Introduction à la programmation fonctionnelle

Cours 1

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

université
PARIS-SACLAY

Attributions

Ce cours reprend de nombreux éléments du cours de Sylvain Conchon (resp. IPF jusqu'en 2020)

La page de l'an dernier : https://www.lri.fr/~conchon/IPF/

Autres ressources :

Apprendre à Programmer avec OCaml (Conchon, Filliâtre, ed. Eyrolles)

Un mot sur l'organisation

- 6 séances de cours et TP
- Cours le mercredi après-midi amphi H4
- TP les mercredi après-midi (LDD2 IMg1/g2, L2 g3), le jeudi matin (L2 g2) et le jeudi après-midi (L2 g1)
- Examen (en présentiel) au moment des partiels ou un peu après

DM ou TP Noté : 40%, examen : 60%

Pour les LDD2 IM Uniquement : mini-projet dans le prolongement de l'UE (5 ou 6 séances de TP jusqu'en décembre).

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions
- 1.1 Langages de programmation
- 1.2 Le langage OCaml
- 1.3 Types simples
- 1.4 Expressions
- 1.5 Fonctions

Définitions

Un *langage* est un système de communication **structuré**. Il permet **d'exprimer une pensée** et de **communiquer** au moyen d'un système de signes (vocaux, gestuel, graphiques, ...) doté **d'une sémantique**, et le plus souvent **d'une syntaxe**.

Un *langage de programmation* est un système de communication **structuré**. Il permet **d'exprimer un algorithme** de façon à ce qu'il soit **réalisable par un ordinateur**. Il est doté **d'une sémantique**, et **d'une syntaxe**.

Syntaxe et sémantique

La *syntaxe* est l'ensemble des **règles de bonne formation du langage**.

Exemple avec du code OCaml:

```
1 + 4          (* est syntaxiquement correct *)

1 / 'Bonjour' (* est syntaxiquement correct *)

1 +            (* est syntaxiquement incorrect *)
```

La *sémantique* est l'ensemble des règles qui donne le **sens** des programmes **bien formés**

```
1 + 4          (* est sémantiquement correct *)

1 / "Bonjour" (* est sémantiquement incorrect *)
```

Quels langages de programmation ?

La tour de Babel, Pieter Brueghel l'Ancien, 156

Quels sont les caractéristiques des langages de programmation ?

- Généralistes ou dédiés : certains langages ont un but spécifique (par exemple SQL pour interroger les bases de données) d'autres sont généralistes (comme C++, Python ou Java)

- Compilés ou interprétés : des langages compilés sont traduits par un **compilateur** en instructions machines (C, C++, Java, ...).
Les langages interprétés disposent d'un **interprète** qui permet d'évaluer des phrases du langages (le Shell, Python, JavaScript, ...).

- Typés dynamiquement ou statiquement

- Impératifs, fonctionnels, orienté objets, logiques, ...

- ...

Est-ce une bonne chose ?

Oui !

- Il n'y a pas un unique langage de programmation qui soit le meilleur
- Un bon programmeur ou une bonne programmeuse se doit de connaître **plusieurs** langages
- Connaître plusieurs langages permet d'aborder des problèmes de façon différentes
- Cela permet aussi de choisir le meilleur outil pour résoudre son problème

Sur le cycle de Licence, au moins 5 langages

- C++ (programmation impérative)
- Python (programmation impérative et objet)
- Java (programmation Orientée Objet)
- **OCaml** (programmation fonctionnelle)
- SQL (interrogation de bases de données)

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions
- 1.1 Langages de programmation ✓
- 1.2 Le langage OCaml
- 1.3 Types simples
- 1.4 Expressions
- 1.5 Fonctions

Caractéristiques du langage

- Langage généraliste : traitement de données, interfaces graphiques, réseau, jeux, calcul scientifique, intelligence artificielle, ...
- Langage compilé (comme C++ ou Java)
- Langage typé **statiquement** (comme C++ ou Java)

Le langage supporte différents paradigmes : fonctionnel, impératif, orienté objet.

Dans ce cours, on n'utilise que **le fragment fonctionnel**

Un premier programme

On considère le fichier `salut.ml`

```
let limit = 40                    (* On définit une variable globale *)
let () = Printf.printf "Quel est votre age ?\n" (* On affiche un message *)
let age = read_int ()     (* On lit un entier sur l'entrée standard *)
let msg =
  if age >= limit then            (* On teste la valeur  *)
     "vieux"
  else
     "toi"
let () = Printf.printf "Salut, %s!\n" msg
```

On peut compiler ce programme dans un terminal :

```
$ ocamlc -o salut.exe salut.ml
$ ./salut.exe
Quel est votre age ? 41
Salut, vieux !
$
```

Qu'y a t'il dans ce programme ?

- Définitions de variables
- Entrées (de l'utilisateur) et affichages (dans la console)
- Manipulation de constantes (nombres, textes, ...)
- Tests de valeurs
- Des commentaires

Le programme est **compilé** (comme Java ou C++)

Comment programmer avec OCaml ?

On privilégie pour les TPs l'éditeur VSCode

1. Ouvrir un terminal
2. Créer et se placer dans un répertoire pour le TP (par exemple `IPF/TP1`)
3. Lancer un VSCode :

```
$ code .
$
```

4. Créer des fichiers `.ml`, éditer le code, sauver le fichier
5. Tester le programme :

```
$ ocamlc -o exo1.exe exo1.ml
$ ./exo1.exe
```

Le site du cours contient des instructions pour installer OCaml sur votre machine.

La boucle d'interaction

Le langage OCaml possède aussi un **mode interactif** qui permet d'évaluer des instructions, comme un shell.

Il suffit de lancer la commande `ocaml` sans argument.

```
$ ocaml
        OCaml version 4.12.1

Down v0.0.4 loaded. Type Down.help () for more info.
# 1 + 1 ;;
- : int = 2
# 3 * 10 ;;
- : int = 30
# let x = 42 ;;
val x : int = 42
# x + 10 ;;
52
#
```

On peut quiter avec `CTRL-d`

Compilation vs. mode intéractif

Mode intéractif

- attend des expressions ou des définitions OCaml et les exécute au fur et à mesure.
- peut être utilisé pour tester des petits morceaux de programmes
- phrase rentrée dans ce mode interactif doit **obligatoirement** se terminer par `;;`

Compilation

- Le programme `ocamlc` compile les fichiers sources en code-objet interprété par une machine virtuelle (comme Java)
- Le programme `ocamlopt` compile les fichiers sources en exécutables natifs (« binaires ») et effectue de nombreuses optimisations.

Programmation fonctionnelle

C'est un paradigme de programmation dans lequel :

- On ne fait pas d'effets de bords : pas de mise à jour de variables, de tableaux (sauf pour les entrées/sorties)
- On n'écrit pas de boucles, mais des fonctions **récursives**
- Les fonctions sont des objets de première classe (comme les entiers et les chaînes de caractères) :
  - On peut passer des fonctions en argument à d'autres fonctions
  - On peut renvoyer des fonctions comme résultat d'autres fonctions
  - On peut stocker des fonctions dans des structures de données
  - On peut définir des fonctions à n'importe quel endroit du code (dans d'autres fonctions en particulier)

C'est une **façon de programmer** particulièrement concise, puissante et qui peut être efficace. Elle vient compléter les autres styles de programmation : impératifs et orienté objet.

Programmation fonctionnelle en OCaml

Pourquoi faire de la programmation fonctionnelle **en OCaml** ?

TOUS les langages de programmation modernes supportent le paradigme fonctionnel :

- C++ (depuis C++11)
- Java (depuis Java 8, 2013)
- Javascript (proprement depuis 2015)
- Python (depuis Python 3)

Mais :

- Ils sont inutilement verbeux (Java, C++)
- Ils se contortionnent pour faire faire rentrer tout ça dans leur concepts de base comme les classes et les interfaces (Java)
- Ils ne sont pas typés (Javascript, Python)
- Ils limitent arbitrairement la récursion (Python)

Au début ...

Les premiers TPs vont peut être paraître arides :

- On se concentre sur une syntaxe nouvelle
- On doit penser différement

Ils deviendront plus sexy au fur et à mesure qu'on avencera dans le langage (programmation système, graphique, ...)

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions
- 1.1 Langages de programmation ✓
- 1.2 Le langage OCaml ✓
- 1.3 Types simples
- 1.4 Expressions
- 1.5 Fonctions

Les entiers (`int`)

En OCaml, les entiers ont une taille fixe : 63bits sur une architecture 64bits ou 31bits sur une architecture 32bits (un bit est reservé dans chaque entier **en plus du bit de signe**) :

```
# 1 ;;
- : int = 1
# -149 ;;
- : int = -149
# 1234567891011 ;;
- : int = 1234567891011
```

Opération sur les entiers

Symbole	Description
+	addition
-	soustraction
*	multiplication
/	division entière
mod	modulo

```
# 1 - 9 ;;
- : int = -8
# 3 * 4 ;;
- : int = 12
# 5 / 3 ;;
- : int = 1
# 10 mod 2 ;;
- : int = 2
# 4 + 3.5 ;;
Error: This expression has type float but an
  expression was expected of type int
```

Erreur de type ?

En OCaml, les expressions ont **un type et un seul**. C'est aussi valable pour les fonctions et les opérateurs. `+` est l'addition entre entiers.

À l'inverse d'autres langages il n'y a pas de conversion implicite entre types, il faut utiliser des conversion explicites.

Appels de fonctions

En OCaml, on appelle une fonction en donnant simplement son nom, suivi des arguments sans parenthèse :

```
f 1 2 3 ;;   (* on appelle la fonction f sur 3 arguments *)

g 4 ;;       (* on appelle la fonction g sur un seul argument *)
g (2 + 2) ;; (* on appelle la fonction g sur 1 seul argument *)
```

Cette notation étrange sera justifiée dans le prochain cours

Les nombres à virgule (`float`)

En OCaml, les « nombres à virgule » ont une précision limitée. On les représente en utilisant la notation scientifique :

```
# 1.5 ;;
- : float = 1.5
# -12.3423e13 ;;
- : float = -123423000000000.0
# 1.55555555555555555555555
- : float = 1.5555555555555558
```

Remarque : `-12.3423e13` = `-12.3423` × 10^{13} = `-123423000000000.0`

Attention : En OCaml, comme dans de nombreux langages, calculer avec des nombres à virgule (nombres *flottants*) peut provoquer des erreurs d'arrondi.

OCaml (comme C, C++, Java, Python, Javascript, ...) utilise le standard IEEE-754 pour les flottants. C'est aussi celui implémenté en matériel par les processeurs et les cartes graphiques.

Opérations sur les flottants

Symbole	Description
`+.`	addition
`-.`	soustraction
`*.`	multiplication
`/.`	division
`**`	puissance
`float` *i*	conversion `int→float`
`float_of_int` *i*	conversion `int→float`
`int_of_float` *f*	conversion `float→int`
`sqrt` *f*	racine carrée
`sin` *f*	sinus
`cos` *f*	cosinus
`tan` *f*	tangeante
`log` *f*	logarithme (naturel)
`log10` *f*	logarithme (base 10)

Opérations sur les nombres flottants

```
# 1.5 +. 1.5 ;;
 - : float = 3.
# 3.141592653589793 *. 2.0 ;;
- : float = 6.2831853071795862 3
# 10.5 /. 3.0 ;;
 - : float = 3.5
# 1.2 +. 1.2 +. 1.2 ;;
 - : float = 3.5999999999999964
# 4.5 ** 100.0 ;;
2.0953249170398633e+65
#  1.0 /. 0.0 ;;
 - : float = infinity
```

Chaînes de caractères (`string`)

On représente les « textes » par des **chaînes de caractères**.

```
# "Bonjour, ça va bien ?"
 - : string = "Bonjour, ça va bien ?"
```

On ne montre que quelques opérations sur les chaînes de caractères :

Symbole	Description
`^`	concaténation
`String.length` *s*	longueur

Entrées/sorties

On se contentera d'entrées et sorties simples :

- Affichage formatté avec `Printf.printf`
- Lecture d'entrées sur le terminal
- Accès aux arguments du programme

Dans un second temps, on verra comment lire et écrire des fichiers.

Printf.printf

La fonction `Printf.printf` est similaire à la fonction C du même nom. C'est une fonction `variadique` (nombre arbitraire d'arugments)

Le premier argument doit être une *chaîne de format* qui indique combien d'arguemnts lire ensuite et comment les afficher.
Dans cette chaîne les séquences suivantes sont spéciales :

- `%s` lit l'argument suivant qui doit être une chaîne et l'insère à cet endroit.
- `%d` lit l'argument suivant qui doit être un `int` et l'insère à cet endroit.
- `%f` lit l'argument suivant qui doit être un `float`

Exemple :

```
Printf.printf "Un entier: %d, une chaîne: \"%s\", un flottant: %f\n"
   42 "foo" 3.14;;
```

```
Un entier: 42, une chaîne: "foo", un flottant: 3.14
```

Quel type pour la fonction `Printf.printf` ?

Si on exécute la fonction `Printf.printf` dans le terminal quel est le type du résultat ?

```
# Printf.printf "1+1 ça fait %d!\n" 2 ;;
1+1 ça fait 2!
- : unit = ()
#
```

Le résultat est du type `unit`. Ce type contient une seule valeur spéciale notée `()`.

Il est utilisé par les fonctions qui ne renvoient pas de résultats (affichage par exemple) ou qui ne prennent aucun argument.

On peut le voir comme un équivalent de `void` en Java.

Lecture au clavier

Plusieurs fonctions permettent de lire des données saisies au clavier :

- `read_int` : permet de lire un entier
- `read_float` : permet de lire un flottant
- `read_line` : permet de lire une ligne de texte

Ces fonctions prennent `()` en argument

Arguments d'un programme

Dans les langages comme C ou Java, il y a une fonction principale `main`

Cette dernière reçoit en argument un tableau contenant les arguments passés au programme sur la ligne de commande.

Dans les langages sans fonction principale comme OCaml (mais aussi Python ou Javascript), les arguments sont stockés dans un tableau global. En OCaml se tableau est dans la variable globale. `Sys.argv`.

On peut accéder aux éléments d'un tableau avec la notation `t.(i)`.
Exemple :

```
if Array.length Sys.argv >= 1 then
   Printf.printf "Le premier argument est %s\n" Sys.argv.(1)
```

Le tableau contient toujours au moins une case, le nom du programme dans lequel on est (dans `Sys.argv.(0)`)

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions
- 1.1 Langages de programmation ✓
- 1.2 Le langage OCaml ✓
- 1.3 Types simples ✓
- 1.4 Expressions
- 1.5 Fonctions

Structures d'un programme

Un programme OCaml est consituté d'une suite d'éléments, terminés par `;;`. Ces éléments peuvent être :

- Des définitions de variables gloables de la forme `let v = e`
- Des expressions sans résultats (par exemple des affichages) de la forme `let () = i`
- Des définitions de fonctions (voir plus loin)

Il n'y a pas de point d'entrée, un programme est exécuté dans l'ordre du fichier.

En OCaml il n'y a pas de notion de « d'instruction », il n'y a que des expressions. Certaines de ces instructions renvoient `()`, pour indiquer qu'elles ont eu un effet (affichage, écriture dans un fichier, ...)

if/then/else

Un test `if/then/else` est une **expression** dont l'évaluation renvoie la valeur de l'expression dans la branche `then` ou `else`

Les deux expressions de chaque branche doivent avoir **le même type**

Ainsi, on peut écrire :

```
1 + (if x > 42 then 3 else 4)
```

Cette expression renvoie `4` si x est plus grand que 42 et `5` sinon.
Si on compare du code C++/Java et du code OCaml

```
let y =
    if x > 42 then
      4
    else
      5
```

```
int y;
if (x > 42) {
    y = 4;
} else {
    y = 5;
}
```

if/then/else (2)

Si la branche `then` est du type `unit` (pas de résultat), alors on peut omettre la branche `else`

```
if e > 10 then
    Printf.printf "e est plus grand que 10!\n"
```

Si on veut mettre plusieurs instructions de type `Unit` à la suite, on peut utiliser les mots clés `begin` et `end` et **séparer** les expressions par des `;`.

```
if e > 10 then begin
   Printf.printf "e est plus grand que 10!\n";
   Printf.printf "Si si je vous jure !\n";
   Printf.printf "Il est vraiment plus grand!\n"  (* pas de ; ici *)
end
```

`begin` et `end` jouent le même rôle que `{` et `}` en Java.

Les booléens

L'algèbre de Boole (George Boole, 1847) est une branche de l'algèbre dans laquelle on ne considère que deux valeurs : `true` et `false`.
Les opérations sur ces valeurs sont la négation (`not`), le « ou logique » (`||`) et le « et logique » (`&&`).

On peut manipuler ces objets en OCaml, comme on le fait avec des entiers, des nombres à virgule ou des chaînes de caractères.

```
# true ;;
- : bool = true
# false ;;
- : bool = false
# not true ;;
- : bool = false
# true || false ;;
- : bool = true
# true && false ;;
- : bool = false
```

Les comparaisons

Les booléens servent à exprimer le résultat d'un **test**. Un cas particulier de test sont les comparaisons. Les opérateurs de comparaisons en OCaml sont :

Symbole	Description
=	égal
<>	différent
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal

Attention : dans les premiers cours on ne comparera que des nombres. Les comparaisons d'autres types (chaînes de caractères par exemple) seront expliquées plus tard. Les comparaisons `==` et `!=` existent aussi, mais on les verra plus tard.

Les comparaisons (exemples)

Le résultat d'une comparaison est toujours un booléens (`True` ou `False`) :

```
# 1 + 1 = 2;;
- : bool = true
# 3 <= 10 ;;
- : bool = true
# let x = 4;;
val x : int = 4
# x > 3 && x < 8;;
- : bool = true
# x <> 4 ;;
- : bool = false
```

Définition de variables

Une **variable** est un moyen de donner un **nom** au résultat d'un calcul.
En OCaml, une variable est une suite de caractères qui commence par une lettre minuscule ou un « _ » et contient des lettres, des chiffres ou des « _ ».
On définit une variable avec le mot clé « `let` ».

```
# let x = 2 ;;
val x : int = 2
# let y = 3 ;;
val y : int = 3
# let z = x + y;;
val z : int = 5
```

Définition de variables locales

On peut définir des variables locales à une expression avec les mots clés `let … in`

```
# let x = 2  in x + x;;
- : int = 4
# let y = 3 ;;
val y : int = 3
# let z = 4 in z + y;;
- : int = 7
# x + y;;
Error: Unbound value x
```

L'expression `let x =` e_1 `in` e_2 permet de définir la variable `x` uniquement le temps du calcul de e_2. Elle prend tout son sens lorsqu'on la combine à d'autres expressions comme le `if/then/else`.

Définitions de variables locales (exemple)

On peut comparer les deux codes OCaml et Java :

```
let norm =
  if z > 10 then
        let x2 = x *. x in
        let y2 = y *. y in
        sqrt (x2 +. y2)
  else
     -1.0
```

```
double norm;
if (z > 10) {
   double x2 = x * x;
   double y2 = y * y;
   norm = Math.sqrt (x2 + y2);
} else {
   norm = -1.0;
}
```

Dans les deux cas, les variables `x2` et `y2` ne sont plus visibles en dehors du bloc `then`.

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions
- 1.1 Langages de programmation ✓
- 1.2 Le langage OCaml ✓
- 1.3 Types simples ✓
- 1.4 Expressions ✓
- 1.5 Fonctions

Définitions de fonctions

En OCaml, on définit une fonction aussi avec le mot clé `let`

```
let carre n = n * n

let aire_triangle base hauteur =
       base *. hauteur * 0.5

let a = aire_triangle 5.0 14.5
```

La syntaxe générale d'une fonction est :

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

où `e` est l'expression dont la valeur est renvoyée.
⇒ il n'y a pas de mot-clé `return` en OCaml.
Bien sûr, un fonction peut avoir un corps complexe avec des `let … in`, des `if/then/else`

Exemple : formattage d'une heure

On veut écrire une fonction qui prend en argument un nombre de secondes et renvoie une chaîne de caractères au format : `j h min s`

```
let format_time t =
    let j = string_of_int (t / (24 * 3600)) in
    let t = t mod (24 * 3600) in
    let h = string_of_int (t / 3600) in
    let t = t mod 3600 in
    let m = string_of_int (t / 60) in
    let s = string_of_int (t mod 60) in
    j ^ "j " ^ h ^ "h " ^ m ^ "m " ^ s ^ "s"

let s = format_time 145999
let () = Printf.printf "%s\n" s
 (* affiche 1j 16h 33m 19s *)
```

Fonctions récursives

On n'a pas vu comment faire des boucles. Hors la répetition de code est un pilier important de la programmation (et sa raison d'être initiale).

On peut contourner l'absence de boucles en écrivant des fonctions **récursives**. Une fonction récursive est une fonction qui s'appelle elle même.
Commençons par l'exemple standard de la factorielle, écrit en OCaml :

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

let () = Printf.printf "fact 10 = %d\n" (fact 10)
(* Affiche 3628800 *)
```

On introduit des fonctions récursives avec le mot clé `let rec`

Écritures de fonctions récursives

Lorsqu'on écrit une fonction récursive, on distingue **TOUJOURS** deux types de cas

- Le ou les cas **de base** : ce sont les cas pour lesquel on n'effectue pas d'appel récursif, ils sont calculables directement
- Le ou les cas **récursifs** : ce sont les cas qui dépendent du calcul récursif

Lorsque l'on fait un appel récursif, l'argument doit toujours **« se rapprocher »** du cas de base.

```
let rec fact n =
  if n <= 1 then (* cas de base *)
     1
   else (* cas récursif *)
     n * fact (n-1)   (* on se rappelle sur n-1, donc on
                         arrivera à 1 ou 0 à un moment *)
```

Pour les premiers cours, les fonctions récursives seront toujours sur des entiers

Écriture de fonctions récursives (2)

On donne un autre exemple, la fonction `fizzbuzz` (utilisée comme « échauffement » dans beaucoup d'interviews techniques)

- La fonction énumère les entiers entre 1 et n
- Si `n` est un multiple de `3` la fonction affiche `Fizz`
- Si `n` est un multiple de `5` la fonction affiche `Buzz`
- Si `n` est un multiple de `3` et de `5` la fonction affiche `FizzBuzz`
- Dans les autres cas on n'affiche rien

```
let rec fizzbuzz_aux i n =
  if i <= n then (* cas récursif *)
     let i3 = i mod 3 = 0 in
     let i5 = i mod 5 = 0 in
     begin
      if i3 && i5 then Printf.printf "FizzBuzz\n"
      else if i3 then Printf.printf "Fizz\n"
      else if i5 then Printf.printf "Buzz\n";
      fizzbuzz_aux (i+1) n   (* on se rappelle sur (i+1) → n *)
     end
;;
```

Écritures de fonctions récursives (2)

Dans un premier temps, les fonctions récursives auront **toujours** la forme (pseudo-code) :

```
let rec f n … =
   if test sur n then
       cas de base
    else
       cas récursif, appel sur f (n±e)
```