Introduction à la programmation fonctionnelle

Cours 4

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

université
PARIS-SACLAY

Résumé des épisodes précédents

- Expressions de base
- Types structurés : produits, produits nommés, types sommes.
- Filtrage par motifs
- Récursion
- Typage, polymorphisme
- Ordre supérieur

Plan

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.

```
Printf.printf "AVANT\n";;
let x = 1 / 0;;
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
;;
```

```
# aire_disque (-2.0);;
Exception: Failure "Rayon négatif".
#
```

Plan

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 $[e_1; \ldots; e_n]$ qui est un alias pour : $e_1 :: e_2 :: \ldots :: e_n ::[]$.

```
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)
```

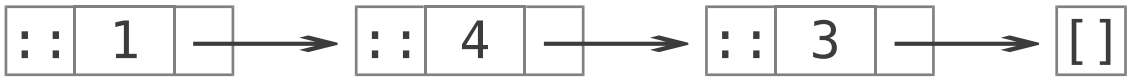
En d'autres termes, c'est un type avec deux cas :

- 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 :



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]
```


```
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 verra des itérateurs complexes dans le prochain cours.

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 ;;
```