Programmation fonctionnelle avec OCaml

2ème séance, 26 février 2015

List	Liste
Tuple	N-uplet
Record	Enregistrement
Variant	Somme

samuel.hornus@inria.fr
http ://www.loria.fr/~shornus/ocaml/

Rappels

Toute *expression* a un *type* et une *valeur*.

Liaison d'un nom à une valeur :

```
let     nom_1 = expr_1 and ...
    and nom_k = expr_k [in expr]
```

Filtrage par motifs :

```
match valeur with
    | motif_1 -> expr_1
    | motif_2 -> expr_2 | ...
```

Fonctions :

```
[let nom =] function arg -> expr
[let nom =] fun a b c -> expr
let nom a b c = expr
```

Rappels

Pour être parfaitement précis, la syntaxe `exacte` pour un fonction est :

```
function filtrage-par-motif
```

```
let nom = function
      motif_1 -> expr_1
    | motif_2 -> expr_2
    | motif_3 -> expr_3
    ...
```

Cas particulier `à un seul motif` :

```
let nom = function x -> expr
```

Les listes

```
# ["bouleau"; "platane"; "chêne"];;
- : string list = ["bouleau"; "platane"; "chêne"]

# [45;2325;67;890];;
- : int list = [45; 2325; 67; 890]

# [['a';'e']; ['i';'o']; ['u';'y']; ['c']];;
- : char list list = [['a'; 'e']; ['i'; 'o']; ['u'; 'y'];
['c']]

# [1;'a'];;
Error: This expression has type char but an expression was
expected of type int

# [];;
- : 'a list = []
```

Les listes : tête et queue

```
# let l = 10::[20;22;56];;
val l : int list = [10; 20; 22; 56]

# let a = 42 in a::l;;
- : int list = [42; 10; 20; 22; 56]

# let c = [4] and d = [5] in c::d::[[6;7];[]];;
- : int list list = [[4]; [5]; [6;7];[]]
```

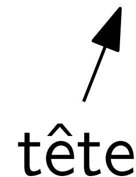
Les listes : tête et queue

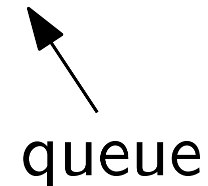
```
# let l = 10::[20;22;56];;
val l : int list = [10; 20; 22; 56]

# let a = 42 in a::l;;
- : int list = [42; 10; 20; 22; 56]

# let c = [4] and d = [5] in c::d::[[6;7];[]];;
- : int list list = [[4]; [5]; [6;7];[]]
```

L'opérateur `t::q` place l'élément `t` en tête de la liste `q` qui devient la queue de la nouvelle liste : `'a :: 'a list`





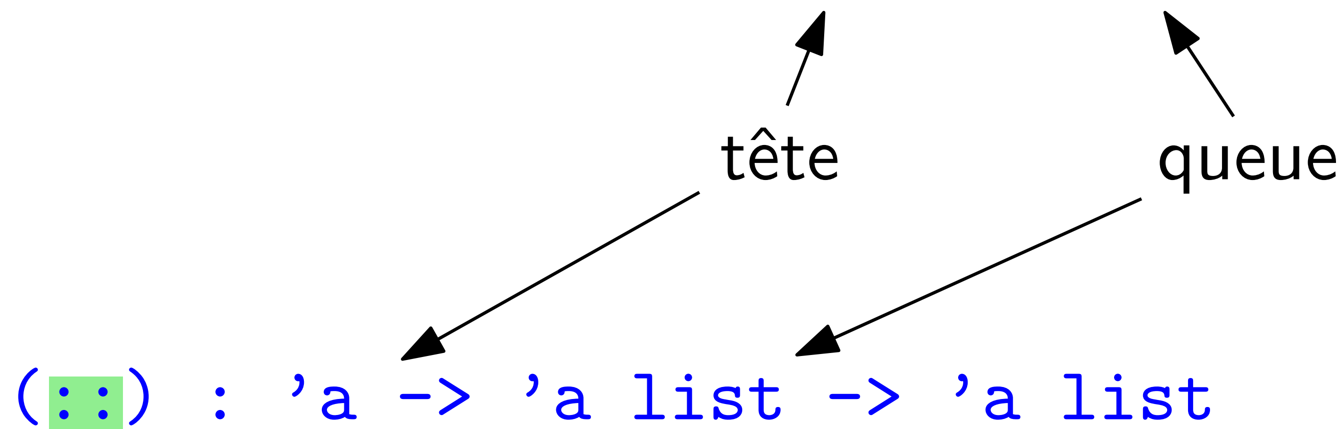
Les listes : tête et queue

```
# let l = 10::[20;22;56];;
val l : int list = [10; 20; 22; 56]

# let a = 42 in a::l;;
- : int list = [42; 10; 20; 22; 56]

# let c = [4] and d = [5] in c::d::[[6;7];[]];;
- : int list list = [[4]; [5]; [6;7];[]]
```

L'opérateur `t::q` place l'élément `t` en tête de la liste `q` qui devient la queue de la nouvelle liste : `'a :: 'a list`



Motifs pour filtrer une liste

Une fonction qui renvoie le deuxième élément d'une liste :

```
# let deuxiemeElt l = match l with
    | e1::e2::reste -> e2
    | _ -> failwith "liste trop courte";;
val deuxieme_elt : 'a list -> 'a = <fun>
```

Motifs pour filtrer une liste

Une fonction qui renvoie le deuxième élément d'une liste :

```
# let deuxiemeElt l = match l with
    | e1::e2::reste -> e2
    | _ -> failwith "liste trop courte";;
val deuxieme_elt : 'a list -> 'a = <fun>
```

Dans ce *filtre*, les noms `e1` et `reste` sont inutiles :

```
# let deuxiemeElt = function
    | _::e2::_ -> e2
    | _-> failwith "liste trop courte";;
val deuxieme_elt : 'a list -> 'a = <fun>
```

Motifs pour filtrer une liste

Une fonction qui renvoie le deuxième élément d'une liste :

```
# let deuxiemeElt l = match l with
    | e1::e2::reste -> e2
    | _ -> failwith "liste trop courte";;
val deuxieme_elt : 'a list -> 'a = <fun>
```

Dans ce *filtre*, les noms `e1` et `reste` sont inutiles :

```
# let deuxiemeElt = function
    | _::e2::_ -> e2
    | _-> failwith "liste trop courte";;
val deuxieme_elt : 'a list -> 'a = <fun>
```

```
# let v = ["Ceres"; "Vesta"; "Ixion"; "Pallas"]
  in deuxiemeElt v;;
- : string = "Vesta"
```

Les listes : exemple : trouver le plus petit élément

Trouver le plus petit élément d'une liste :

```
let rec trouveMin = function
    | []   -> failwith "liste vide"
    | [x]  -> x
    | x::r -> min (trouveMin r) x
val trouveMin : 'a list -> 'a = <fun>
```

Les listes : exemple : trouver le plus petit élément

Trouver le plus petit élément d'une liste :

```
let rec trouveMin = function
    | []   -> failwith "liste vide"
    | [x]  -> x
    | x::r -> min (trouveMin r) x
val trouveMin : 'a list -> 'a = <fun>
```

```
# trouveMin ['q';'w';'e';'r';'t';'y'];;
- : char = 'e'
# trouveMin [567;456;345;678;123;890];;
- : int = 123
# min;;
- : 'a -> 'a -> 'a = <fun>
```

Les listes : exemple : *search* / *replace*

Remplacer les occurences d'un élément dans une liste :

```
let rec remplace a b l = match l with
  | []   -> []
  | x::r -> (if x = a then b else x)::
               (remplace a b r)
```

```
val remplace : 'a -> 'a -> 'a list -> 'a list = <fun>
```

Les listes : exemple : *search* / *replace*

Remplacer les occurences d'un élément dans une liste :

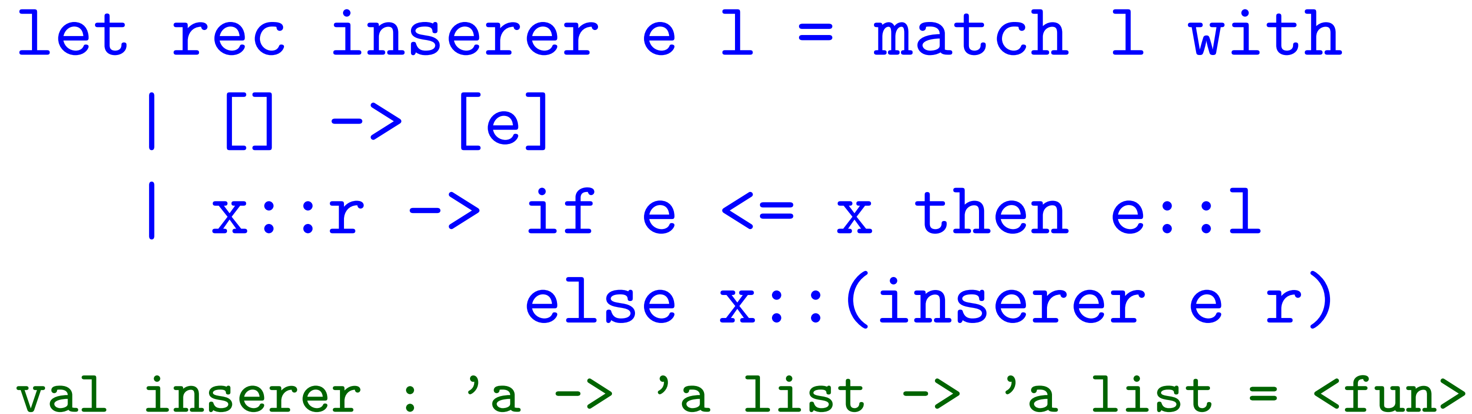
```
let rec remplace a b l = match l with
  | []    -> []
  | x::r -> (if x = a then b else x)::
                (remplace a b r)
val remplace : 'a -> 'a -> 'a list -> 'a list = <fun>
```

```
# remplace 10 11 [19;17;10;480;-50;-10;10;22];;
- : int list = [19;17;11;480;-50;-10;11;22]
```

```
# let phrase = ["Je"; "vis"; "dans"; "un"; "appart"]
  in remplace "un" "une"
      (remplace "appart" "maison" phrase);;
- : string list = ["Je"; "vis"; "dans"; "une"; "maison"]
```

Les listes : exemple : insertion dans une liste triée

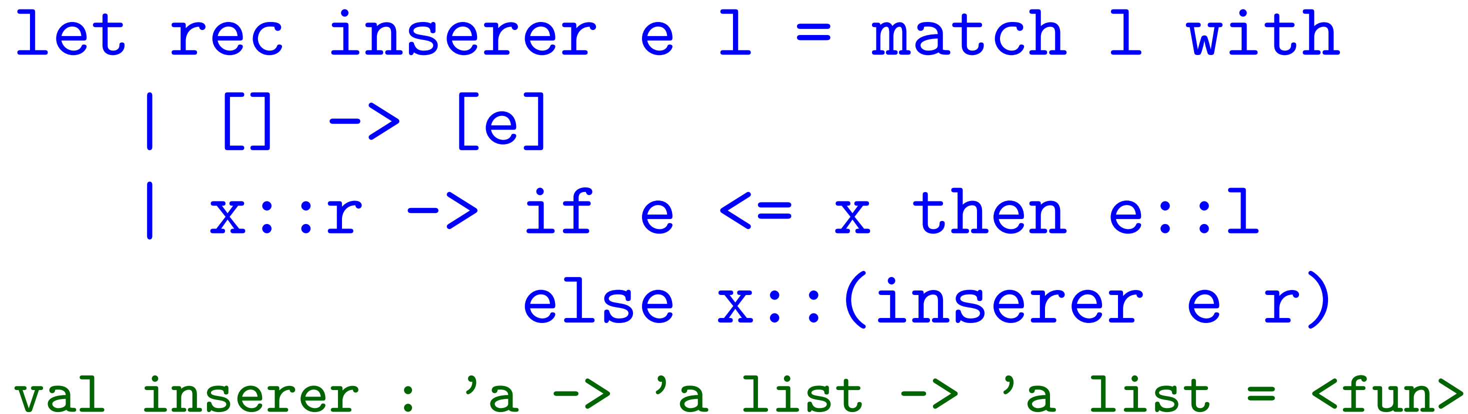
La liste `l` est déjà triée :



```
let rec inserer e l = match l with
    | [] -> [e]
    | x::r -> if e <= x then e::l
              else x::(inserer e r)
val inserer : 'a -> 'a list -> 'a list = <fun>
```

Les listes : exemple : insertion dans une liste triée

La liste `l` est déjà triée :



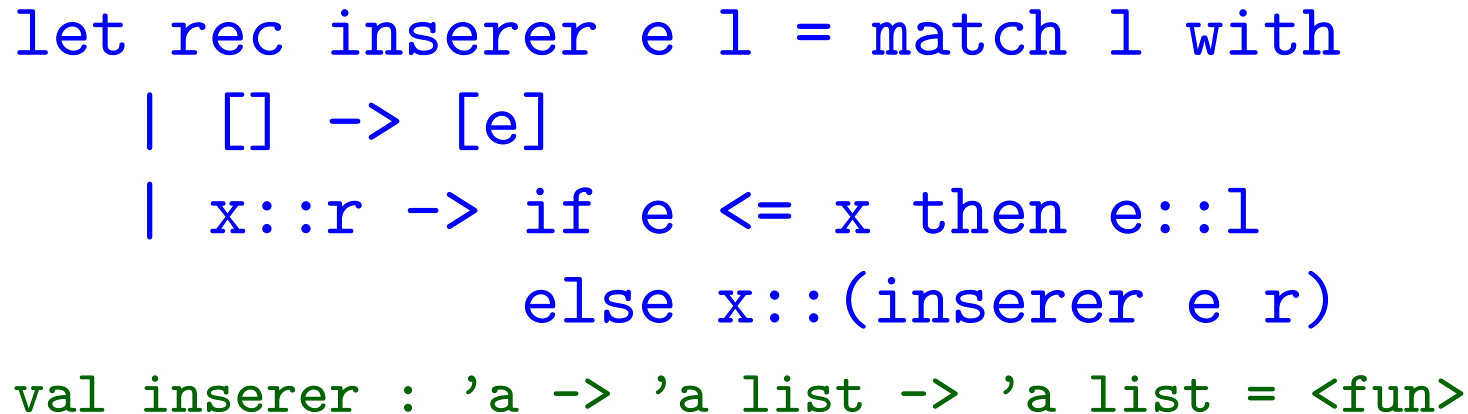
```
let rec inserer e l = match l with
    | [] -> [e]
    | x::r -> if e <= x then e::l
                      else x::(inserer e r)
val inserer : 'a -> 'a list -> 'a list = <fun>
```

Le « tri par insertion » (simple, mais peu efficace) :

```
let rec tri = function
    | [] -> []
    | x::r -> inserer x (tri r)
val tri : 'a list -> 'a list = <fun>
```

Les listes : exemple : insertion dans une liste triée

La liste `l` est déjà triée :



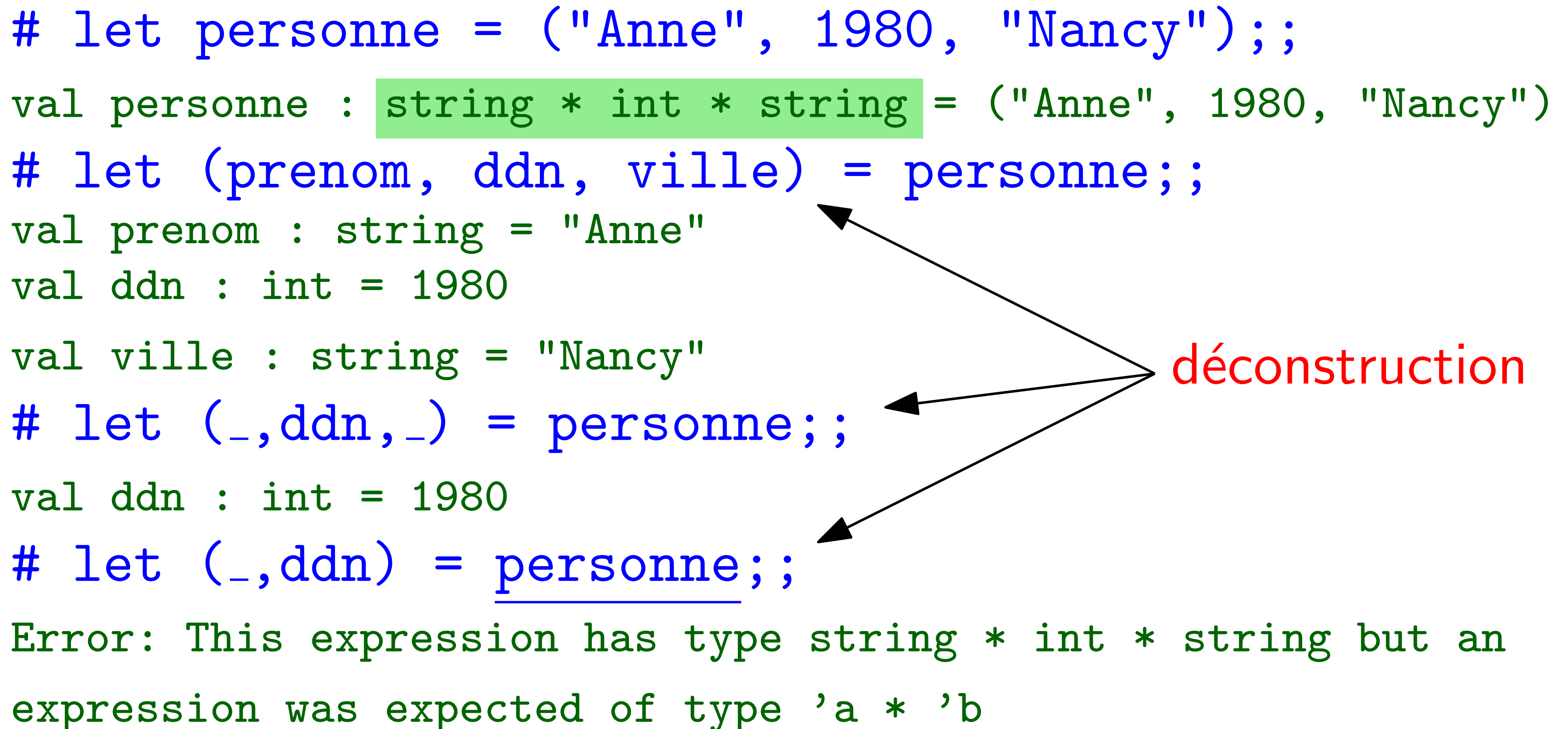
```
let rec inserer e l = match l with
    | [] -> [e]
    | x::r -> if e <= x then e::l
                        else x::(inserer e r)
val inserer : 'a -> 'a list -> 'a list = <fun>
```

Le « tri par insertion » (simple, mais peu efficace) :

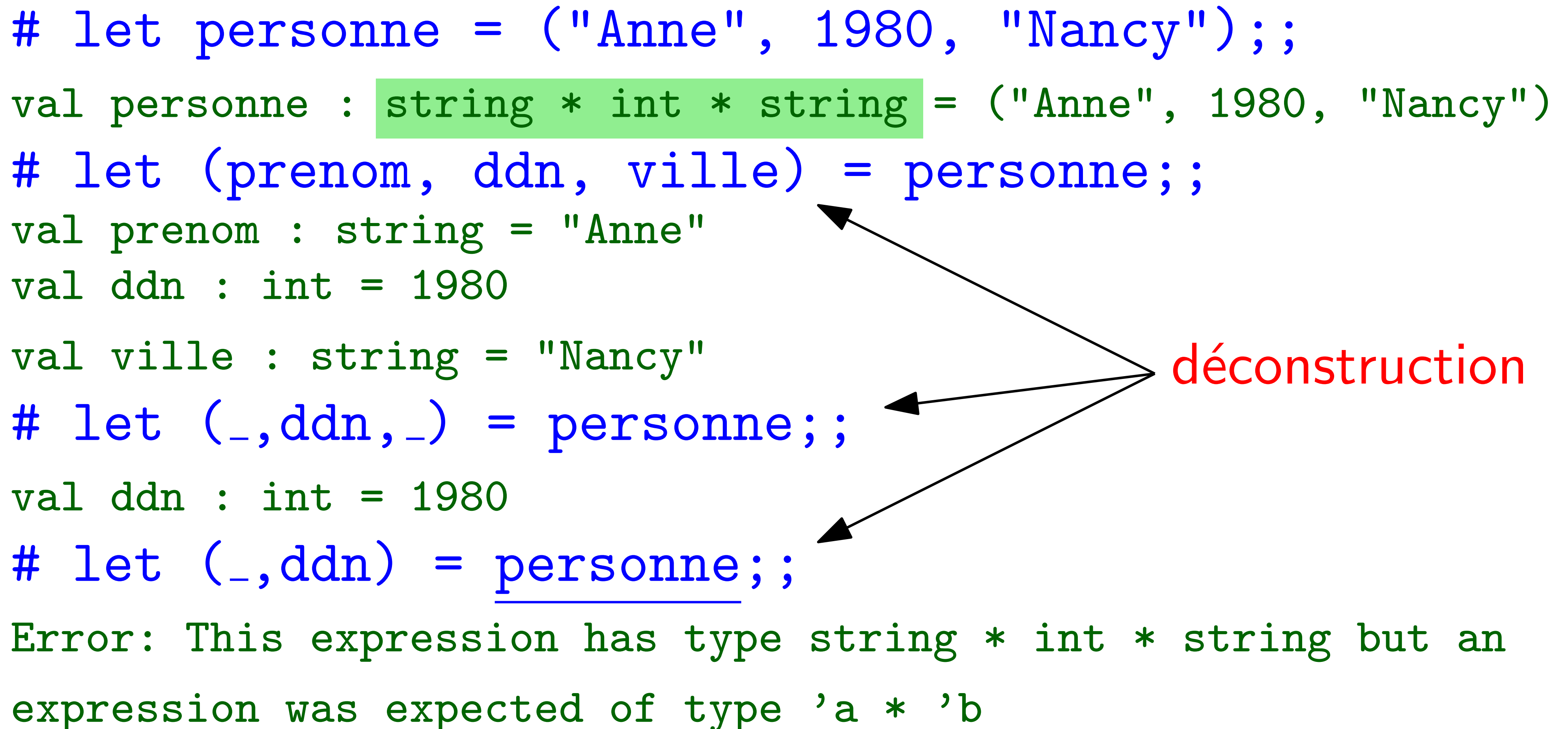
```
let rec tri = function
    | [] -> []
    | x::r -> inserer x (tri r)
val tri : 'a list -> 'a list = <fun>
```

```
# tri [9;0;8;1;7;2;6;3;5;4];;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Les n-uplets (*tuples*)



Les n-uplets (*tuples*)



Les éléments d'un n-uplet peuvent avoir des types différents, alors que tous les éléments d'une liste ont le même type.

On peut manipuler des listes de taille quelconque (grâce à `::`)
On ne peut manipuler des n-uplets que de taille raisonnable.

Appartée sur les liaisons nom $\longleftrightarrow$ valeur

```
let personne = ("Anne", 1980, "Nancy");;
```

```
let (prenom, ddn, ville) = personne;;
```

En fait, la syntaxe exacte est :

let *motif* = *expr*

Exemple

```
# let a::_::c = [1;2;3;4;5;6];;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val a : int = 1
val c : int list = [3; 4; 5; 6]
```

Exemple : représenter un point du plan

On représente un point du plan par une paire `float * float`.

Exemple : représenter un point du plan

On représente un point du plan par une paire `float * float`.

```
# let translate delta p = match (p,delta) with
    | ((x,y),(dx,dy)) -> (x +. dx,y +. dy);;
val translate : float * float -> float * float -> float * float
```

Exemple : représenter un point du plan

On représente un point du plan par une paire `float * float`.

```
# let translate delta p = match (p,delta) with
   | ((x,y),(dx,dy)) -> (x +. dx,y +. dy);;
val translate : float * float -> float * float -> float * float
```

Plus simplement :

```
# let translate (dx, dy) (x, y) =
                                (x +. dx, y +. dy);;
val translate : float * float -> float * float -> float * float
```

Exemple : représenter un point du plan

On représente un point du plan par une paire `float * float`.

```
# let translate delta p = match (p,delta) with
   | ((x,y),(dx,dy)) -> (x +. dx,y +. dy);;
val translate : float * float -> float * float -> float * float
```

Plus simplement :

```
# let translate (dx, dy) (x, y) =
                              (x +. dx, y +. dy);;
val translate : float * float -> float * float -> float * float
```

Exemple

```
# let up = translate (0.0,1.0);;
val up : float * float -> float * float = <fun>
# up (23.3, 12.0);;
- : float * float = (23.3, 13.)
```

Points du plan (suite)

On représente un point du plan par une paire `float * float`.

```
# let produit (x,y) (a,b) = x *. a +. y *. b;;
val produit : float * float -> float * float -> float

# let norme p = sqrt (produit p p);;
val norme : float * float -> float

# produit (0.0, 1.0) (4.0, 5.0);;
- : float = 5.0

# norme (3.,4.);;
- : float = 5.0
```

Exemple 2 : division entière

On peut utiliser un n-uplet pour `renvoyer plusieurs valeurs`.
Ici : le quotient et le reste d'une division entière.

Exemple 2 : division entière

On peut utiliser un n-uplet pour renvoyer plusieurs valeurs.
Ici : le quotient et le reste d'une division entière.

```
# let rec div a b =
    if a<0 || b <=0 then invalid_arg "div" else
    if a < b then (0,a)
    else let (q,r) = div (a-b) b in
         (q+1, r);;
val div : int -> int -> int * int = <fun>
```

Exemple 2 : division entière

On peut utiliser un n-uplet pour `renvoyer plusieurs valeurs`.
Ici : le quotient et le reste d'une division entière.

```
# let rec div a b =
   if a<0 || b <=0 then invalid_arg "div" else
   if a < b then (0,a)
   else let (q,r) = div (a-b) b in
        (q+1, r);;
val div : int -> int -> int * int = <fun>

# div 101 10;;
- : int * int = (10, 1)

# div 7 23;;
- : int * int = (0, 7)
```

Les enregistrements (*records*)

Interlude

On peut déclarer un nouveau type comme suit :

`type` *[paramètres] nom* `=` *nouveau_type* `;;`

ce qui correspond à peu près, en C, à :

`typedef` *nouveau_type nom* `;`

Exemples simples :

```
# type entier = int;;
type entier = int
```

```
# type date = int * string * int;;
type date = int * string * int
```

Les enregistrements (*records*)

Nous allons voir maintenant deux nouvelles manières de construire des types qui nécéssitent l'utilisation d'une définition de type (`type ... = ...`) :

1. les enregistrements
2. les sommes

Les enregistrements (*records*)

Les enregistrements sont des n-uplets plus flexibles :

chaque élément est nommé.

```
# type date = {année: int; mois: int; jour: int};;
type date =  année : int; mois : int; jour : int;

type nom = {prenom: string; nomFamille: string}

type étudiant = {nom: nom; ddn: date; promo: int}
```

Les enregistrements : création

```
# {annee = 2011; mois = 3; jour = 3};;
- : date = {annee = 2011; mois = 3; jour = 3}

# let makeDate a m j = {jour=j; annee=a; mois=m};;
val makeDate : int -> int -> int -> date = <fun>

# let d1=makeDate 1999 6 24;;
val d1 : date = {annee = 1999; mois = 6; jour = 24}

# let n = {prenom="Samuel"; nomFamille="Hornus"};;
val n : nom = {prenom="Samuel"; nomFamille="Hornus"}

# let e = {nom=n; ddn=d1; promo=2022};;
val e : etudiant = {nom = {prenom = "Samuel"; nomFamille =
"Hornus"}; ddn = {annee = 1999; mois = 12; jour = 31}; promo =
2022}
```

Les enregistrements : accès aux champs

```
# n.prenom;;
- : string = "Samuel"

# e.ddn.jour;;
- : int = 31

# let age_sortie e = e.promo - e.ddn.annee;;
val age_sortie : etudiant -> int = <fun>

# age_sortie e;;
- : int = 23
```

Les enregistrements : filtrage par motifs

```
# let isNoël d = match d with
    | {jour=25; mois=12} -> true
    | _ -> false;;
val isNoël : date -> bool
```

(exemple du livre *Apprentissage de la programmation avec OCaml*, Hermès-Science.)

Les enregistrements : filtrage par motifs

```
# let isNoël d = match d with
    | {jour=25; mois=12} -> true
    | _ -> false;;
val isNoël : date -> bool
```

(exemple du livre *Apprentissage de la programmation avec OCaml*, Hermès-Science.)

```
let rec jeune = function
    [] -> []
    | {nom=n; ddn={annee=a}; promo=p}::r
      when (p-a <= 21) -> n::(jeune r)
    | _::r -> (jeune r);;
val jeune : etudiant list -> nom list = <fun>
```

Les types somme (*variants*)

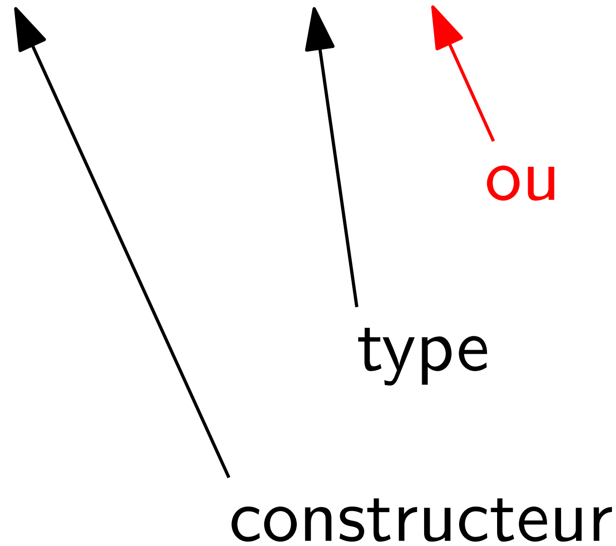
En OCaml :

```
type data = Entier of int | Car of char;
```

Les types somme (*variants*)

En OCaml :

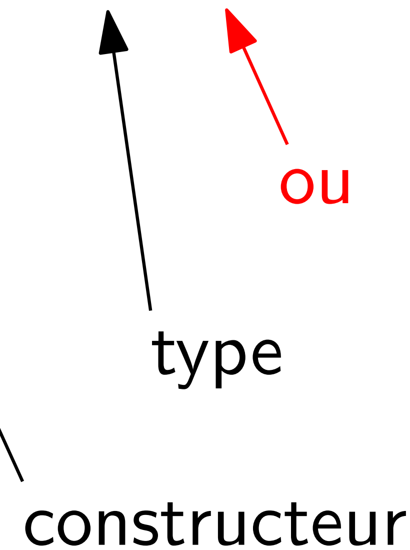
```
type data = Entier of int | Car of char;
```



Les types somme (*variants*)

En OCaml :

```
type data = Entier of int | Car of char;
```



Les types somme : syntaxe

```
type [param] nom = Constructeur_1 [of type_1]
                 | Constructeur_2 [of type_2]
                 | ...
                 | Constructeur_n [of type_n]
```

Les types somme : syntaxe

```
type [param] nom = Constructeur_1 [of type_1]
                 | Constructeur_2 [of type_2]
                 | ...
                 | Constructeur_n [of type_n]
```

Exemple 1 :

```
type nombre = Ent of int | Flot of float
```

```
# Ent 12;;
- : nombre = Ent 12
```

```
# Flot 12.0;;
- : nombre = Flot 12.
```

Les types somme : exemple 1 (suite)

```
# let ajoute a b = match a with
   | Ent(x)  -> (match b with
                  | Ent(y) -> Ent (x+y)
                  | Flot(y) -> Flot (float x +. y))
   | Flot(a) -> (match b with
                  | Ent(b) -> Flot (a +. float b)
                  | Flot(b) -> Flot (a+.b));;
val ajoute : nombre -> nombre -> nombre = <fun>
```

Les types somme : exemple 1 (suite)

```
# let ajoute a b = match a with
   | Ent(x)  -> (match b with
                  | Ent(y) -> Ent (x+y)
                  | Flot(y) -> Flot (float x +. y))
   | Flot(a) -> (match b with
                  | Ent(b) -> Flot (a +. float b)
                  | Flot(b) -> Flot (a+.b));;
val ajoute : nombre -> nombre -> nombre = <fun>

# ajoute (Ent 3) (Flot 5.4);;
- : nombre = Flot 8.4
```

Les types somme : exemple 2 : créer un type de liste

```
type 'a liste =   ListeVide
                | Element of 'a * 'a liste
```

```
# let l = Element('b', Element('r', Element('a',
ListeVide)));;
val l : char liste = Element ('b', Element ('r', Element ('a',
ListeVide)))
```

Les types somme : exemple 2 : créer un type de liste

```
type 'a liste =   ListeVide
                | Element of 'a * 'a liste

# let l = Element('b', Element('r', Element('a',
ListeVide)));;
val l : char liste = Element ('b', Element ('r', Element ('a',
ListeVide)))


let maxInListe l = match l with
  | ListeVide -> failwith "liste vide"
  | Element(x,ListeVide) -> x
  | Element(x, reste) -> max (maxInListe reste) x
val maxInListe : 'a liste -> 'a = <fun>
```

Les types somme : exemple 2 : créer un type de liste

```
type 'a liste =   ListeVide
                | Element of 'a * 'a liste
```

```
# let l = Element('b', Element('r', Element('a',
ListeVide)));;
val l : char liste = Element ('b', Element ('r', Element ('a',
ListeVide)))
```

```
let maxInListe l = match l with
  | ListeVide -> failwith "liste vide"
  | Element(x,ListeVide) -> x
  | Element(x, reste) -> max (maxInListe reste) x
val maxInListe : 'a liste -> 'a = <fun>
```

```
# maxInListe l;;
- : char = 'r'
```

Un cas particulier : les Enums

```
type couleur = Rouge | Orange | Jaune | Vert |
Cyan | Bleu | Violet | Noir | Blanc
```

Un cas particulier : les Enums

```
type couleur = Rouge | Orange | Jaune | Vert |
Cyan | Bleu | Violet | Noir | Blanc
```

Mélange des genres :

```
type activ_cnx_info = { start_time : int }

type cnx_status = Inactive
                | Active of activ_cnx_info
```

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

Les types somme : exemple 3 : expression arithmétique

```
type expr = | Valeur of int
            | Produit of expr * expr
            | Somme of expr * expr
```

Les types somme : exemple 3 : expression arithmétique

```
type expr = | Valeur of int
            | Produit of expr * expr
            | Somme of expr * expr
```

```
# let ex1 = Produit ( Valeur 2, Somme ( Valeur 3,
Valeur 4 ) );;
val ex1 : expr = Produit (Valeur 2, Somme (Valeur 3, Valeur
4))
```

Les types somme : exemple 3 : expression arithmétique

```
type expr = | Valeur of int
            | Produit of expr * expr
            | Somme of expr * expr
```

```
let rec eval = function
  | Valeur i -> i
  | Produit (e,f) -> eval e * eval f
  | Somme (e,f) -> eval e + eval f
val eval : expr -> int = <fun>
```

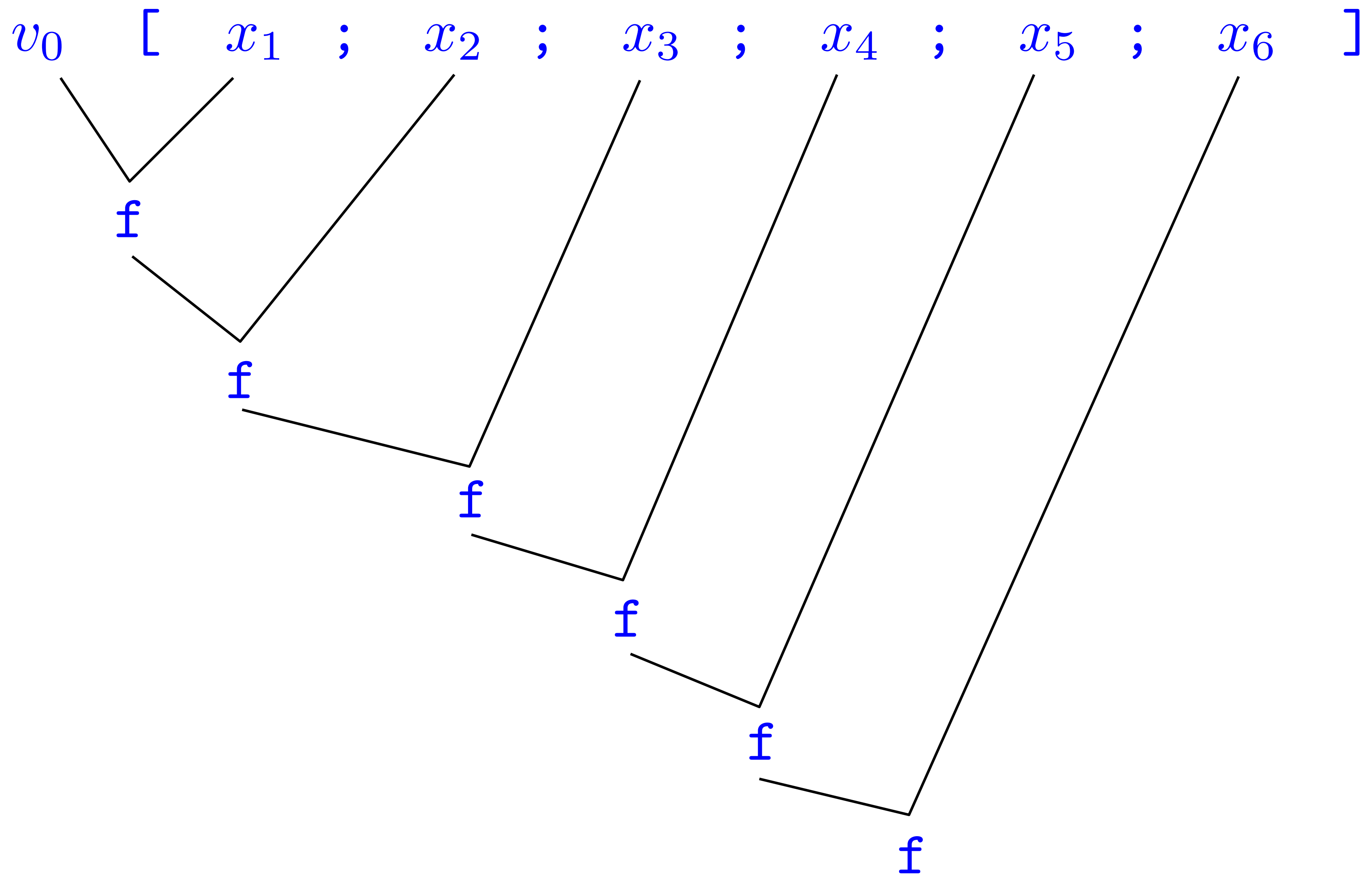
Les types somme : exemple 3 : expression arithmétique

```
type expr = | Valeur of int
            | Produit of expr * expr
            | Somme of expr * expr
```

```
let rec eval = function
  | Valeur i -> i
  | Produit (e,f) -> eval e * eval f
  | Somme (e,f) -> eval e + eval f
val eval : expr -> int = <fun>
```

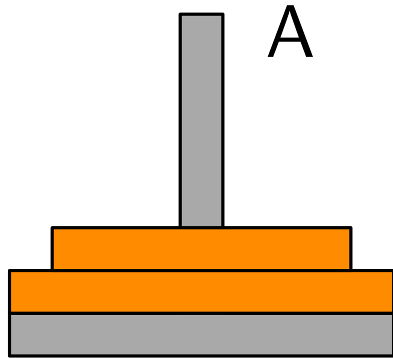
```
# eval ex1;;
- : int = 14
```

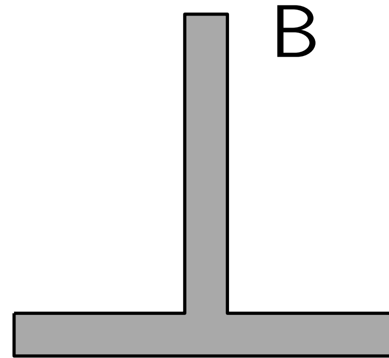
fold left

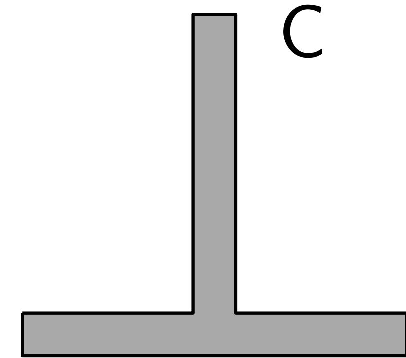


`fold_left f v0 l`

Les tours de Hanoï

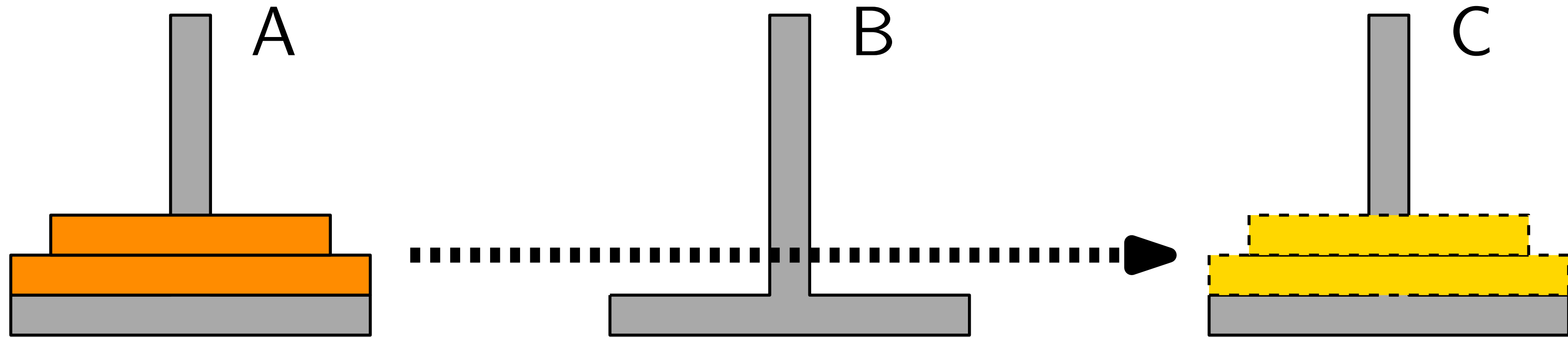






On ne peut déplacer qu'un disque en haut d'une pile.
On ne peut poser un disque que sur un disque plus grand.

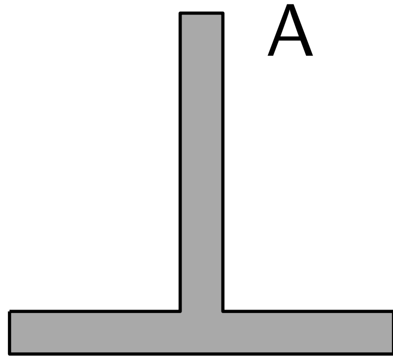
Les tours de Hanoï

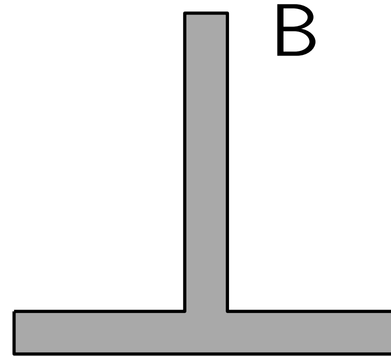


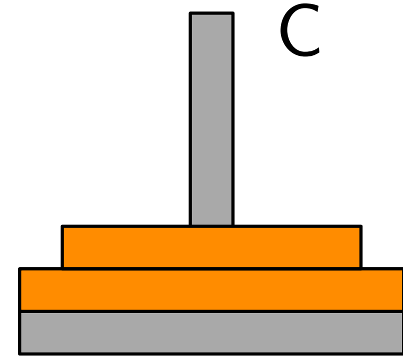
On ne peut déplacer qu'un disque en haut d'une pile.

On ne peut poser un disque que sur un disque plus grand.

Les tours de Hanoï

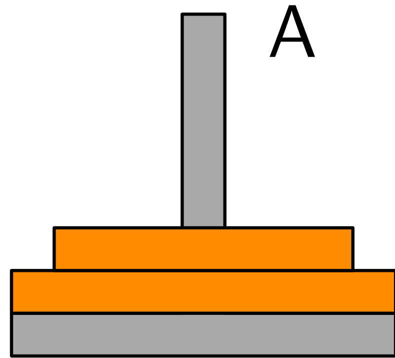


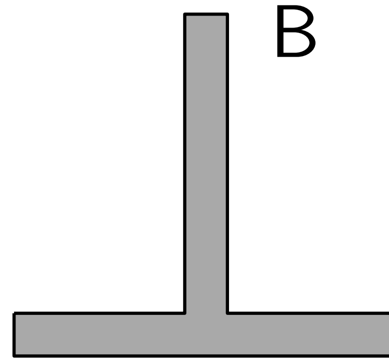


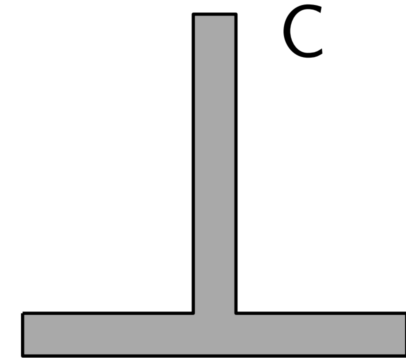


On ne peut déplacer qu'un disque en haut d'une pile.
On ne peut poser un disque que sur un disque plus grand.

Les tours de Hanoï



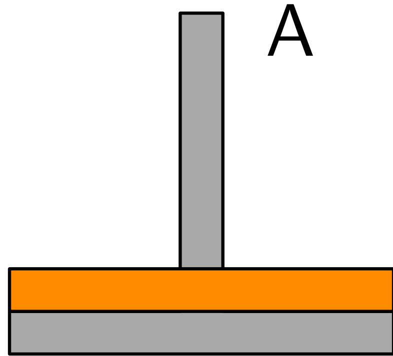


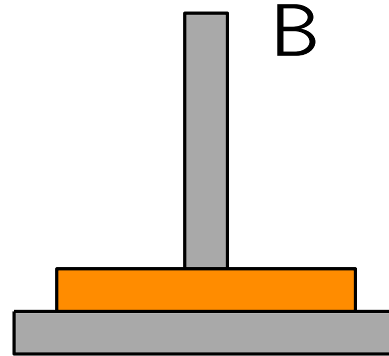


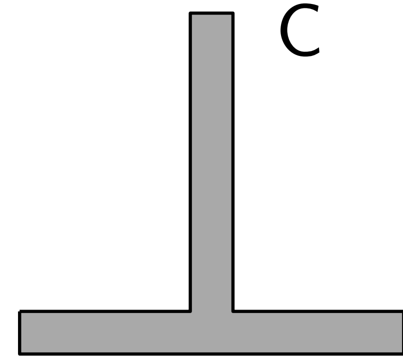
On ne peut déplacer qu'un disque en haut d'une pile.

On ne peut poser un disque que sur un disque plus grand.

Les tours de Hanoï



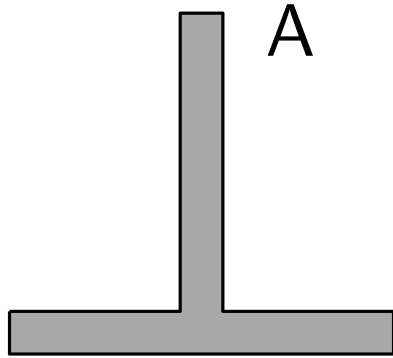


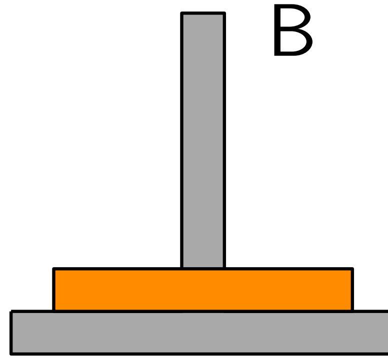


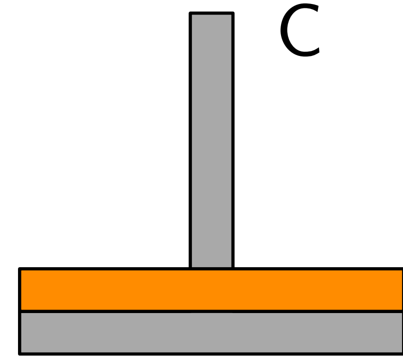
On ne peut déplacer qu'un disque en haut d'une pile.

On ne peut poser un disque que sur un disque plus grand.

Les tours de Hanoï



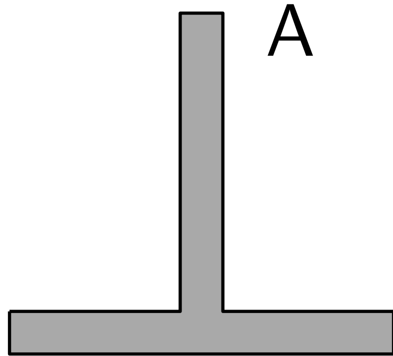


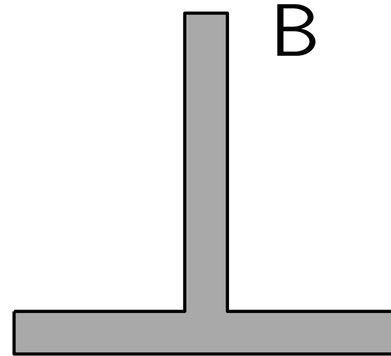


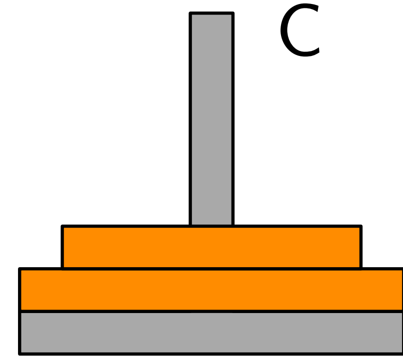
On ne peut déplacer qu'un disque en haut d'une pile.

On ne peut poser un disque que sur un disque plus grand.

Les tours de Hanoï



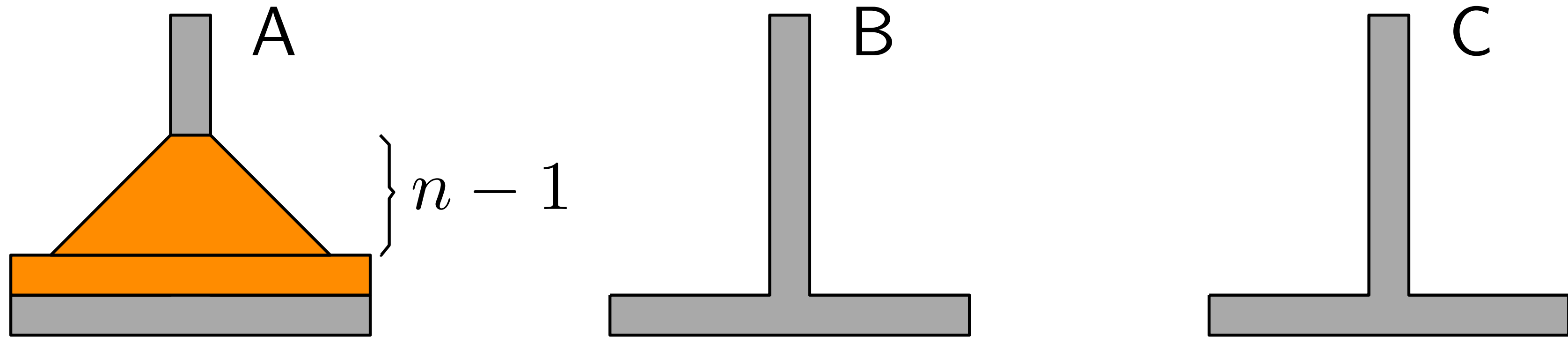




On ne peut déplacer qu'un disque en haut d'une pile.

On ne peut poser un disque que sur un disque plus grand.

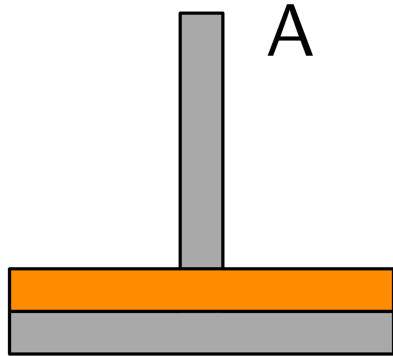
Les tours de Hanoï (illustration)

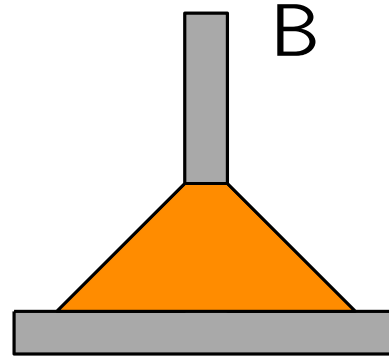


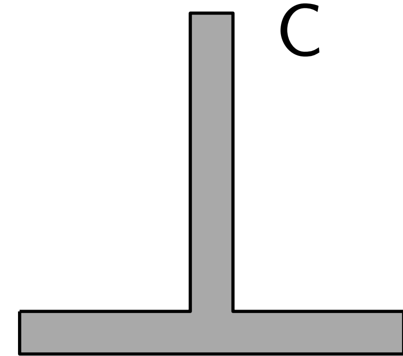
On ne peut déplacer qu'un disque en haut d'une pile.
On ne peut poser un disque que sur un disque plus grand.

```
let rec hanoi n départ inter arrivée = if n > 0
then begin
    hanoi (n-1) départ arrivée inter;
    Printf.printf "%c -> %c\n" départ arrivée;
    hanoi (n-1) inter départ arrivée
end;;
```

Les tours de Hanoï (illustration)



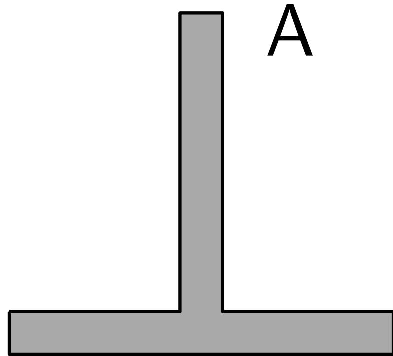


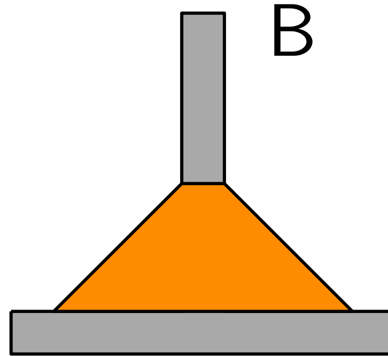


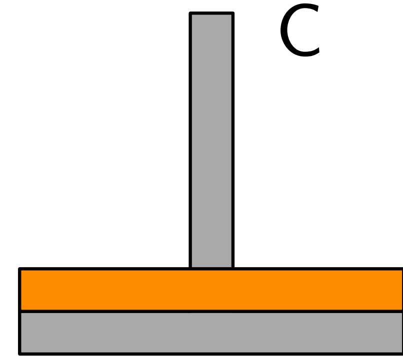
On ne peut déplacer qu'un disque en haut d'une pile.
On ne peut poser un disque que sur un disque plus grand.

```
let rec hanoi n départ inter arrivée = if n > 0
then begin
    hanoi (n-1) départ arrivée inter;
    Printf.printf "%c -> %c\n" départ arrivée;
    hanoi (n-1) inter départ arrivée
end;;
```

Les tours de Hanoï (illustration)



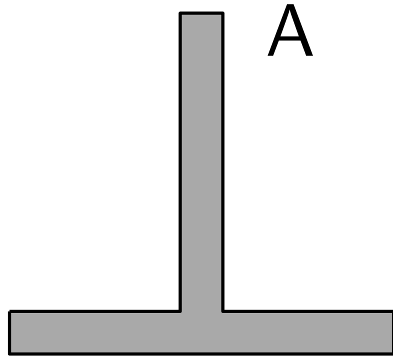


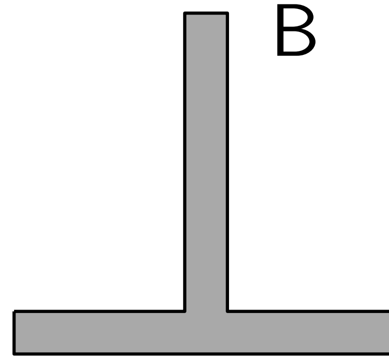


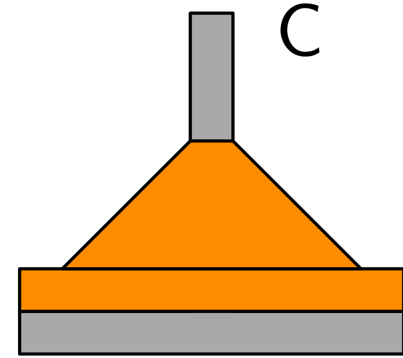
On ne peut déplacer qu'un disque en haut d'une pile.
On ne peut poser un disque que sur un disque plus grand.

```
let rec hanoi n départ inter arrivée = if n > 0
then begin
    hanoi (n-1) départ arrivée inter;
    Printf.printf "%c -> %c\n" départ arrivée;
    hanoi (n-1) inter départ arrivée
end;;
```

Les tours de Hanoï (illustration)



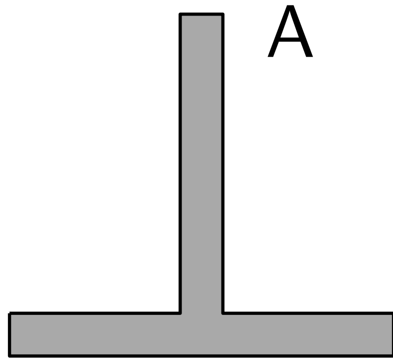


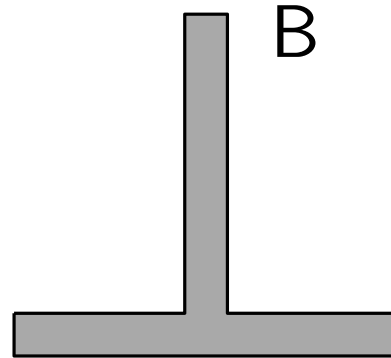


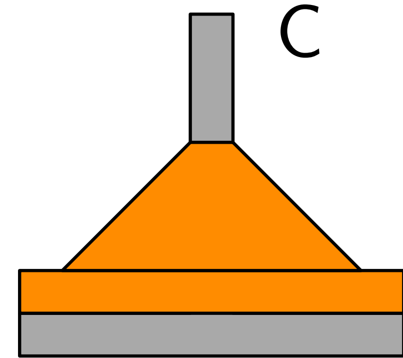
On ne peut déplacer qu'un disque en haut d'une pile.
On ne peut poser un disque que sur un disque plus grand.

```
let rec hanoi n départ inter arrivée = if n > 0
then begin
    hanoi (n-1) départ arrivée inter;
    Printf.printf "%c -> %c\n" départ arrivée;
    hanoi (n-1) inter départ arrivée
end;;
```

Les tours de Hanoï (illustration)







On ne peut déplacer qu'un disque en haut d'une pile.
On ne peut poser un disque que sur un disque plus grand.

```
let rec hanoi n départ inter arrivée = if n > 0
then begin
    hanoi (n-1) départ arrivée inter;
    Printf.printf "%c -> %c\n" départ arrivée;
    hanoi (n-1) inter départ arrivée
end;;
```

Quel est la complexité de cet algorithme ?