

Au programme

Programmation Fonctionnelle

Objective Caml

David Teller

2007-01-08

Programmation fonctionnelle
Objective Caml

Paradigmes de base

- Impératif** Le système est décrit sous la forme d'opérations à enchaîner, qui manipulent les données du problème.
- Orienté-Objets** Le système est décrit sous la forme d'objets qui représentent les données du problème, échantent des messages et changent d'état.
- Déclaratif** Le problème est retranscrit directement comme un ensemble de spécifications.
- Fonctionnel** Le système est décrit sous la forme de fonctions mathématiques.
- Logique** Le système est décrit sous la forme de propositions logiques et de règles de déduction.

Paradigmes composés

- Prouvée** Construire une preuve mathématique et en extraire un programme.
- Dynamique** Programmes auto-modifiables.
- Passage de Messages** Plusieurs programmes s'exécutent en même temps et communiquent.
- Objets Concurrents/Distribués** Les objets communiquent et changent d'état simultanément.

...

Oui mais...

Q Pourquoi enseigner un autre paradigme/langage de programmation ?

A Parce qu'un seul paradigme/langage de programmation, ça rétrécit l'esprit.

A "LISP is worth learning for a different reason – the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use LISP itself a lot." (Source : *How to become a Hacker*)

A "A programming language is low level when its programs require attention to the irrelevant." – et la programmation fonctionnelle a pour objectif d'augmenter le niveau d'abstraction.

A L'essentiel des nouveaux concepts de programmation commencent leur vie dans le monde de la programmation fonctionnelle (exceptions, généricité, systèmes de types...)

... ça continue !

Q Pourquoi enseigner un autre paradigme/langage de programmation ?

A Parce qu'un certain nombre de problèmes se traitent de manière beaucoup plus naturelle avec un langage fonctionnel.

A Parce qu'il faut toujours avoir plusieurs cordes à son arc.

A Et puis parce qu'il s'agit d'une autre manière de programmer, plus rigoureuse, plus proche des mathématiques (et peut-être plus simple).

Objectifs du semestre

- ▶ Concepts de base de la programmation fonctionnelle.
- ▶ Récursivité.
- ▶ Abstraction.
- ▶ Et surtout, comment *penser* fonctionnel !

... Le tout à l'aide d'OCaml.

Objectifs du jour

1. Qu'est-ce que la programmation fonctionnelle ?
2. À propos d'OCaml.
3. Premiers pas avec OCaml.
4. Modalités de notation.

Introduction

La programmation fonctionnelle

OCaml

- Valeurs et types
- Fonctions
- Interlude
- Compilation
- Déclaration de valeurs

Modalité de notation

Fonctions

- Application
- Construction
- Récursivité

Types de données

- Système de types
- Types
- Polymorphisme

Conclusions



La programmation fonctionnelle

La programmation fonctionnelle est un paradigme dans lequel un système est décrit comme un ensemble de fonctions mathématiques.

Q Que manque-t-il par rapport à la programmation impérative ?

A Les variables. Mais vous verrez qu'on peut s'en passer.



Pourquoi ?

Q Oui, mais pourquoi faire ?

A Composabilité : si deux programmes n'utilisent pas de variables, on peut les exécuter l'un après l'autre (ou simultanément) et être certain qu'ils auront le même effet.

A Modularité : il est plus simple de décomposer un programme en nombreuses fonctions avec des dépendances claires.

A Fondements mathématiques : un programme fonctionnel est un théorème et a une signification mathématique précise.

A Prouvabilité : un programme fonctionnel peut être totalement prouvé.

A Abstraction : ne programmer qu'une seule fois les tâches répétitives, exprimer "naturellement" des contraintes complexes.

A Débogage : programmes simples à tester et à vérifier.

Et puis c'est fun.



Quelques langages fonctionnels

1. Lisp, Scheme
2. ML (OCaml, SML)
3. Haskell
4. Erlang
5. F#



Quelques applications fonctionnelles

Assistants de Preuve Coq, Isabelle, HOL
Mise en page Xslt, Ant, Texmacs
Réseau MLDonkey, SpamOracle
Utilitaires Unison, IceDock
Langages de programmation ...
Langages de développement de matériels ...

C'est tout

C'est tout sur la programmation fonctionnelle.
Enfin, pour le moment.

Introduction
La programmation fonctionnelle

OCaml

Valeurs et types
Fonctions
Interlude
Compilation
Déclaration de valeurs

Modalité de notation

Fonctions

Application
Construction
Récursivité

Types de données

Système de types
Types
Polymorphisme

Conclusions

OCaml

Objective Caml est

- ▶ un langage de programmation
- ▶ français
- ▶ un des successeurs de ML
- ▶ conçu (initialement) pour écrire d'autres langages de programmation
- ▶ fortement typé
- ▶ capable de manipuler des fonctions
- ▶ rapide
- ▶ concis
- ▶ multi-paradigmes
- ▶ sûr
- ▶ ...

C'est parti

OCaml dispose d'un compilateur, comme Java, ou d'une ligne de commande (toplevel).

Commençons par dire "bonjour" :

```
# "Hello_world" ;;
```

```
- : string = "Hello_world"
```

Bonjour

```
# "Hello_world" ;;  
- : string = "Hello_world"
```

Décomposons cela :

- ▶ # est l'invite de commande : OCaml attendant des instructions – *vous n'avez pas à le copier !*
- ▶ "Hello_world" est une valeur – c'est d'ailleurs une chaîne de caractères.
- ▶ ;; est la fin de phrase, elle dit à OCaml que les instructions sont terminées et doivent être exécutées.
ensuite vient la réponse d'OCaml :
- ▶ - désigne le résultat de votre dernière instruction
- ▶ : string précise que le résultat des instructions précédentes est une chaîne de caractères
- ▶ - "Hello_world" donne la valeur du résultat de la dernière instruction.

Ce programme n'a aucun *effet*. Par contre, il renvoie un *résultat*.

Arithmétique

De même, on aura

```
# 1 ;;  
- : int = 1  
# 1 + 2 ;;  
- : int = 3
```

```
* 1 + 3 ;;  
This expression has type string but is here used with type int  
# 1 / 0 ;;  
Exception : Division_by_zero .
```

Notez le retour

- ▶ du système de types ;
- ▶ des exceptions (nous y reviendrons).

Booléens

OCaml dispose, bien entendu, de booléens :

```
* true;;
- : bool = true
* not true;;
- : bool = false
* not (not true);;
- : bool = true
* not (not (not true));;
- : bool = false
* true && false;;
- : bool = false
* not !;;
This expression has type int but is here used with type bool

* if true then "Oui!" else "Non!";;
- : string = "Oui!"
```

Notez au passage que toute expression "légale" a un type et une valeur.

Conversions

On peut (essayer de) convertir des valeurs d'un type à un autre

```
* string_of_int 1000;;
- : string = "1000"
* int_of_string "1000";;
- : int = 1000
* int_of_string "Gloubiboulga";;
Exception: Failure "int_of_string".
```

Notez les conversions entre nombres flottants et nombres entiers.
Notez aussi qu'un nombre flottant et un entier ne peuvent pas être additionnés !

Au fait...

Q Qu'est-ce que c'est que ce `string_of_int` ?

A Demandons à OCaml !

```
* string_of_int 1000;;
- : string = "1000"
```

En d'autres termes...

```
* string_of_int 1000;;
- : string = "1000"
```

- ▶ - : commence la réponse d'OCaml
- ▶ `int -> string` précise que le résultat des instructions précédentes est une fonction à un seul argument (qui doit être de type `int`) et dont le résultat est une chaîne de caractères (c'est-à-dire `string`)
- ▶ - <fun> précise que OCaml ne veut/peut pas afficher le contenu de la fonction elle-même.

Moralité Les fonctions *sont* des valeurs !

En parlant de fonctions

Les opérateurs sont des fonctions comme les autres.

```
* (+) ::
- : int -> int -> int = <lex>
* (-) ::
- : int -> int -> int = <lex>
* (+) 10 22 ::
- : int = 32
```

Q Sachant que (+) est une fonction, comment lire les résultats précédents ?



Affichage de valeurs

OCaml définit des fonctions `print_string`, `print_int` ... pour afficher les valeurs.

```
# print_string ;;
```

```
# print_string "HelloWorld";;
```

Notez le type `unit` et la valeur `()`.



Composons un peu...

Bien souvent, il est utile de composer des valeurs.

Ainsi, on peut écrire

```
* ( 3 + 5 ) - 2 ::
- : int = 6
* (-) ((+) 3 5) 2 ::
- : int = 6
* (if 3=5 then 0 else 10) + 5 ::
- : int = 15
```

ou même

```
* (if 3=5 then (+) else (-)) 5 10::
```

Q Quel est le résultat ?

A Demandons à OCaml !



[Introduction](#)

[La programmation fonctionnelle](#)

OCaml

[Valeurs et types](#)

[Fonctions](#)

[Interlude](#)

[Compilation](#)

[Déclaration de valeurs](#)

[Modalité de notation](#)

[Fonctions](#)

[Application](#)

[Construction](#)

[Récursivité](#)

[Types de données](#)

[Système de types](#)

[Types](#)

[Polymorphisme](#)

[Conclusions](#)



Ce que nous avons vu

- ▶ La notion de programmation fonctionnelle.
- ▶ Le toplevel de OCaml.
- ▶ Quelques valeurs, quelques types
- ▶ ... y compris quelques fonctions.



Note sur le travail individuel

Même si vous n'avez pas d'ordinateur personnel, vous aurez besoin de maîtriser OCaml rapidement, de manière à mener à bien votre projet. D'ailleurs, vous aurez un projet à rendre en fin de semestre et à développer par vous-mêmes d'ici-là.
Plus de détails à la fin de ce cours
Retour à OCaml.



Documents

- ▶ Toute la documentation de OCaml est disponible sur <http://www.ocaml.org>.
- ▶ OCaml aussi est disponible sur <http://www.ocaml.org>.
- ▶ L'ensemble est gratuit.
- ▶ Sous Windows, il existe aussi une variante d'OCaml adaptée aux produits Microsoft (et à Visual Studio), nommée F#.

Introduction

La programmation fonctionnelle

OCaml

Valeurs et types

Fonctions

Interlude

Compilation

Déclaration de valeurs

Modalité de notation

Fonctions

Application

Construction

Récursivité

Types de données

Système de types

Types

Polymorphisme

Conclusions



Autres modes d'exécution

Il est possible d'utiliser OCaml comme

1. un toplevel (`ocaml`)
2. un interpréteur (`ocaml`)
3. un compilateur (`ocamlc`).

Compilateur

Pour compiler un fichier `mon_fichier.ml` et obtenir un exécutable `mon_executable`, on emploie la ligne de commande suivante :

```
ocamlc mon_fichier.ml -o mon_executable
```

Testons !

Il suffit alors d'appeler `mon_executable`.

Interprétation

Pour exécuter directement un programme stocké dans un fichier `mon_fichier.ml`, on emploie la ligne de commande suivante :

```
ocaml mon_fichier.ml
```

Subtiles distinctions

Considérons les deux programmes suivants : `"Hello World ";;` et `print_string "Hello World ";;`

Q Quelle est la différence ?

A Le premier a une valeur mais aucun effet. Le deuxième a un effet mais aucune valeur.

Valeurs simples

La déclaration des valeurs se fait sous la forme "soit x défini par ...".

```
# let x = 1;;
```

Ou, si l'on veut,

```
# let x, y = 1, 2;;
```

```
val x : int = 1
```

```
val y : int = 2
```

```
# let z = 1, 2;;
```

```
val z : int * int = (1, 2)
```

Vous avez remarqué ?

Vous n'avez pas besoin de donner le type de la valeur à OCaml.
OCaml déduit le type à partir de la valeur.
C'est ce qu'on appelle l'*inférence de types*.

Utilisation de valeurs

Il est possible de donner un nom à une valeur

```
# let x = 1;;  
val x : int = 1
```

```
# x;;  
- : int = 1
```

```
# let y = x + 1 ;;  
val y : int = 2
```

Masquage de valeurs

Attention, une valeur peut en masquer une autre !

```
# let x = 1;;
val x : int = 1

# x;;
- : int = 1

# let y = x + 1;;
val y : int = 2

# let x = "bleuu!";;
val x : string = "bleuu!"

# x;;
- : string = "bleuu!"

# y;;
- : int = 2

# let y = x+1;;
This expression has type string but is here used with type int
```

⏪ ⏩ ⏴ ⏵ ↺ 🔍

Déclarations locales

Souvent, on déclarera les valeurs de manière locale.

```
# let x = 1 in x*x;;
- : int = 1

# x;;
Unbound value x
```

⏪ ⏩ ⏴ ⏵ ↺ 🔍

Masquages locaux

De même, une valeur locale peut masquer (temporairement) une autre valeur

```
# let x = 1 in let x = 2 in x*x;;
- : int = 4

# let x = 1;;
val x : int = 1

# let x = 10 in x*x;;
- : int = 100

# x;;
- : int = 1
```

⏪ ⏩ ⏴ ⏵ ↺ 🔍

Introduction

La programmation fonctionnelle

OCaml

- Valeurs et types
- Fonctions
- Interlude
- Compilation
- Déclaration de valeurs

Modalité de notation

Fonctions

- Application
- Construction
- Récursivité

Types de données

- Système de types
- Types
- Polymorphisme

Conclusions

⏪ ⏩ ⏴ ⏵ ↺ 🔍

Exercices

Comme au premier semestre, entre deux cours, des exercices de programmation ou/et de théorie.

Les exercices seront

- ▶ plus courts qu'à u premier semestre
- ▶ obligatoires
- ▶ notés systématiquement.

Projet

Un projet à mener à bien par vous-mêmes, avec rapport à rendre à la fin du semestre, à faire par binôme.

Quelques idées :

- ▶ traceur de courbes
- ▶ cryptographie
- ▶ partage de secrets
- ▶ approximation de courbes
- ▶ moteur d'Intelligence Artificielle
- ▶ d'autres idées ?

Examen

Et enfin un examen sur machine, en fin d'année, dans l'esprit de l'examen du premier semestre.

Introduction

La programmation fonctionnelle

OCaml

Valeurs et types

Fonctions

Interlude

Compilation

Déclaration de valeurs

Modalité de notation

Fonctions

Application

Construction

Récursivité

Types de données

Système de types

Types

Polymorphisme

Conclusions

Fonctions

Une fonction est une valeur comme une autre.

```
# function x -> x+1;;
- : int -> int = <fun>
# let f = function x -> x +1;;
val f : int -> int = <fun>
# let g x = x +1;;
val g : int -> int = <fun>
# let h = fun x -> x +1;;
val h : int -> int = <fun>
```



Construire une fonction

D'ailleurs, une fonction peut prendre une autre fonction en argument ou renvoyer une fonction.

```
# let valeur_absolue f = function x -> if (f x > 0) then
val valeur_absolue : ('a -> int) -> 'a -> int = <fun>

# let valeur_absolue f x = if (f x > 0) then f x else (-
val valeur_absolue : ('a -> int) -> 'a -> int = <fun>
```



Application d'une fonction

Pour appliquer une fonction, on écrit la fonction suivie de ses arguments (avec des parenthèses, si nécessaire, pour forcer les priorités).

```
# (function x -> x+1) 100;;
- : int = 101
# g (g (g 100));;
- : int = 103
```

Comme pour toute autre valeur, on peut remplacer une fonction par son nom.



Au fait

Q Vous avez remarqué 'a ?

A 'a se prononce "alpha" – c'est une variable de type.
Nous en discuterons plus tard.



Fonctions récursives

Une fonction peut s'appeler elle-même, à condition d'avoir été déclarée à l'aide de `let rec`.

```
# let rec factorielle n = if (n > 2) then n * (factorielle (n - 1)) else 1
val factorielle : int -> int = <fun>
```

La *récursivité* peut être vue d'au moins deux manières :

- ▶ du point de vue du programmeur, c'est une manière plus puissante d'écrire des boucles
- ▶ du point de vue du mathématicien, c'est une manière de se ramener systématiquement au cas précédent.

Nous parlerons de la théorie sous-jacente en Mathématiques pour l'Informatique.

Sachez juste que la récursivité est omniprésente dans OCaml.



Introduction

La programmation fonctionnelle

OCaml

- Valeurs et types
- Fonctions
- Interlude
- Compilation
- Déclaration de valeurs

Modalité de notation

Fonctions

- Application
- Construction
- Récursivité

Types de données

- Système de types
- Types
- Polymorphisme

Conclusions



Types de données

Le *système de types* d'un langage détermine quelles sont les interactions possibles entre valeurs. Il sert à détecter des erreurs lorsque vous écrivez le programme – plutôt que pendant que vous faites la démonstration en public.

Exemples Une fonction peut être appliquée à une valeur. Par contre, un entier ne peut pas être appliqué. Une fonction de type `int -> ...` peut être appliquée uniquement à des entiers, etc.

OCaml est un langage fortement, statique typé :

typiquement l'analyse des types a lieu lorsque vous tapez ou compilez le programme, avant l'exécution (à peu près comme en Java, par opposition à Python, PHP ou JavaScript)

fortement si votre programme passe l'analyse, vous pouvez être certain qu'un élément décrit comme étant de type `t` contiendra effectivement des valeurs de type `t` (comme en Java, par opposition à C ou PHP).



Limitations du système de types

Certains langages de programmation n'ont pas ou presque pas de système de types (C, Basic) ou ont un système de types incohérent (C++).

OCaml a un système de types complet et cohérent.

Dans presque tous les langages de programmation, OCaml y compris, les types de données restent imprécis. Par exemple, en OCaml, il n'existe pas de type de donnée pour décrire "tous les entiers compris entre 7 et 15" ou "toutes les chaînes de caractères".

Si vous trouvez le système de types d'OCaml trop imprécis, allez voir du côté de Coq. Vous y trouverez des types, des types de types, des types de types de types...



Types simples

En OCaml, toute valeur a un type.

Type simples :

```
entiers int (+, -, *, / ...)
nombres à virgule float (+., -., *., / ...)
booléens bool (valeurs false/true, &&, ||, not ...)
caractères char.
unité unit (valeur ()).
```

Quelques exemples

```
chaînes "Bonjour, le monde": string
fonctions fun x -> x+1: int -> int
couples 1,true: int * bool
```

Types composés

```
chaînes string (concaténation ^)
fonctions type -> type, par exemple int -> float
couples, triplets... type * type, type * type * type * type ..., par
exemple int * bool
listes type list
tableaux type array
... et tous les types que vous définirez un jour ou l'autre.
```

Q Des exemples ?

Inférence de types

Q Comment OCaml détermine-t-il le type des valeurs ?

A Par une série de déductions. Les déductions sont toujours correctes (enfin, probablement).

Corollaire Si OCaml détecte une erreur de type ou si vous constatez que le type affiché par OCaml ne correspond pas à ce que vous vouliez, c'est que votre code est erroné.

Ou que vous avez atteint les limites du système de types de OCaml, qui n'est pas parfait – mais ça ne devrait pas vous arriver avant un certain temps.

Variables de types

Les *variables de types* 'a, 'b... (prononcées α , β ...) dénotent qu'une fonction peut être appliquée à plusieurs types.

```
* f :: x -> y -> (x = y) ::
- : 'a -> 'a -> bool = <f>
```

Ce type se prononce $\forall \alpha \in \text{Types}, \alpha \rightarrow \alpha \rightarrow \text{bool}$.

En OCaml, c'est ce qu'on appelle la *polymorphisme*. Attention, le terme a une signification différente en Java.



Dernière question

Nous arrivons à la fin du chapitre.

Exercice Que font les fonctions `fst` et `snd` ?



Variante de types (suite)

```
* f :: x -> y -> (x = y) ::
- : 'a -> 'a -> bool = <f>
```

En d'autres termes, la fonction est à la fois

- ▶ une fonction de comparaison entre entiers
- ▶ une fonction de comparaison entre chaînes de caractères
- ▶ une fonction de comparaison entre listes
- ▶ ...
- ▶ mais pas une fonction de comparaison entre un entier et une chaîne de caractères !

```
* (f :: x -> y -> (x = y)) 5 10 ::
- : bool = false
* (f :: x -> y -> (x = y)) 5 "bouhu!" ::
This expression has type string but is here used with type int
```



- Introduction
- La programmation fonctionnelle
- OCaml
 - Valeurs et types
 - Fonctions
 - Interlude
 - Compilation
 - Déclaration de valeurs
- Modalité de notation
- Fonctions
 - Application
 - Construction
 - Récursivité
- Types de données
 - Système de types
 - Types
 - Polymorphisme
- Conclusions



Bilan

Pour la suite

Nous avons vu

- ▶ comment utiliser le toplevel de OCaml
- ▶ comment manipuler et déclarer des valeurs, fonctions y compris
- ▶ le système de type d'OCaml.

- ▶ définition de types
- ▶ filtrage par motifs.