Cours 1 – Le langage OCaml

Programmation fonctionnelle
CFA INSTA - Master 1 - Analyste Développeur

Pierre Talbot (pierre.talbot@univ-nantes.fr)

Université de Nantes

9 avril 2019

UNIVERSITÉ DE NANTES

Le menu

Organisation du cours

- Apprentissage du paradigme de **programmation fonctionnelle**.
  - Principalement via le langage **OCaml**.
- Méthode d'enseignement :
  - Alterner entre contenu théorique et mise en pratique rapide.
  - On part d'un petit **langage noyau** bien défini qu'on étend au fil des cours.
  - Évaluation continue fréquente.
- **80% du cours en 1 semaine**
  - Votre investissement est crucial cette semaine.
  - Modèle d'enseignement flexible pour la progression de chacun.

Évaluation

- Examen fin de semestre (écrit) : 35%
  - Épreuve de ~2h sur papier, document autorisé.
  - Le 3 mai.
- Contrôle continu (écrit) : 30%
  - 3 épreuves de ~45 minutes sur papier, document non autorisé.
  - (correction rapide en classe).
  - Le 10, 11 et 16 avril.
- Un projet : 35%
  - Improvisation musicale et *live coding*.
  - À rendre pour le 28 avril 23h59.
  - Présentation des projets et retour le 3 mai.

Ressources

À votre disposition pour réussir dans les meilleures conditions :

- Site web : `http://www.hyc.io/teaching/ocaml.html`
- Email : pierre.talbot@univ-nantes.fr
- Google Group : `https://groups.google.com/d/forum/ocaml-insta`

N'hésitez pas à poser des questions sur le groupe ou par email.

Programmation fonctionnelle

Quelques traits de la programmation fonctionnelle :

- Mémoire **non mutable** : pas de $x = x + 1$, on doit ranger la valeur dans une nouvelle case mémoire !
- **Fonction de premier ordre** : on peut passer des fonctions à des fonctions qui nous renvoie des fonctions, c'est clair ? :

Et OCaml ?

- Typé statiquement avec inférence de types (*Hindley–Milner type system*).
- Polymorphisme paramétrique.
- Types algébriques avec du *pattern matching*.
- ... des types quoi.

Pourquoi la programmation fonctionnelle ?

Énorme **influence sur les langages modernes** ces dernières années :

- Java 8 : lambdas, itérateurs fonctionnels (`stream`).
- C++11,14,17 : lambdas, inférence de types, et autres idiomes via `Boost`.
- Rust : lambdas, inférence de types, itérateurs fonctionnels.

Nouveaux langages fonctionnels :

- Scala : langage fonctionnel objet.
- F# : langage fonctionnel dans l'environnement Microsoft.
- Swift : langage fonctionnel dans l'environnement Apple.
- ReasonML : version de OCaml avec une autre syntaxe de Facebook.

Pourquoi OCaml ?

- Statiquement et fortement typé.
- Multi-paradigmes : fonctionnelle, modulaire, impératif, objet.
- Efficace.
- Récent, environnement de programmation moderne (`opam`, IDE, `dune`).

Pourquoi OCaml ?

Pour la passion et la pa$$ion.

Plan du cours

- Day 1. Introduction à la programmation fonctionnelle.
- Day 2. Types algébriques.
- Day 3. Système de type.
- Day 4. Programmation modulaire.
- Day 5. Présentation des projets, examen et notions avancées.

Crédits

Ce cours est inspiré de....

- Cours OCaml de l'Université de Nantes de Charlotte Truchet.
- Cours MPIL de Sorbonne Université de Emmanuel Chailloux : `http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2018/ue/3I008-2019fev/`
- MOOC OCaml : `https://www.fun-mooc.fr/courses/course-v1:parisdiderot+56002+session03/about`

Merci à eux !

Le menu

Syntaxe du noyau réduit OCaml (K_1)

$\langle n, m, p, q \ldots \rangle$ ::=	**Expression**
x, **y**, **f**, **blah**, ...	(identifiant)
0, **1**, **2**, ...	(nombre entier)
true, **false**	(booléen)
() | `(`p`)`	(unit et parenthésage)
n `+` m | n `-` m | n `*` m | n `/` m | n `mod` m	(arithmétique)
n `>` m | n `>=` m | n `<` m | n `<=` m | n `=` m	(comparaison)
`not` n | n `&&` m | n `||` m	(logique)
`if` n `then` p `else` q	(alternative)
`let` **x** `=` p `in` q | `let` **x** `=` p	(déclaration)
`fun` **x** `->` p	(fonction)
`let rec` **x** `=` p `in` q	(déclaration récursive)
f n	(appel de fonction)
	entrées-sorties (en bonus...)
`let _ = Printf.printf "hello world %d" 1 in` p	(affichage)
`let` **x** `= Scanf.scanf "%d" (fun` **x** `->` **x**`) in` p	(saisie)

Un exemple de bout en bout

Soit la célébrissime suite de Fibonacci :

$$\begin{cases} \texttt{fib}(0) & = & 1 \\ \texttt{fib}(1) & = & 1 \\ \texttt{fib}(n) & = & \texttt{fib}(n-2) + \texttt{fib}(n-1) \end{cases}$$

Le programme OCaml correspondant est écrit de la sorte :

```
let rec fib = fun n ->
  if n = 0 || n = 1 then
    1
  else
    fib (n-1) + fib (n-2)

let _ =
  let n = Scanf.scanf "%d" (fun x -> x) in
  let res = fib n in
  let _ = Printf.printf "Resultat : fib(%d) = %d" n res in
  ()
```

Un exemple de bout en bout

- Compilation de ce programme avec la commande :

```
ocamlc fibonacci.ml -o fibo
```

- Exécution de ce programme avec la commande :

```
./fibo
```

Vous avez aussi à votre disposition un REPL (*read-evaluate-process-loop*), c'est-à-dire un évaluateur interactif :

```
$ rlwrap ocaml
        OCaml version 4.07.1

# let x = 4 + 2;;
val x : int = 6
```

Ajout des nombres flottants et string (K_2)

$\langle n, m, p, q \ldots \rangle$	$::=$	**Expression**
$\mid$	...	(noyau réduit K_1)
$\mid$	**0.**, **1.23**, ...	(nombre flottant)
$\mid$	$n +. m \mid n -. m \mid n *. m \mid n /. m$	(arithmétique)
$\mid$	**"a"**, **"abc"**, ...	(chaine de caractère)
$\mid$	$n \hat{} m$	(concaténation)

Attention : Pas de conversion implicite entre les types primitifs en OCaml. Tout doit être converti explicitement, vous avez les fonctions suivantes :

```
int_of_float: float -> int
float_of_int: int -> float
int_of_string: string -> int
(...)
```

Voir `https://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html`

Types du langage K_2 (Ty_2)

$\langle T, U, \ldots \rangle$ ::=

Types

`int`, `float`, `string`, `char`, `bool`, `unit`	(types primitifs)
`T -> U`	(fonction)

On aura donc les expressions suivantes :

```
4: int
"blah": string
(): unit
'a': char
(fun x -> x + 1): (int -> int)
```

Quelques exercices !

Le menu

Les fonctions

- Raccourci syntaxique.
- *Variable shadowing*.
- Imbrication de fonction.
- Récursion terminale.
- Technique de l'accumulateur.
- Fermeture fonctionelle.
- Application partielle.
- Currification et décurrification.

Raccourci syntaxique

Simplification de la déclaration de fonctions :

```
let f = fun x -> x + 1      →   let f x = x + 1
fun x -> fun y -> x + y     →   fun x y -> x + y
```

Éléments de syntaxe

- Les paramètres de fonctions de sont pas séparés par des virgules.
- À l'appel d'une fonction, parenthèses et virgules non nécessaires !

```
let f = fun x y -> x + y in
Printf.printf "%d" (f 2 4)
```

Variable Shadowing

L'expression `let` *x* `=` *p* n'est pas une affectation au sens des langages impératifs mais une **définition** qui **ne changera jamais**.

- On peut cacher temporairement un identifiant par un autre du même nom.

```
let x = 1 in
if x = 1 then
  let y = 2 in
  let x = 4 in
  x + y   (* x=4 et y=2 *)
else
  x - 1   (* x=1 *)
```

Imbrication de fonction

On peut facilement déclarer des fonctions à l'intérieur d'autres fonctions :

```
let x_times_n x n =
  let rec aux n =
    if n = 0 then 0 else x + (aux x (n-1))
  in
    aux n
```

La fonction aux (pour *auxiliaire*) n'est visible que dans la fonction `x_times_n`.

Récursion terminale

Fibonacci comme on l'a écrit est problématique, pourquoi ?

```
let rec fib n =
  if n = 0 || n = 1 then 1
  else fib (n-1) + fib (n-2)
```

Récursion terminale

Fibonacci comme on l'a écrit est problématique, pourquoi ?

```
let rec fib n =
  if n = 0 || n = 1 then 1
  else fib (n-1) + fib (n-2)
```

- Parce que la fonction n'est pas *récursive terminale*.
- Une fonction est récursive terminale si il ne reste pas d'opération à effectuer la récursion entamée.

Récursion terminale

Une version en temps linéaire de la fonction de Fibonacci existe :

```
let fib n =
  let rec aux a b n =
    if n = 0 then a
    else aux b (a + b) (n-1)
  in aux 1 1 n
```

Et elle est également terminale récursive.

- Le compilateur effectue une optimisation appelée *tail call optimization* sur les fonctions récursive terminales.

Technique de l'accumulateur

- *Stack overflow* possible si récursion non terminale.
- Solution : Utiliser un accumulateur en paramètre contenant le résultat courant du calcul.

Voir exercices 4.1 et 4.2.

Capture des variables

- Les fonctions *capturent l'environnement de variable*.

```
let f x =
  let g y = x + y in
  g
in
  f 5 4
```

- La fonction `g`, à sa définition, a accès à `x`.
- La fonction `f` retourne la fonction `g` qui a *capturé la variable* x.

Fermeture fonctionnelle

- On appelle une *fermeture fonctionnelle* une **fonction + environnement de variables**.
- Notez que l'environnement de variables **reste inchangé** : pas de crainte que la variable `x` capturée change lorsqu'on exécutera `g`.

```
let f x =
  let g y = x + y in
  g
in
  f 5 4
```

Fonction de premier ordre

Une fonction qui renvoie une autre fonction, ici `f` renvoie `g`.

Application partielle

- Une des forces du paradigme fonctionnelle est la grande flexibilité dans la manipulation des fonctions.
- Cela vient en partie du fait qu'on ne soit pas obliger de donner tous les paramètres à une fonction.

```
let plus x y = x + y in
let plus_3 = plus 3 in
plus_3 5
```

On verra que c'est très utile pour passer des fonctions à d'autres fonctions.

Currification

Currification

Transformer une fonction a plusieurs paramètres en une fonction a un seul paramètre :

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

Décurrification

Transforme *une fonction retournant une fonction* en *une fonction retournant une valeur* :

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

Permet la *défonctionalisation* d'un programme : enlever les fonctions de premier ordre.

Le menu

Conclusion

- Ce premier cours couvre la base de la programmation fonctionnelle, à savoir les fonctions.
- Il y a de nombreuses techniques associées à ce paradigme, non communes dans le paradigme impératif.
- Par exemple : l'application partielle, les fonctions de premier ordre, la récursion terminale.