,lr,_) →
      if (height ll ≥ height lr) then node ll lv (node lr v r)
      else (match lr with
        | Node(lrl,lrv,lrr,_) →
            node (node ll lv lrl) lrv (node lrr v r)
        | _ → raise Les_maths_sont_inconsistantes)
    | _ → raise Les_maths_sont_inconsistantes)
```

Pas d'autre cas pour ce déséquilibre !

Ensembles — AVL

```
let balance l v r =
  let hl = height l and hr = height r in
  if hl > hr + 1 then (match l with
    | Node(ll,lv,lr,_) →
      if (height ll ≥ height lr) then node ll lv (node lr v r)
      else (match lr with
        | Node(lrl,lrv,lrr,_) →
            node (node ll lv lrl) lrv (node lrr v r)
        | _ → raise Les_maths_sont_inconsistantes)
    | _ → raise Les_maths_sont_inconsistantes)
  else if hr > hl + 1 then  ...
```

Cas de déséquilibre à droite symétrique

Ensembles — AVL

```
let balance l v r =
  let hl = height l and hr = height r in
  if hl > hr + 1 then (match l with
    | Node(ll,lv,lr,_) →
      if (height ll ≥ height lr) then node ll lv (node lr v r)
      else (match lr with
        | Node(lrl,lrv,lrr,_) →
            node (node ll lv lrl) lrv (node lrr v r)
        | _ → raise Les_maths_sont_inconsistantes)
    | _ → raise Les_maths_sont_inconsistantes)
  else if hr > hl + 1 then  ...
  else node l v r
```

Enfin cas sans déséquilibre

Ensembles — AVL

- Recherche — Efficace
- Ajout — Léger surcoût
- Retrait
- ...

Associations

On veut : $x \mapsto E_x$ — Efficacement

Trouver rapidement E_x étant donné x

Opération la plus courante : recherche

Ranger les E_x en fonction des x

Recherche efficace $\rightsquigarrow$ Base AVL

Associations

```
type key                                  (* Type des x *)
type 'a pmap                          (* 'a type des E_x *)
empty : 'a pmap
add : key → 'a → 'a pmap → 'a pmap
mem : key → 'a pmap → bool
find : key → 'a pmap → 'a
remove : key → 'a pmap → 'a pmap
```


Associations

```
type ('a,'b) pmap (* 'a type des clefs, 'b type des E_x *)
comp : 'a -> 'a -> int       (* comparaison sur clefs *)
empty : ('a,'b) pmap
add : 'a → 'b → ('a,'b) pmap → ('a,'b) pmap
mem : 'a → ('a,'b) pmap → bool
find : 'a → ('a,'b) pmap → 'b        (* évt Not_found*)
remove : 'a → ('a,'b) pmap → ('a,'b) pmap
```


Associations

Informations nécessaires : clef et élément lié (binding)

AVL construit sur clefs $\rightsquigarrow$ information supplémentaire : élément lié

```
type ('a,'b) pmap =
  | EM
  | NM of (('a,'b) pmap * 'a * 'b * ('a,'b) pmap * int);;

comp : 'a -> 'a -> int
empty : ('a,'b) pmap
height : ('a,'b) pmap → int

add : 'a → 'b →('a,'b) pmap → ('a,'b) pmap (*overriding*)
mem : 'a → ('a,'b) pmap → bool
```


Associations

Lecture de l'association : `find`

Données : clef x et association m

Valeur de `find` x m :

- E_x si associé à x dans m
- Sinon exception levée : `Not_found`

```
find : 'a → ('a,'b) pmap → 'b        (* évt Not_found*)
```

Associations — graphes

Application $N \to \mathcal{P}(N)$

Recherche d'un nœud, de successeurs

```
type 'a graph

add_vertex : 'a → 'a graph → 'a graph

add_arc : 'a → 'a → 'a graph → 'a graph
```

Graphe comme association :

```
type 'a graph = ('a, 'a avl) pmap
```


Associations — graphes, parcours

Trouver un chemin : bonne fondation ?

On peut :

- Parcourir le graphe en profondeur
- Remplir les déjà vus
- Interrompre en cas de succès (ou d'échec)

Ensembles — Données chaînées

Données composites : listes, etc.

Comparaison : coûteux (pas atomique)

Mémoire : coûteux (redondance)

$\rightsquigarrow$ Recherche : dommage

Ensembles — Données chaînées

Données composites : listes, etc.

Comparaison : pas à pas jusqu'à différence

Mémoire : débuts communs $\rightsquigarrow$ partage

$\rightsquigarrow$ Recherche : bornée par longueur du plus grand élément

Partage des préfixes : arbres de préfixes trie

Ensembles — Arbres de préfixes

- Nœuds : `bool`
- Branches : étiquetées par composants d'éléments

Élem. contenu si `bool` final = **`true`**

Invariant : bien formé

Invariant ⇒ recherche bornée par longueur du plus grand
QUEL QUE SOIT le nombre d'éléments !

```
type 'a trie = T of (bool * ('a, 'a trie) pmap)
mem : 'a list → 'a trie → bool
add : 'a list → 'a trie → 'a trie
remove : 'a list → 'a trie → 'a trie
inter : 'a trie → 'a trie → 'a trie
```

`map`, `fold`, etc.

Ensembles — Arbres de préfixes

Toujours plus loin...

`bool` ou autre

⇝ Associations

Si **`false`**, que faire ?

⇝ Type option

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