Programmation Fonctionnelle Avancée

Cours 2

kn@lri.fr
http://www.lri.fr/~kn

Résumé de l'épisode précédent

On a fait des rappels sur le langage OCaml

- Un fichier est une suite de définitions introduites par `let ... = ...`
- Les expressions simples peuvent contenir des valeurs entières, flottantes, des chaînes, des booléens et les opérations associées
- On peut former des types structurés :
  - n-uplets : qui permettent de grouper de l'information : `(1,2)`, `("Hello", false)`, …
  - enregistrements : même usage, mais permet de donner un nom symbolique aux composantes :

```
type time = { h : int; m: int; s: float }
let midnight = { h=0; m=0; s=0.0 }
```

  - types sommes : permet de définir plusieurs cas

```
type color = RGB of int * int * int
           | RGBa of int * int * int * float
           | Symbolic of string
let red = Symbolic "red"
let transp_yellow = RGBa (255, 255, 0, 0.5)
```

Résumé de l'épisode précédent (2)

Les structures de contrôles :

- `if … then … else …`
- Les fonctions (éventuellement récursives, prenant d'autres fonctions en argument, ...)

```
let rec iter_int f i j =
  if i < j then begin
    f i;
    iter_int (i+1) j
  end
let pr_int n = Printf.printf "-> %d\n" n
let () = iter_int pr_int 0 10
```

- le pattern-matching (filtrage) :

```
let pr_color c =
   match c with
    RGBa(r, g, b, alpha) -> Printf.printf "(%d, %d, %d, %f)" r g b a
    | Symbolic name -> Printf.printf "<%s>" name
```

Plan

1 PFA (1) : Rappels d'OCaml ✓

2 PFA (2) : Rappels d'OCaml (suite et fin)

- 2.1 Fonctions récursives terminales
- 2.2 Récursion mutuelle
- 2.3 Inférence de types
- 2.4 Polymorphisme
- 2.5 Listes
- 2.6 Fonctions anonymes
- 2.7 Itérateurs avancés
- 2.8 Applications partielles
- 2.9 Exceptions et gestion des erreurs

Récursion terminale

Considérons la fonction `fact_alt` :

```
let rec fact_alt n acc =
if n <= 1 then
 acc
else
 fact_alt (n - 1) (n * acc)
fact_alt 6 1
       fact_alt 5 6
           fact_alt 4 30
               fact_alt 3 120
                   fact_alt 2 360
                       fact_alt 1 720
                       720 ← cas de base
                   720
               720
           720
       720
 720
```

Récursion terminale (2)

La fonction `fact_alt` calcule son résultat « en descendant » :

- Elle utilise un argument auxiliaire `acc` dans lequel elle accumule les résultats partiels
- Lorsqu'on arrive au cas de base, le calcul est terminé
- Les « retours » d'appels récursifs ne font que propager le résultat

`fact_alt` est une fonction récursive terminale.

Une fonction est récursive terminale si tous les appels récursifs sont terminaux.

Un appel récursif est terminal si c'est la dernière instruction à s'exécuter.

Récursion terminale (3)

```
let rec fact_alt n acc =
if n <= 1 then
 acc
else
 fact_alt (n - 1) (n * acc)
```

```
let rec fact n =
if n <= 1 then
 1
else
 n * fact (n-1)
```

- Dans `fact_alt` l'appel récursif est terminal, c'est la dernière chose que l'on fait dans la fonction
- Dans `fact` l'appel récursif est non-terminal : il faut prendre la valeur qu'il renvoie et la multiplier par `n`. La multiplication s'effectue après l'appel récursif.

Le compilateur OCaml optimise les fonctions recursives terminales en les compilant comme des boucles, ce qui donne du code très efficace et qui consomme une quantité de mémoire bornée (et non pas une pile arbitrairement grande)

Fonction imbriquée

En OCaml, une fonction est une valeur comme une autre. On peut donc définir des fonctions locales à des expressions :

```
let x = 42

let x2 =
   let carre n = n * n
   in
      carre x
```

La fonction `carre` est une fonction locale à l'expression dans laquelle elle se trouve. La notation :

```
  let f x₁ … xₙ =
           e_f
  in
    e
```

permet de définir la fonction et de ne l'utiliser que pour l'expression `e`. La fonction `f` n'est pas visible à l'extérieur.

Fonction imbriquée

Une utilisation courante des fonctions locales est la définition de fonctions auxiliaires à l'intérieur d'une fonction principale :

```
let fact m =
  let rec fact_alt n acc =
    if n <= 1 then
     acc
    else
     fact_alt (n - 1) (n * acc)
  in
    fact_alt m 1
```

Ici, il est impossible d'appeler `fact_alt` depuis l'extérieur (et en particulier avec de mauvais paramètres). C'est une forme d'encapsulation.

Remarque : `fact` n'a pas besoin d'être récursive, il n'y a que `fact_alt` qui doit être déclarée avec « `let rec` ».

Plan

1 PFA (1) : Rappels d'OCaml ✓

2 PFA (2) : Rappels d'OCaml (suite et fin)

- 2.1 Fonctions récursives terminales ✓
- 2.2 Récursion mutuelle
- 2.3 Inférence de types
- 2.4 Polymorphisme
- 2.5 Listes
- 2.6 Fonctions anonymes
- 2.7 Itérateurs avancés
- 2.8 Applications partielles
- 2.9 Exceptions et gestion des erreurs

Récursion mutuelle

Les appels récursifs ne sont pas limités à une seule fonction. Il est courant de vouloir définir deux fonctions qui s'appellent l'une l'autre :

```
let rec pair n =
   if n == 0 then
      true
   else
     impair (n-1)
and impair n =
   if n == 0 then
      false
   else
      pair (n-1)
```

```
pair 6
    impair 5
          pair 4
              impair 3
                    pair 2
                        impair 1
                              pair 0
                              true  ← cas de base.
```

Remarque : ces deux fonctions sont récursives terminales !

On aura l'occasion de revoir les fonctions mutuellement récursives lorsqu'on travaillera sur des structures de données plus complexes que des entiers.

Plan

1 PFA (1) : Rappels d'OCaml ✓
2 PFA (2) : Rappels d'OCaml (suite et fin)
- 2.1 Fonctions récursives terminales ✓
- 2.2 Récursion mutuelle ✓
- 2.3 Inférence de types
- 2.4 Polymorphisme
- 2.5 Listes
- 2.6 Fonctions anonymes
- 2.7 Itérateurs avancés
- 2.8 Applications partielles
- 2.9 Exceptions et gestion des erreurs

Inférence de types

Le compilateur OCaml effectue une inférence de types :

- Il devine le type des variables
- Il en déduit le type des expressions
- Il en déduit le type des fonctions

```
# let x = 2 ;;
val x : int = 2
# let y = 3 ;;
val y : int = 3
# let f n = n + 1;;
val f : int -> int = <fun>
# let g x = sqrt (float x);;
val g : int -> float = <fun>
```

Inférence de types (2)

Le compilateur affecte à chaque variable de programme une variable de type.

Il pose ensuite des équations entre ces variables de types

Si des équations sont contradictoires, OCaml indique une erreur de type

```
let f n  =
   if n <= 1 then
     42
   else
     n + 45
```

α_n : type de n

α_f : type de retour de f

α_n = int (car n <= 1)

α_n = int (car n + 45)

α_f = int (car on renvoie 42)

α_f = int (car on n + 45)

n : α_n= int

f : α_n -> α_f = int -> int

```
let g n =
   if n <= 1 then
     42
   else
     "Boo!"
```

α_n : type de n

α_g : type de retour de g

α_n = int (car n <= 1)

α_g = int (car on renvoie 42)

α_g = string (car on renvoie "Boo!")

n : int ≠ string ⇒ erreur de typage

Type des fonctions

Soit une fonction :

```
let f x₁ … xₙ =
   e_f
```

Le type de `f` se note : t_1 `->` … `->` t_n `->` t_f où :

- t_i est le type du paramètre x_i (pour $1 \leq i \leq n$)
- t_f est le type de retour de la fonction

Types de fonctions (exemples)

```
let mult_add a b c =  a * b + c (* int -> int -> int -> int *)

let carte n =
  if n = 1 then "As"
  else if n = 11 then "Valet"
  else if n = 12 then "Reine"
  else if n = 13 then "Roi"
  else if n > 13 then "invalide"
  else string_of_int n
   (* int -> string *)

let us_time h m =
  let s = if h > 12 then "pm" else "am" in
  let h = if h > 12 then h - 12 else h in
  Printf.printf "%d:%d %s" h m s
   (* int -> int -> unit *)

let to_seq j h m s =
  float (j * 3600 * 24 + h * 3600 + m * 60) +. s
   (* int -> int -> int -> float -> float *)
```

Utilité du typage

Well-typed expressions do not go wrong.

Robin Milner 1978.

Dans les langages statiquement typé (comme OCaml), les opérations sont toujours appliquées à des arguments du bon type.

Certaines classes d'erreurs sont donc impossible. Cela donne du code robuste et permet d'éviter toute un catégorie de tests.

Certaines erreurs *dynamiques* sont toujours possible : division par 0, stack overflow, ...

L'inféference de type évite au programmeur le travail fastidieux d'écrire les types pour toutes les variables de son programme.

Utilité du typage (2)

Comme les tests, le typage est un **outil précieux** pour le développement et la maintenance des programmes.

Histoire d'horreur (adaptée)

Un programmeur Javascript définit une fonction `f(s)` qui prend en argument une chaîne de caractère représentant une date `s`. Cette fonction est une fonction utilitaire, utilisée par plein d'autres collaborateurs dans leur code.
Le programmeur décide changer cette fonction en `f(d,m,y)` qui prend trois entiers et il oublie de prévenir ses collègues.
Le code de l'application continue de s'exécuter. La fonction `f` est appelée avec une chaîne de caractère. Utilisée comme un nombre, elle est convertie en `NaN` (nombre invalide) et ne provoque pas d'erreur, elle renvoie juste des résultats incorrects.

En OCaml (mais aussi en Java ou en C++) : les collègues sont avertis automatiquement, le code ne compile plus ! Ils modifient leur code pour appeler `f` avec les bons arguments.

Plan

1 PFA (1) : Rappels d'OCaml ✓
2 PFA (2) : Rappels d'OCaml (suite et fin)
- 2.1 Fonctions récursives terminales ✓
- 2.2 Récursion mutuelle ✓
- 2.3 Inférence de types ✓
- 2.4 Polymorphisme
- 2.5 Listes
- 2.6 Fonctions anonymes
- 2.7 Itérateurs avancés
- 2.8 Applications partielles
- 2.9 Exceptions et gestion des erreurs

Fonctions `fst` et `snd`

On revient un moment sur les fonctions `fst` et `snd` qui permettent d'extraire la première et seconde composante d'une paire.
Elles sont définies dans la bibliothèque standard d'OCaml avec un simple filtrage :

```
let fst p =
    match p with
      (x, _) -> x

let snd p =
    match p with
      (_, y) -> y

let p1 = (1, 2)
let x1 = fst p1  (* x1 : int = 1 *)
let p2 = (true, "Hello")
let y2 = snd p2 (* y2 : string = "Hello" *)
```

Quel est le type de `fst` et `snd` ?

Fonctions `fst` et `snd` (2)

Essayons de simuler le compilateur OCaml et écrivons les équations que l'on peut obtenir en typant `fst` :

```
let fst p =
    match p with
      (x, _) -> x
```

- `p` : X_p, `fst` : X_p`->`X_{fst}
- X_p = X_a`*`X_b car `p` est comparé à un motif de paire.
- X_{fst} = X_a car la fonction renvoie `x` la première composante de la paire.
- C'est tout ! Il n'y a pas d'autre contrainte (`x` n'est pas comparé à un entier, on ne fait pas `x-1`, ...)

Le type final de `fst` est donc : X_a`*`X_b`->`X_a
C'est exactement ce qu'OCaml nous affiche :

```
# fst;;
val fst : 'a * 'b -> 'a = <fun>
```

Polymorphisme

Une fonction est dite **polymorphe** si le type de ses arguments ou de ses résultats n'est pas contraint. OCaml signale de tels types par : `'a`, `'b`, `'c`, ... (on lit α, β, γ, ...)

```
# fst;;
val fst : 'a * 'b -> 'a = <fun>
# snd;;
val snd : 'a * 'b -> 'b = <fun>
# let id x = x;;
val id : 'a -> 'a = <fun>
# let dup x = (x, x);;
val dup : 'a -> 'a * 'a = <fun>
```

Polymorphisme (2)

De telles fonctions peuvent être appliquées à n'importe quelles valeurs si les types sont compatibles :

```
# fst (1, 2);;
- : int = 1
# snd (true, "Hello");;
- : string = "Hello"
# id 42.0
- : float = 42.0
# dup false;;
- : bool * bool = (false, false)
```

On peut appliquer `fst` à n'importe quel type paire.

Note : c'est équivalent aux *variables génériques* du langage Java `HashMap<K, V>`

Plan

1 PFA (1) : Rappels d'OCaml ✓

2 PFA (2) : Rappels d'OCaml (suite et fin)

- 2.1 Fonctions récursives terminales ✓
- 2.2 Récursion mutuelle ✓
- 2.3 Inférence de types ✓
- 2.4 Polymorphisme ✓
- 2.5 Listes
- 2.6 Fonctions anonymes
- 2.7 Itérateurs avancés
- 2.8 Applications partielles
- 2.9 Exceptions et gestion des erreurs

Collections

Une des première chose que l'on veut faire en programmant est de gérer des collections de valeurs (liste d'élèves, ensemble de points, lignes d'un fichier, ...).
Pour cela, les types produits (n-uplets ou nommés) ne sont pas adaptés, car :

- La taille de la collection n'est pas connue à priori
- La taille de la collection peut varier au cours du calcul

Dans le cadre de la programmation fonctionnelle, on va s'intéresser aux collections immuables, c'est à dire non modifiables.

type 'a list

OCaml déifinit le type polymorphe `'a list`. Il s'agit de listes dont le type des éléments est `'a`. Ainsi, le type des listes d'entiers est `int list`, celui des listes de flottants `float list`, celui des listes de paires d'entiers `(int * int) list`, ...

- La liste vide se note `[ ]`
- On peut ajouter un élément en tête de liste avec l'opérateur `::`
- On peut créer une liste complète avec la notation $[e_1; \dots; e_n]$ qui est un alias pour :

$e_1 :: e_2 :: \dots :: e_n :: []$.

```
let l1 = [ 1; 2; 3 ]
let l2 = 0 :: l1      (* la liste [ 0; 1; 2; 3 ] *)
let l3 = 42 :: l1     (* la liste [ 42; 1; 2; 3 ] *)
```

type `'a list` (2)

Le type `'a list` est en fait un type somme définit dans la bibliothèque standard d'OCaml comme :

```
type 'a list = [] | :: of ('a * 'a list)
```

En d'autres termes, c'est un type avec deux cas :

- Un constructeur constant `[]` (la liste vide, aussi appelé Nil)
- Un constructeur `::` (appelé Cons) prenant deux arguments : la valeur en tête de liste et la suite de la liste.

Le code OCaml :

```
let lst = 1 :: 4 :: 3 :: [] (* équivalent à [ 1; 4; 3; ] *)
```

correspond en mémoire à :

::	1	⟶

::	4	⟶

::	3	⟶

[]

type `'a list` (3)

```
type 'a list = [] | :: of ('a * 'a list)
```

On remarque que la définition du type `'a list` fait référence au type `'a list` : c'est un type récursif.

Ça n'est pas spécifique à OCaml : les listes chaînées (en C, LinkedList<E> en Java, ...) sont des types récursifs aussi.

⇒ La plupart des fonctions sur les listes vont être *récursives*

On remarque aussi que le type `'a list` est un type somme avec deux constructeurs.

⇒ La plupart des fonctions sur les listes vont utiliser du *filtrage*.

Exemple

On souhaite calculer la longueur d'une liste :

```
let rec longueur l =
  match l with
    []  -> 0
  | _ :: ll -> 1 + longueur ll

(* val longueur : 'a list -> int *)
```

On a deux types de motifs possible sur les listes :

- `[] -> …` cas de la liste vide
- `e :: ll -> …` cas récursif : la liste est composée d'un premier élément `e` et d'une suite `ll`

Exécution de la fonction `longueur`

```
longueur [1; 4; 3]

longueur
          | :: | 1 | → |   | :: | 4 | → |   | :: | 3 | → |   | [] |

1 + longueur
          | :: | 4 | → |   | :: | 3 | → |   | [] |

1 + 1 + longueur
          | :: | 3 | → |   | [] |

1 + 1 + 1 + longueur
          | [] |

1 + 1 + 1 + 0
```

Une meilleure longueur

La fonction `longueur` n'est pas récursive terminale, on peut corriger ça :

```
let longueur lst =
  let rec loop l acc =
    match l with
      []  -> acc
    | _ :: ll -> loop ll (1+acc)
  in
    loop lst 0
(* val longueur : 'a list -> int *)
```

Exemple (2)

On souhaite écrire une fonction qui affiche une liste d'entiers dans le terminal :

```
let rec pr_int_list l =
  match l with
   [ ] -> ()  (* ne rien faire *)
  | i :: ll ->
       Printf.printf "%d\n" i;
       pr_int_list ll

 (* val pr_int_list : int list -> unit *)
```

On remarque que cette fonction est récursive terminale.

Exemple (3)

Une fonction qui renvoie le nombre de jour dans un mois :

```
let jours_mois = [
 ("janvier", 31); ("février", 28); ("mars", 31); ("avril", 30);
 ("mai", 31); ("juin", 30); ("juillet", 31); ("août", 31);
 ("septembre", 30); ("octobre", 31); ("novembre", 30); ("décembre", 31);
] (* val jours_mois : (string * int) list *)

let nb_jours m =
  let rec trouve_aux m l =
    match l with
    [ ] ->  failwith "Mois invalide"
    | (n, j) :: ll ->
        if m = n then (* on a trouvé le bon mois *)
          j
        else
          trouve_aux m ll
    in
  trouve_aux m jours_mois

 (* val nb_jours : string -> int *)
```

Exemple (4)

Une fonction qui inverse l'ordre des éléments d'une liste :

```
let renverse l =
   let rev renverse_aux l acc =
     match l with
        [] -> acc
     | e :: ll -> renverse_aux ll (e :: acc)
    in
    renverse_aux l []

  (* val renverse : 'a list -> 'a list *)
```

La fonction `renverse_aux` est récursive terminale.

Renverser une liste :

```
renverse [ 1; 2; 10; 3 ]

renverse_aux [ 1; 2; 10; 3 ] [ ]

renverse_aux [ 2; 10; 3 ] (1 :: [ ])

renverse_aux [ 10; 3 ] (2 :: (1 :: []))

renverse_aux [ 3 ] (10 :: (2 :: (1 :: [])))

renverse_aux [  ] (3 :: (10 :: (2 :: (1 :: []))))

(3 :: (10 :: (2 :: (1 :: []))))

[ 3; 10; 2; 1 ]
```

Listes et ordre supérieur

Le type `'a list` est paramétré par le type des éléments.
Il est donc naturel que les fonctions qui travaillent sur les listes soient paramétrées par d'autres fonctions permettant de spécialiser leur comportement.

On prend l'exemple de la fonction `iter` qui appelle une fonction `f` pour chaque élément d'une liste. Cette fonction ne renvoie pas de résultat.

La fonction `iter`

```
let rec iter f l =
  match l with
   [ ] -> ()  (* ne rien faire *)
  | e :: ll ->
       f e;
       iter f ll

 (* val iter : ('a -> unit) -> 'a list -> unit *)
```

Quelle est l'utilité de cette fonction ?

La fonction `iter` (2)

```
(* Des fonctions d'affichage *)
let pr_int i = Printf.printf "%d" i
(* val pr_int : int -> unit *)

let pr_float f = Printf.printf "%f" f
(* val pr_float : float -> unit *)

let pr_int_int p = Printf.printf "<%d, %d>" (fst p) (snd p)
(* val pr_int_int : (int * int) -> unit *)

let pr_int_list l = iter pr_int l
(* val pr_int_list : int list -> unit *)

let pr_float_list l = iter pr_float l
(* val pr_float_list : float list -> unit *)

let pr_int_int_list l = iter pr_int_int l
(* val pr_int_int_list : (int * int) list -> unit *)
```

Listes et ordre supérieur (2)

On va être amené a définir les opérations sur les listes en deux temps :

- On définit les parcours récursifs une fois pour toute dans des fonctions appelées *itérateurs de listes* (comme la fonction `iter`).
- On utilise ensuite ces itérateurs d'ordre supérieur en leur passant des fonctions spécifiques au type des éléments de la liste.

On doit vraiment écrire ces itérateurs ?

En pratique non ! Ils sont définis dans le module `List` de la bibliothèque standard. On en donne trois pour faire le TP de cette semaine :

- `List.iter : ('a -> unit) -> 'a list -> unit` : équivalent à la fonction `iter` présentée avant.
- `List.rev : 'a list -> 'a list` : la fonction qui renverse une liste.
- `List.assoc : 'a -> ('a * 'b) list -> 'b` : équivalent à la fonction `trouve_aux` écrite précédemment.

List.assoc

La fonction `List.assoc` prend en argument une clé et une liste de paires et renvoie la valeur associée à la clé dans la liste. Si aucune clé ne correspond, la fonction lève l'exception `Not_found`.

```
let dico = [ ("A", 10); ("B"; 100); ("D", 23) ]

let b = List.assoc "B" dico (* 100 *)

let z = List.assoc "Z" dico (* provoque une erreur Not_found *)

let  jours_mois = [ ("janvier", 31); ... ]

let nb_jours m = List.assoc m jours_mois
```

Plan

1 PFA (1) : Rappels d'OCaml ✓

2 PFA (2) : Rappels d'OCaml (suite et fin)

- 2.1 Fonctions récursives terminales ✓
- 2.2 Récursion mutuelle ✓
- 2.3 Inférence de types ✓
- 2.4 Polymorphisme ✓
- 2.5 Listes ✓
- 2.6 Fonctions anonymes
- 2.7 Itérateurs avancés
- 2.8 Applications partielles
- 2.9 Exceptions et gestion des erreurs

Les fonctions sont des valeurs comme les autres

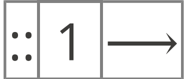
En programmation fonctionnelle lest fonctions sont des valeurs comme les autres :

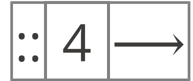
- Les fonctions peuvent prises comme argument (ordre supérieur) ✓
- On doit pouvoir définir des fonctions « n'importe où »
- On doit pouvoir renvoyer des fonctions comme résultat

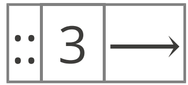
Qu'est-ce qu'une fonction ?

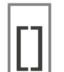
(on se pose la question du point de vue de la représentation en mémoire)

- Un entier : une suite de 64 bits
- Un flottant : une suite de 64 bits (organisée différemment)
- Une chaîne :un pointeur vers une zone mémoire. La zone mémoire contient un entier `t` (la taille de la chaîne) sur 64bits, puis `t` octets (les caractères)
- (un pointeur) :une suite de 64 bits
- Une liste : soit une valeur spéciale `[]` (correspondant au pointeur `null`) soit un pointeur vers un bloc de 2x64 bits : un pointeur vers la valeur dans la cellule, et un pointeur vers la suite









- une fonction ?

Représentation des fonctions

Considérons :

```
# let f =
  Printf.printf "Hello\n";
  let u = 42 in
  let g x = x + u in
  g;;
Hello
val f : int -> int = <fun>
# f 1;;
- : int = 43
# u;;
Error: Unbound value u
```

- `f` est un « alias » pour `g`
- Lorsqu'on exécute `f` (donc `g`), `u` est bien visible et défini
- `g` « se souvient » de la variable `u`

Représentation des fonctions (2)

En mémoire, les fonctions sont représentées par des *clôtures*, c'est à dire un couple (c, e) où :

- c est un pointeur vers la zone mémoire contenant le code de la fonction
- e est un ensemble contenant les valeurs des variables non-locales à la fonction au moment de sa définition.

On peut stocker de tels objets dans des structures de données simplement (c'est juste un couple, on peut le mettre dans une liste, dans un autre couple, dans une variable, ...).

Lorsqu'on exécute une fonction, le processeur effectue un `call` (ou `jal`) vers l'adresse où se trouve le code. Avant ça il place e ainsi que tous les arguments sur la pile.

Lorsque le code accède à une variable non-locale, il va la chercher dans e

Fonctions anonymes

On a vu qu'on peut définir des fonctions partout avec la notation `let f x = ... in ...`

Si on reprend l'exemple de `List.iter` :

```
let pr_int x = Printf.printf "%d" x ;;
let pr_int_list l = List.iter print_int l ;;
```

```
let pr_int_list l =
 let pr_int x = Printf.printf "%d" x
 in
 List.iter pr_int l
;;
```

Dans le deuxième cas, on n'a pas polué les définition avec une fonction globale.

Est-ce qu'on peut écrire ça de façon plus simple ?

Fonctions anonymes (2)

Dans la définition précédente, on a défini une fonction pour ne l'utiliser qu'à un seul endroit :

```
let pr_int_list l =
 let pr_int x = Printf.printf "%d" x
 in
 List.iter pr_int l
;;
```

On peut ré-écrire le code comme ceci :

```
let pr_int_list l =
 List.iter (fun x -> Printf.printf "%d" x) l
;;
```

La notation `fun` x_1 … x_n`-> e` permet de définir une fonction anonyme.

Fonctions anonymes (3)

Les fonctions anonymes sont particulièrement utiles lorsqu'elles sont utilisées avec des fonctions d'ordre supérieur:

```
let pr_int_list l =
List.iter (fun x -> Printf.printf "%d" x) l
;;

let pr_float_list l =
List.iter (fun x -> Printf.printf "%f" x) l
;;

let pr_int_int_list l =
List.iter (fun x -> Printf.printf "<%d, %d>" (fst x) (snd x)) l
;;
```

Plan

1 PFA (1) : Rappels d'OCaml ✓

2 PFA (2) : Rappels d'OCaml (suite et fin)

- 2.1 Fonctions récursives terminales ✓
- 2.2 Récursion mutuelle ✓
- 2.3 Inférence de types ✓
- 2.4 Polymorphisme ✓
- 2.5 Listes ✓
- 2.6 Fonctions anonymes ✓
- 2.7 Itérateurs avancés
- 2.8 Applications partielles
- 2.9 Exceptions et gestion des erreurs

le module `List`

En OCaml, les programmes sont structurés en modules. En particulier, chaque fichier définit un module. L'ensemble des fonctions et variables d'un module est accessible en utilisant le nom du module avec une Majuscule.

Le module `List` définit un grand nombre de fonctions utilitaires sur les listes.

le module List (2)

```
(* Ces trois fonctions sont à utiliser avec parcimonie ou pas du tout.
On privilégiera le filtrage *)

List.length : 'a  list -> int    (* Longueur d'une liste *)
List.hd : 'a list -> 'a (* Première valeur. Lève une exception sur la liste
                       vide *)
List.tl : 'a list -> 'a (* Suite de la liste. Lève une exception sur la liste
                       vide *)
(* *)
List.append : 'a list -> 'a list -> 'a list (* Concatène deux listes. Peut aussi
                                           se noter l1 @ l2 *)
List.rev : 'a list -> 'a list (* Renverse une liste *)
```

Itérateurs de listes

Parmi les fonctions du module `List`, certaines sont des itérateurs. Elles permettent d'appliquer une fonction d'ordre supérieur à chaque élément d'une liste.

List.iter

C'est le premier itérateur que l'on a vu. Il permet d'appliquer une fonction qui ne renvoie pas de résultat à tous les éléments d'une liste. On l'utilisera principalement pour faire des affichages.

```
List.iter : ('a -> unit) -> 'a list -> unit
```

```
# List.iter (fun x -> Printf.printf "%b\n" x) [ true; false; true ] ;;
  true
false
true
- : unit = ()
```

List.filter

Permet de filtrer, c'est à dire de renvoyer la liste de tous les éléments qui remplissent une certaine condition.

```
List.filter : ('a -> bool) -> 'a list -> 'a list
```

```
# List.filter (fun x -> x mod 2 = 0) [ 4; 5; 42; 1; 37; 49 ] ;;
    - : int list = [ 4; 42 ]
# List.filter (fun x -> x < 25.0) [ 10.5; 2.3; 99.0 ] ;;
    - : float list = [ 10.5; 2.3 ]
```

Le premier argument est appelé un prédicat.

List.map

Permet d'appliquer une transformation à chaque élément d'une liste et de renvoyer la liste des images.

`List.map f [`v_1`; … ;`v_n`]` ⇝ [(f v_1); … ; (f v_n)]

```
List.map : ('a -> 'b) -> 'a list -> 'b list
```

```
# List.map (fun x -> x * x) [ 4; 8; 3 ] ;;
    - : int list = [ 16; 64; 9 ]
# List.map string_of_int [ 1; 2; 3 ] ;;
    - : string list = [ "1"; "2"; "3" ]
```

Ça va jusqu'ici ?

Ça va se corser un peu ...

Agrégations

On souhaite souvent « combiner » tous les éléments d'une liste. Exemple

$$\sum_{i=1..n} i^2 = (((1 + 4) + 9) + \dots) + n^2$$

- Le $+$ est une opération binaire. On le rend « n-aire » en l'appliquant une première fois entre deux éléments. Puis en le répétant sur le résultat précédant et le nouvel élément, puis ...
- Quelle valeur renvoyer quand $n = 0$? 0 semble un choix raisonnable.

Ce type d'opération se retrouve souvent : somme, produit, min, max, ... :

$$\mathrm{Min}\ \{ v_1, \dots, v_n \} = \min(\min(\min(\min(v_1, v_2), v_3), \dots), v_n)$$

Comment exprimer ce genre d'opérations par un opérateur générique ?

List.fold_left

Permet d'appliquer une fonction d'agrégation aux éléments d'une liste.

```
List.fold_left : ('a -> 'b -> 'a ) -> 'a -> 'b list -> 'a
```

La fonction prend trois arguments

- La fonction d'agrégation
- La valeur initiale (utilisée en particulier en cas de liste vide)
- La liste d'éléments

```
List.fold_left f a [ v1; …; vn ] ⇝ f (f (f (f (f a v1) v2) v3) …) vn
```

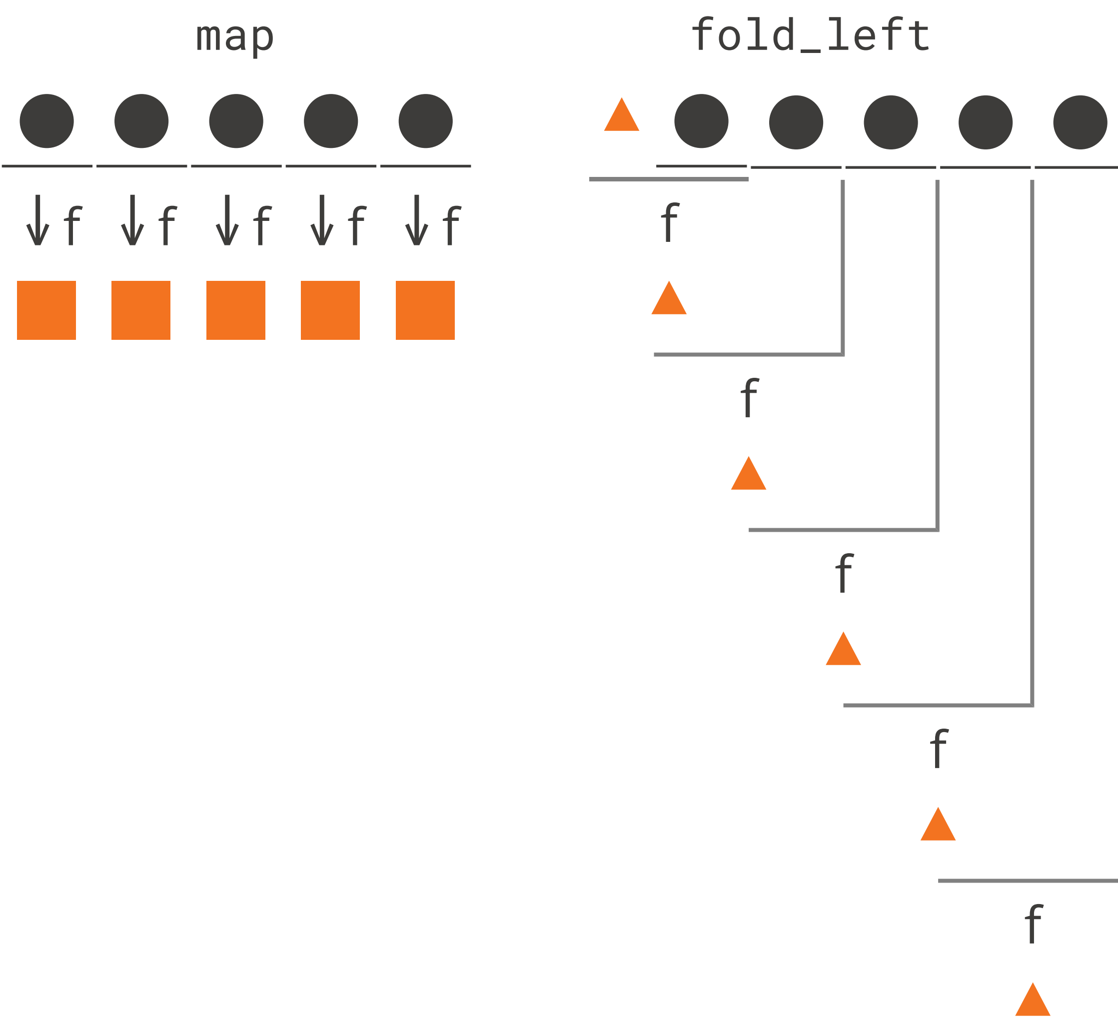
List.fold_left

```
# List.fold_left (fun a x -> a + x) 0 [ 1; 3; 7 ] ;;
    - : int = 11
# let f a x = a ^ " " ^ string_of_int x;;
    val f : string -> int -> string = <fun>
# List.fold_left f "" [ 1; 3; 7 ] ;;
    - : string = "1 3 7 "
```

Dans le code ci-dessus:

```
List.fold_left f "" [ 1; 3; 7 ]
f (f (f "" 1) 3) 7
f (f  "1 " 3) 7
f "1 3 " 7
f "1 3 7 "
```

Visuellement



Tri d'une liste

À proprement parler, le tri d'une liste n'est pas un itérateur. Il utilise cependant de l'ordre supérieur.

```
List.sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

La fonction `List.sort` prend en argument une fonction de comparaison.
La fonction de comparaison prend en argument deux valeur `a` et `b` et renvoie un nombre négatif (`a < b`), nul (`a = b`) ou positif (`a > b`).
En OCaml, la fonction prédéfinie `compare` a ce comportement.

```
compare : 'a -> 'a -> int
```

```
# List.sort compare [ 4; 8; 3 ] ;;
    - : int list = [ 3; 4; 8 ]
# List.sort compare [ "C"; "A"; "B"; "AX" ] ;;
    - : string list = [ "A"; "AX"; "B"; "C" ]
```

Plan

1 PFA (1) : Rappels d'OCaml ✓
2 PFA (2) : Rappels d'OCaml (suite et fin)
- 2.1 Fonctions récursives terminales ✓
- 2.2 Récursion mutuelle ✓
- 2.3 Inférence de types ✓
- 2.4 Polymorphisme ✓
- 2.5 Listes ✓
- 2.6 Fonctions anonymes ✓
- 2.7 Itérateurs avancés ✓
- 2.8 Applications partielles
- 2.9 Exceptions et gestion des erreurs

Exemple

Considérons la fonction suivante :

```
let f x =
(fun y -> x + y)
;;
```

C'est une fonction qui :

- prend en argument un entier `x`
- renvoie une fonction qui attend un `y` et renvoie `x + y`

```
# let f x =
  (fun y -> x + y)
;;
    val f : int -> int -> int = <fun>
```

Exemple (2)

```
# let g = f 3
  val g : int -> int = <fun>
# g 4;;
  - : int = 7
```

Ici, `f 3` renvoie une fonction (qu'on stocke dans `g`). Cette fonction attend un autre argument `y` et renvoie `3 + y`.

Si on s'intéresse aux types, quelle différence de type entre :

```
let f x = (fun y -> x + y) ;;
let add x y = x + y ;;
```

... aucune : `int -> int -> int`

Application partielle

On dit qu'une application est partielle si on applique une fonction à un nombre d'arguments inférieur à celui attendu.
En OCaml, ce n'est pas une erreur.
Si une fonction est de type : `t₁ -> t₂ -> … -> tₙ -> s`, alors on peut l'appliquer à au plus `n` arguments.
Si on l'applique à `k < n` arguments, le résultat est une fonction qui attend les `n-k` arguments restant.

```
# let f x y z = x + y + z;;
  val f : int -> int -> int -> int  = <fun>
# let g = f 10;;
  val g : int -> int -> int  = <fun>
# let h = g 5;;
  val h : int -> int  = <fun>
# h 6;;
  - : int  = 21
```

Application partielle et ordre supérieur

L'application partielle est particulièrement utile, combinée à l'ordre supérieur

```
let pr_int_list = List.iter (Printf.printf "%d");;
(* int list -> unit *)

let incr_int_list = List.map ((+) 1);;
(* int list -> int list *)

let sum_list = List.fold_left (+) 0 ;;
(* int list -> int *)
```

Plan

1 PFA (1) : Rappels d'OCaml ✓
2 PFA (2) : Rappels d'OCaml (suite et fin)
- 2.1 Fonctions récursives terminales ✓
- 2.2 Récursion mutuelle ✓
- 2.3 Inférence de types ✓
- 2.4 Polymorphisme ✓
- 2.5 Listes ✓
- 2.6 Fonctions anonymes ✓
- 2.7 Itérateurs avancés ✓
- 2.8 Applications partielles ✓
- 2.9 Exceptions et gestion des erreurs

Erreur

Il est courant, en programmation de vouloir signaler une erreur. Une des raisons pour laquelle une erreur peut se produire est liée à la différence entre une fonction mathématique et une fonction dans un langage de programmation.
Considérons :

$$f : (x, y) \mapsto x \div y \qquad \text{définie pour } x \in \mathbb{N}, y \in \mathbb{N} \setminus \{0\}$$

La fonction OCaml correspondante est :

```
let f x y = x / y   (* val f : int -> int -> int *)
```

En mathématiques, on ne **peut pas** appliquer la fonction à 0. En OCaml (comme dans de nombreux langages), le seul type à notre disposition est `int`, auquel `0` appartient. Un programmeur peut donc écrire `f 1 0` .

Exceptions

Dans le cas de `f 1 0` le programme OCaml lève une exception :

```
# let f x = x / y;;
val f : int -> int -> int = <fun>
# f 1 0;;
Exception: Division_by_zero.
```

Ici OCaml signale une erreur indiquant qu'une division par 0 est survenue.

Exceptions (2)

Une exception, si elle n'est pas gérée, interrompt le programme brutalement et affiche un message dans la console.

```
let () = Printf.printf "AVANT\n"
let x = 1 / 0
let () = Printf.printf "APRÈS\n"
```

```
$ ocamlc -o test test.ml
$ ./test
AVANT
Fatal error: exception Division_by_zero
$
```

Utilisation des exceptions

Une exception est utilisée lorsqu'un programme veut signaler que la poursuite du calcul est impossible. La plupart du temps, elle est levée par une fonction en réponse à un argument invalide.
Une exception peut aussi être déclanchée par OCaml sur du code parfaitement valide, pour signaler une erreur système. Par exemple :

- Un débordement de pile (exception `Stack_overflow`)
- Interruption par CTRL-C (exception `Break`)

Le type `exn`

En OCaml, toutes les exceptions appartiennent au même type : `exn`. Ce dernier est un type somme. Pour simplifier, on peut imaginer que celui ci a été défini comme :

```
type exn = Division_by_zero | Failure of string | Break | Not_found | …
```

Il est possible de définir ses propres exceptions, ce qui correspond à ajouter un nouveau cas au type `exn`.

```
exception MonErreur
exception MonErreurAvecMessage of string
exception MauvaiseValeur of int
```

Rattrapage d'exceptions

L'interêt des exceptions est qu'on peut les rattraper. On utilise pour cela la construction `try/with` qui ressembles à l'opération de filtrage `match/with` :

```
try
 e
with
Exception1 -> e₁
| Exception2 (s) ->  e₂
| …
```

Ici, `e` est évaluée en premier. Si elle ne provoque pas d'erreur, sa valeur est renvoyée. Sinon, si l'erreur provoquée est `Exception1` alors e_1 est renvoyée. Sinon si l'erreur `Exception2` est provoquée, alors le contenu de l'exception est stocké dans `s` et l'expression e_2 est renvoyée. Si aucune des exceptions listées ne correspond à l'erreur, cette dernière est propagée. Elle pourra alors interrompre le programme.

Fonction raise

La fonction prédéfinie `raise e` permet de lever l'exception `e` :

```
exception MonErreur(x, y)

let f x y =
if x < y then
raise (MonErreur (x, y)) (* on veut que x soit plus grand ! *)
else
x - y

...

let g u v =
try
let res = f u v in
Printf.printf "Résultat : %d\n" res
with
MonErreur (x, y) -> Printf.printf "Erreur, %d est plus petit que %d" x y
```

Exception avec message

Il est courant de vouloir lever une exception avec un message d'erreur.
L'exception prédéfinie en OCaml est `Failure of string`. Cette exception est tellement courante qu'il existe une fonction prédéfinie `failwith msg` qui lève cette exception avec le message passé en argument :

```
let aire_disque r =
  if r < 0.0 then
    failwith "Rayon négatif"
  else
    r *. r *. 3.14159
```

```
#
```