

Programmation fonctionnelle

enslIE

Semestre 3 – 2026/27

Organisation du cours

Évaluation :

- ▶ Contrôle continu incluant 1 projet $\left(\frac{1}{3}\right)$
- ▶ examen final $\left(\frac{2}{3}\right)$



Introduction

Rappel : Programmation

Exécutable : suite de 0 et 1 directement compréhensible par la machine

```
001001000100010011  
110100010101001010  
111110010101001101
```

Programme : **abstraction** d'un exécutable pour le rendre plus facile à comprendre et à écrire

```
let _ =  
  Printf.printf "Bonjour _!"
```

Paradigme de programmation

Différents types d'abstractions possibles

⇒ différents paradigmes de programmation

- ▶ Assembleur : très proche de l'exécutable
- ▶ Impérative : modification de la mémoire, effets de bord
- ▶ Orientée Objet : structures de données et traitements associés regroupés au sein d'un « objet »
- ▶ Fonctionnelle : évaluation de fonctions mathématiques, persistance des données
- ▶ ...

Pas de paradigme meilleur que les autres

Plus adapté dans un certain contexte

Problème $\rightsquigarrow$ bonne structure $\rightsquigarrow$ langage adapté

Programmation fonctionnelle

Fonction : citoyen de première classe

- ▶ fonction = valeur comme une autre
- ▶ passage de fonctions comme arguments

Persistance :

- ▶ une valeur définie n'est **JAMAIS** modifiée
- ▶ très utile en cas de retour en arrière (undo, backtrack)

Programmation proche d'un raisonnement mathématique

- ▶ abstraction élevée
- ▶ code compact et clair
- ▶ modélisation et preuve formelle plus aisées

Ceci N'est PAS un cours de OCaml

Néanmoins on ne peut pas apprendre la programmation sans la pratiquer

On utilisera donc OCaml pour illustrer le cours

Langage d'origine française

<https://ocaml.org/>

Caractéristiques :

- ▶ Noyau fonctionnel + traits impératifs + couche objet
- ▶ Fortement typé, avec typage implicite
- ▶ Évaluation stricte
 - ▶ calcul des arguments avant appel de fonction
- ▶ Garbage Collector
 - ▶ gestion de la mémoire “par le langage”
 - ▶ rend la persistance efficace
- ▶ Langage interprété et compilé
 - ▶ Code compilé relativement efficace



Concepts de base

Programmation fonctionnelle

Principe de base : déclaration de constantes

`let identifiant = expression`

expression est évaluée et la valeur obtenue est donnée à la constante *identifiant*

Un programme est une succession de déclarations,
une expression pouvant utiliser les constantes déclarées auparavant

Fonctions et valeurs

Les **fonctions sont des valeurs** comme les autres

```
fun x -> expr
```

fonction (anonyme) qui à un argument *a* associe l'évaluation de *expr*
où on a remplacé *x* par *a*

Sucre syntaxique :

```
let f x y z = expr
```

pour

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

Ordre supérieur

Les fonctions elles-mêmes peuvent être passée en argument

```
fun f -> f 0
```

fonction qui prend en paramètre une fonction *f* et qui retourne la valeur de *f* en 0

```
let zero f = f 0
```

Les types OCaml sont **inférés** :

- ▶ le programmeur n'a pas besoin de les écrire explicitement
- ▶ le type le plus général d'une constante est calculé en fonction de l'expression

Le typage est fort :

- ▶ pas de conversion implicite

Le typage peut être polymorphe :

- ▶ un même algorithme peut être appliqué à des données de types différents
- ▶ ex : `let identite x = x`

Types de base

int : entiers machines

- ▶ 0 1 2 ...
- ▶ opérateurs usuels + - * / mod lxor

float : nombres à virgule flottante

- ▶ 0. 3.14 nan ...
- ▶ opérateurs usuels +. -. *. /. **

char : caractères ($\neq$ entiers!)

- ▶ 'a' 'b' '0' ...

string : chaînes de caractères ($\neq$ tableau de char!)

- ▶ "coucou" "Hello_World!\n" ...
- ▶ Concaténation ^

Types de base, suite

bool : booléens ($\neq$ entiers !)

- ▶ deux valeurs possibles **true** **false**
- ▶ opérateurs logiques `||` `&&` `not`
- ▶ comparaisons `=` `>` `<=` `<>`
- ▶ conditionnelle `if b then e1 else e2`
 - ▶ similaire au code C : `b ? e1 : e2`

unit : type spécial contenant **une** valeur

- ▶ `()`
- ▶ utilisé pour les effets de bord (cf. suite)
- ▶ séquence ;

Opérateurs de comparaison

Les opérateurs de comparaison sont polymorphes :

```
let equal x y = (x = y)
val f : 'a -> 'a -> bool = <fun>
```

```
let _ = 45.3 <= 20.
- : bool = false
```

```
let _ = "OCaml" > "Haskell"
- : bool = true
```

```
let _ = 'a' = 97
```

```
Error: This expression has type int but an
expression was expected of type char
```

Égalité physique vs. égalité structurelle

ATTENTION! `==` : égalité physique, même emplacement en mémoire, égalité de pointeurs

```
let _ = "coucou" == "coucou";;  
- : bool = false
```

`=` : égalité structurelle, même donnée

```
let _ = "coucou" = "coucou";;  
- : bool = true
```

Idem `!=` (différence physique) vs. `<>` (différence structurelle)

► **N'utiliser que `=` et `<>` !**

Combinaisons de valeurs

tuples : n -uplets d'éléments séparés par ,

```
let _ = 1, true, 2.3  
- : int * bool * float = (1, true, 2.3)
```

Attention : $(a, (b, c)) \neq (a, b, c) \neq ((a, b), c)$

listes : suite finie d'expressions **de même type**

```
let _ = []  
- : 'a list = []  
let _ = 1::2::3::[]  
- : int list = [1; 2; 3]  
let _ = 4.3::true::[];;  
Error: This expression has type bool but an  
expression was expected of type float
```

Appels de fonctions

Syntaxe : *fonction argument*

Parenthéser autour de *fonction* et/ou autour de *argument* si nécessaire

Currification

Fonction à n arguments : fonction à 1 argument qui retourne une fonction à $n - 1$ arguments

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

Application partielle :

```
let plus2 = add 2  
val plus2 : int -> int = <fun>
```

Éviter d'utiliser des n -uplets

Masquage

La valeur assignée à une constante est **permanente**

Elle ne peut pas être modifiée ou réaffectée

Elle peut par contre être **masquée**

```
let a = 1
val a : int = 1
let f x = x + a
val f : int -> int = <fun>
let a = 2
val a : int = 2
let _ = 0 + a
- : int = 2
let _ = f 0
- : int = 1    (* utilisation du a masqué *)
```

Déclarations locales

Syntaxe :

```
let identifiant = e1 in e2
```

identifiant est défini localement pour le calcul de *e2*, mais n'est pas utilisable en dehors

Autres usages du `let`

- ▶ Déclarations parallèles

```
let a = e1 and b = e2
```

- ▶ Déstructuration

```
let a, b, c = e1
```

```
let x::q = e2
```

(Attention ! e2 ne doit pas être [] *)*

Récurtivité

Une fonction peut être définie en s'appelant elle-même

```
let rec f = (fun x -> e)
```

► f peut apparaître dans e

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

Récurtivité mutuelle

```
let rec even x =  
  if x = 0 then true  
  else odd (x - 1)  
  
and odd y =  
  if y = 0 then false  
  else even (y - 1)
```

Filtrage de motif

Syntaxe :

```
match e with  
  pattern1 -> e1  
| pattern2 -> e2  
...  
| patternk -> ek
```

e1 e2 ... ek de même type

Dans les motifs, on peut utiliser des variables qui prendront la valeur des sous-expressions

Exemple de filtrage

```
let _ = match [1] with
  [] -> 0
  | x::[] -> 1
  | _ -> 2
- : int = 1
```

```
let _ = match [42; 3; -5] with
  [] -> 0
  | x::_ -> x
- : int = 42
```

Conditionnelle vs. filtrage

Branchement et filtrage sont deux choses différentes

- ▶ **if** = test sur les valeurs
 - ▶ utile par exemple pour tester la valeur d'un entier
- ▶ **match** = test sur la structure d'une expression
 - ▶ utile par exemple pour raisonner sur la longueur d'une listes
 - ▶ permet d'extraire des valeurs en fonction de la structure

Exceptions

Syntaxe :

```
try e  
with pattern1 -> e1 | ... | patternk -> ek
```

```
exception My_exception
```

```
exception My_other_exception of type
```

```
raise Exception
```

```
raise (Exception e)
```



Exemples

Quelques fonctions prédéfinies

- ▶ fonctions sur les entiers `abs succ pred`
- ▶ fonctions mathématiques sur les réels `sqrt log sin etc.`
- ▶ quelques constantes `max_int epsilon_float etc.`
- ▶ fonctions liées à la comparaison `min max compare`
- ▶ manipulation des couples `fst snd`

Cf. module Pervasives (Stdlib depuis 4.07)

Calcul de 2^n

Comment faire ?

- pas de boucle. . .

Calcul de 2^n

Comment faire ?

- pas de boucle. . . mais on a la récursivité $\rightsquigarrow$ **let rec**

$$2^0 = 1$$

$$2^n = 2 \times 2^{n-1}$$

si $n > 0$

Calcul de 2^n

Comment faire ?

- pas de boucle...mais on a la récursivité $\rightsquigarrow$ `let rec`

$$2^0 = 1$$

$$2^n = 2 \times 2^{n-1} \qquad \text{si } n > 0$$

```
let rec pow2 = fun n ->  
  if n = 0 then 1  
  else 2 * pow2 (n - 1)
```

Longueur d'une liste

Comment faire ?

Longueur d'une liste

Comment faire ?

- ▶ Raisonnement par cas sur la liste (vide ou non) $\rightsquigarrow$ **match**
- ▶ Calcul récursif $\rightsquigarrow$ **let rec**

Longueur d'une liste

Comment faire ?

- ▶ Raisonnement par cas sur la liste (vide ou non) $\rightsquigarrow$ **match**
- ▶ Calcul récursif $\rightsquigarrow$ **let rec**

$$\text{longueur } [] = 0$$

$$\text{longueur } (x :: q) = 1 + \text{longueur } q$$

Longueur d'une liste

Comment faire ?

- ▶ Raisonnement par cas sur la liste (vide ou non) $\rightsquigarrow$ **match**
- ▶ Calcul récursif $\rightsquigarrow$ **let rec**

$$\text{longueur } [] = 0$$

$$\text{longueur } (x :: q) = 1 + \text{longueur } q$$

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

Maximum d'une liste d'entier

Comment faire ?

Maximum d'une liste d'entier

Comment faire ?

- ▶ Déconstruire la liste $\rightsquigarrow$ **match**
- ▶ Calcul récursif $\rightsquigarrow$ **let rec**
- ▶ Cas de [] ??? $\rightsquigarrow$ exception

Maximum d'une liste d'entier

Comment faire ?

- ▶ Déconstruire la liste $\rightsquigarrow$ **match**
- ▶ Calcul récursif $\rightsquigarrow$ **let rec**
- ▶ Cas de [] ??? $\rightsquigarrow$ exception

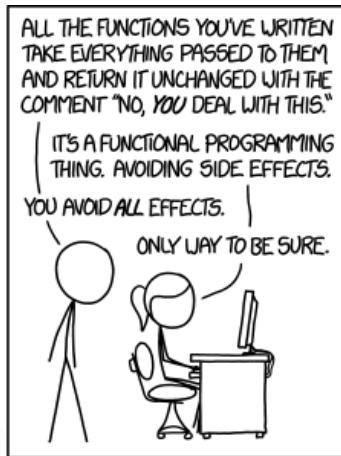
Une solution

```
let rec max_list l =  
  match l with  
    [] -> raise (Invalid_argument  
                  "max_list_of_an_empty_list")  
  | x::[] -> x  
  | x::q -> let m = max_list q in  
             max x m
```



Effets de bord

Effets de bord



extrait de <https://xkcd.com/1790/>

Entrées/sorties

En pratique, les programmes OCaml sont compilés

Pour avoir une utilité, il faut qu'ils puissent interagir avec leur environnement

- ▶ Affichage (terminal, graphique)
- ▶ Entrées utilisateur
- ▶ Lecture/écriture dans fichiers

Non fonctionnel, mais possible

Le module Printf

Fournit la fonction `Printf.printf` de syntaxe similaire à celle de `printf` en C

`Printf.printf` *chaine_de_format* *e1* ... *ek*

Exemple :

```
let _ =  
    Printf.printf "%d->%s%g\n" 1 "OK" 2.3  
1 -> OK 2.3  
- : unit = ()
```

Aussi `Printf.sprintf`, `Printf.fprintf`, etc.

La fonction `readline`

Permet de récupérer une ligne de texte

```
let line = read_line ()
```

Bonjour !

```
val line : string = "Bonjour_!"
```

Argument `()` permet de lancer l'exécution

Aussi `read_int`, `read_float`

Le module Scanf

Fournit la fonction `Scanf.scanf` un peu similaire à `scanf` en C mais prend une fonction en dernier argument (dit que faire des données lues)

```
Scanf.scanf format (fun x1 ... xk -> e)
```

Exemple :

```
let r =  
  Scanf.scanf "%d␣%g"  
    (fun x f -> float_of_int x +. f)  
2 3.14  
val r : float = 5.1400000000000000057
```



Types définis par le programmeur

Enregistrements

Syntaxe

```
type enr = {  
  champ1 : type1;  
  champ2 : type2;  
  ...  
  champn : typen  
}
```

```
let v = { champ1 = e1;  
  champ3 = e3; ...;  
  champ2 = e2 }
```

```
let u = v.champ4
```

Persistence

Attention : fonctionnel $\rightarrow$ non modifiable

`v.champ1 = e` : test d'égalité

Possibilité d'utiliser :

```
let w = { v with champ5 = e5; champ1 = e1' }
```

- `w` nouvelle valeur identique à `v` sauf pour les champs `champ5` et `champ1`

Types énumérés

Syntaxe

```
type t = Constr1 | Constr2 | ... | Constrn
```

Manipulation par test ou par filtrage

- Ici, question de goût

Types somme

Syntaxe

```
type t = Constr1 of type1  
    | Constr2 (* volontairement sans type *)  
    | ...  
    | Constrn of typen
```

Manipulation par filtrage

- ▶ permet de récupérer la valeur associée

Types inductifs

Syntaxe

```
type t = Constr1 of type1  
      | Constr2 (* volontairement sans type *)  
      | ...  
      | Constrn of typen
```

Possibilité d'utiliser t dans type1, ..., typen

Manipulation par filtrage et récursion

Exemples

Listes :

```
type intlist = Nil | Cons of int * intlist
```

Arbres binaires

```
type inttree =  
    Leaf of int  
    | Node of int * inttree * inttree
```

Polymorphisme

Enfin, un type peut dépendre d'un type :

- ▶ on parle de polymorphisme
- ▶ essentiel pour définir des structures de données **génériques**

Notation pour les variables de type = 'ident

Listes polymorphes :

```
type 'a list = Nil | Cons of 'a * 'a list
```

Arbres binaires

```
type 'a tree =  
    Leaf of 'a  
    | Node of 'a * 'a tree * 'a tree
```

Alias de type

Nouveau nom pour un type existant

```
type t = type existant
```

Exemple :

```
type intlist = int list
```

Attention ! on peut utiliser le nouveau nom dans les types, mais en général c'est celui existant qui est retourné par OCaml