Initiation au langage Ocaml

École Polytechnique – Majeure d'Informatique 1

Jean-Christophe Filliâtre – Benjamin Monate

25 septembre 2002

Support de cours

Dû à Didier Rémy

Version en ligne avec de nombreux liens et exercices :

`http://pauillac.inria.fr/~remy/poly/ocaml/`

Plan

Motivation

Pour l'enseignement

- Langages de Programmation (majeure 1)
- Modularité, Objets, et Types (majeure 1)
- Compilation (majeure 2)

\+ Certains stages d'options en informatique

Après l'X

- Connaître différents langages de programmation pour choisir le mieux approprié dans chaque situation
- Le langage Ocaml, et d'autres langages de la même famille, sont de plus en plus utilisés dans l'industrie

Avertissement

Il s'agit d'un survol du langage Ocaml, vu sous l'angle de la pratique

1. Les exemples sont simples et classiques
2. Les constructions de bases sont présentées, mais avec très peu d'exemples
3. L'ordre supérieur, le filtrage ou le polymorphisme mériteraient plus de considération et d'exercices

Les TD mettront en pratique la plupart des notions évoquées ici

1. RSA
2. Polynômes creux
3. Images et arbres
4. Parcours de labyrinthe
5. ...

Apprentissage d'un langage

Par la pratique : exercez-vous !

Par réutilisation

- Regarder les solutions d'exercices
- Réutiliser et modifier le code déjà écrit

En se référant à la définition du langage

- la documentation : manuel de référence, bibliothèques standards
- le tutorial, l'expérience des autres (FAQ)
- les bibliothèques développées par les utilisateurs

Il existe aussi plusieurs livres sur Caml

`http://caml.inria.fr/`

La petite histoire

Ocaml est le fruit de développements récents et continus :

1975 Robin Milner propose ML comme métalangage (langage de script) pour l'assistant de preuve LCF

Il devient rapidement un langage de programmation à part entière

1981 Premières implantations de ML

1985 Développement de Caml à l'INRIA, de Standard ML à Edinburgh, de "SML of New-Jersey", de Lazy ML à Chalmers, de Haskell à Glasgow, ...

1990 Implantation de Caml-Light par X. Leroy et D. Doligez

1995 Compilateur vers du code natif + système de modules

1996 Objets et classes (Ocaml)

2000 Arguments étiquetés et optionnels, variants par J. Garrigue

Domaines d'utilisation du langage Ocaml

Langage d'usage général

Domaines de prédilection

- Calcul symbolique :
  preuves mathématiques, compilation, interprétation, analyse de programmes
- Prototypage rapide
- Langage de script
- Programmation distribuée (bytecode rapide et portable)

Enseignement et recherche

- Classes préparatoires
- Utilisé dans de grandes universités (Europe, US, Japon, ...)

Industrie

- Startups, CEA, EDF, France Telecom, Simulog, ...

Quelques mots clés

Le langage Ocaml est :

- fonctionnel : les fonctions sont des valeurs de première classe, elles peuvent être retournées en résultat et passées en argument à d'autres fonctions
- à gestion mémoire automatique (un GC, comme Java)
- fortement typé : le typage garantit l'absence d'erreur (de la machine) à l'exécution
- avec synthèse de types : les types sont facultatifs et synthétisés par le système
- interactif, compilé vers du bytecode ou du code natif

De plus le langage possède :

- une couche à objets assez sophistiquée
- un système de modules très expressif

Premiers pas en Ocaml

Le mode compilé

Éditer le source

hello.ml

```
print_string "Hello World!\n";;
```

Compiler et exécuter

Sous le Shell d'Unix

```
% ocamlc -o hello hello.ml
% ./hello
Hello World!
```

Le mode interprété

```
% ocaml
```

```
Objective Caml version 3.06
```

```
# print_string "Hello World!\n";;
```

```
Hello World!
- : unit = ()
```

```
# let euro x = floor (100.0 *. x /. 6.55957) *. 0.01;;
```

```
val euro : float -> float = <fun>
```

```
# let baguette = 4.20;;
```

```
val baguette : float = 4.2
```

```
# euro baguette;;
```

```
- : float = 0.64
```

```
# exit 0;;
```

Utilisation du mode interprété

Comme une grosse calculette

Pour la mise au point des programmes

Évaluation des phrases du programme en cours de construction (utilisation avec un éditeur)

Comme langage de script

```
% ocaml bienvenue.ml
```

Plus simplement encore

bonjour

```
#!/usr/bin/ocaml
print_string "Bonjour!"; print_newline();;
```

Sous le Shell d'Unix

```
% chmod +x bonjour
% ./bonjour
```

Syntaxe

Les programmes

Un programme est une suite de phrases :

– définition de valeur	`let` $x = e$
– définition de fonction	`let` $f\ x_1\ \ldots\ x_n = e$
– définition de fonctions mutuellement récursives	`let` [`rec`] $f_1\ x_1\ \ldots = e_1\ \ldots$ [`and` $f_n\ x_n\ \ldots = e_n$]
– définition de type(s)	`type` $q_1 = t_1 \ldots$ [`and` $q_n = t_n$]
– expression	e

Les phrases se terminent par `;;` optionnel en général

```
(* Cela est un commentaire (* et ceci un commentaire à
l'intérieur d'un commentaire *) sur plusieurs lignes *)
```

Les expressions

– définition locale	`let` $x = e_1$ `in` e_2
(définition locale de fonctions mutuellement récursives)	
– fonction anonyme	`fun` x_1 ... x_n `->` e
– appel de fonction	f e_1 ... e_n
– variable	x $(M.x$ si x est défini dans le module $M)$
– valeur construite	(e_1, e_2)
dont les constantes	1, 'c', "aa"
– analyse par cas	`match` e `with` p_1 `->` $e_1 \mid \ldots \mid p_n$ `->` e_n
– boucle for	`for` $i = e_0$ `to` e_f `do` e `done`
– boucle while	`while` e_0 `do` e `done`
– conditionnelle	`if` e_1 `then` e_2 `else` e_3
– une séquence	$e; e'$
– parenthèses	(e) `begin` e `end`

Remarques

- Il n'y a ni instructions ni procédures
- Toutes les expressions retournent une valeur
- La valeur `()` de type `unit` ne porte aucune information
  (c'est l'unique valeur possible dans son type)
- L'expression e dans les boucles `for` et `while` et dans la séquence doit être de type `unit`

Types et opérations élémentaires

Types, constantes et primitives de base

unit	()	pas d'opération !
bool	true false	&& || not
char	'a' '\n' '\097'	code chr
int	1 2 3	+ - * / max_int
float	1.0 2. 3.14	+. -. *. /. cos
string	"a\tb\010c\n"	^ s.[i] s.[i] <- c

Tableaux

Les tableaux sont du type polymorphe `'a array` :

on peut créer des tableaux homogènes d'entiers, de booléens, etc.

`[| 0; 1; 3 |]`	`int array`
`[| true; false |]`	`bool array`

Les indices de tableaux varient de 0 à $n-1$ où n est la taille

Les opérations sont polymorphes :

`fun x -> x.(0)`	`'a array -> 'a`
`fun t k x -> t.(k) <- x`	`'a array -> int -> 'a -> unit`

Les tableaux sont toujours initialisés :

`Array.create`	`int -> 'a -> 'a array`

Tableaux (exemples)

```
let chr = Array.create 10 'X';;
```

```
val chr : char array = [|'X'; 'X'; 'X'; 'X'; 'X'; 'X'; 'X';
'X'; 'X'; 'X'|]
```

```
for i = 0 to 9 do
    chr.(i) <- Char.chr (Char.code '0' + i) done;;
let chiffre_trois = chr.(3);;
```

```
val chiffre_trois : char = '3'
```

Polymorphisme :

```
Array.set;;
```

```
- : 'a array -> int -> 'a -> unit = <fun>
```

```
Array.set chr;;
```

```
- : int -> char -> unit = <fun>
```

N-uplets

Ils sont hétérogènes, mais leur arité est fixée par leur type :

- une paire `(1, "toto") : int * string`
- un triplet `(1, 2, true) : int * int * bool`

Les projections sont polymorphes sur les n-uplets de même arité :

`fun (x, y, z) -> x`	`('a * 'b * 'c) -> 'a`

Les paires ne sont qu'un cas particulier de n-uplet,
pour lequel les deux projections `fst` et `snd` sont prédéfinies

`fst`	`'a * 'b -> 'a`
`snd`	`'a * 'b -> 'b`

Enregistrements

Les enregistrements

Il faut les déclarer au préalable

```
type monnaie = {nom : string; valeur : float}
```

```
let x = {nom = "euro"; valeur = 6.55957}
x.valeur
```

Champs polymorphes

```
type 'a info = {nom : string; valeur : 'a};;
```

```
fun x -> x.valeur;;
```

```
- : 'a info -> 'a
```

Les structures mutables

Dans les types enregistrements, on peut déclarer des champs mutables

```
type personne = {nom : string; mutable âge : int};;
```

permet de modifier la seconde composante, mais pas la première

```
let p = {nom = "Paul"; âge = 23};;
```

```
val p : personne = {nom="Paul"; âge=23}
```

```
let anniversaire x = x.âge <- x.âge + 1;;
```

```
val anniversaire : personne -> unit = <fun>
```

```
p.nom <- "Clovis";;
```

```
Characters 0-17 :
The label nom is not mutable
```

Les références

Le passage d'arguments se fait toujours par valeur

Il n'existe pas de constructions telles que `&x` ou `*x` en C

La valeur d'une variable n'est pas modifiable

Mais on peut modifier les champs d'un enregistrement. Le système prédéfinit

```
type 'a ref = { mutable contents : 'a }
```

En C	En Ocaml	Raccourci :
`int x = 0;`	`let x = {contents = 0}`	`let x = ref 0`
`x = x + 1;`	`x.contents <- x.contents + 1`	`x := !x + 1`

Peu de valeurs ont ainsi besoin d'être des références en Ocaml

Égalité

Égalité structurelle (notée = et <> pour la différence))

C'est l'égalité mathématique : deux valeurs sont égales
si elles ont la même structure et si leurs sous-structures respectives sont égales

```
1 = 1 && [| 2 |] = [| 2 |] && "oui" = "oui";;
```

```
- : bool = true
```

Égalité physique (notée == et != pour la différence)

C'est l'égalité de l'emplacement mémoire

```
[| 1 |] != [| 1 |] && "eq" != "eq" &&
(let x = ref 1 in x == (fun y -> y) x);;
```

```
- : bool = true
```

Pour les valeurs non allouées, les deux égalités coïncident

Entrées / Sorties

La ligne de commande

Les arguments passés à un exécutable

Ils sont placés dans le tableau `Sys.argv : string array`

Le nom de la commande est compté

Exemple La commande Unix `echo`

echo.ml

```
for i = 1 to Array.length Sys.argv - 1
  do print_string Sys.argv.(i) ; print_char ' ' done ;
print_newline();;
```

Usage avancé

La bibliothèque `Arg` permet de lire ces arguments plus facilement

Les entrées-sorties

Canaux prédéfinis

```
 stdin : in_channel
stdout : out_channel
stderr : out_channel
```

Ouverture de canaux

```
 open_out : string -> out_channel
  open_in : string -> in_channel
close_out : out_channel -> unit
```

Lecture sur stdin

```
read_line : unit -> string
 read_int : unit -> int
```

Écriture sur stdout

```
print_string : string -> unit
   print_int : int -> unit
```

Les entrées sorties avancées

- Voir la bibliothèque de base, `Pervasives`
- Voir la bibliothèque `Printf` par exemple, on écrira comme en C :

```
Printf.printf "mois de %s, année %d\n" "Janvier" 1999;;
```

Les fonctions

Les fonctions

Les arguments sont passés sans parenthèses

Écriture recommandée

```
let milieu x y = (x + y) / 2
```

```
val milieu :
int -> int -> int = <fun>
```

```
milieu 5 11;;
```

Mauvaise écriture

```
let milieu (x, y) = (x + y) / 2
```

```
val milieu :
int * int -> int = <fun>
```

```
milieu (5, 11);;
```

Fonctions récursives

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

… et mutuellement récursives

```
let rec ping n = if n > 0 then pong (n - 1) else "ping"
and pong n = if n > 0 then ping (n - 1) else "pong";;
```

Applications

Quel est l'intrus dans

```
f x y      (f x) y      f (x y)      (f(x))(y)
```

Application partielle

On peut ne passer qu'un argument a une fonction qui en attend deux

```
let plus x y = x + y;;
```

```
val plus : int -> int -> int = <fun>
```

```
let incr2 = plus 2;;
```

```
val incr2 : int -> int = <fun>
```

Remarque : on aurait pu écrire :

```
let plus = ( + );;
```

Abstraction

Si `f` est une fonction de type `int -> int -> int`, il n'y a pas de différence entre

```
f        fun x -> f x        fun x y -> f x y
```

(en l'absence d'effets de bord)

L'ordre supérieur

Les fonctions sont des valeurs

- En particulier, elles peuvent être retournées en résultat et passées en argument à d'autres fonctions
- Les fonctions se manipulent comme en mathématiques

La composition de deux fonctions est une fonction

```
let compose f g = fun x -> f (g x);;
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Fonctions anonymes

La fonction `(fun x -> f (g x))` est anonyme. Les définitions

```
let compose = fun f -> fun g -> fun x -> f (g x)
let compose f g x = f (g x);;
```

sont toutes équivalentes ainsi que leurs formes intermédiaires

La puissance des fonctions

Fonction puissance

```
let rec power f n =
  if n = 0 then (fun x -> x)
  else compose f (power f (n-1));;
```

```
val power : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

Dérivation

```
let derivative dx f = fun x -> (f(x +. dx) -. f(x)) /. dx;;
```

```
val derivative : float -> (float -> float) -> float -> float
= <fun>
```

La puissance de la dérivée

```
let pi = 4.0 *. atan 1.0;;
let sin''' = (power (derivative 1e-5) 3) sin in sin''' pi;;
```

```
- : float = 0.999999
```

à la précision des calculs près...

Les types somme et le filtrage

Les types somme et le filtrage

Les définitions de types somme

Analogues aux définitions de types d'enregistrements.

Exemple : les valeurs booléennes sont éléments d'un type énuméré :

```
type booléen = Vrai | Faux;;
```

Les constructeurs sont toujours capitalisés

Le filtrage permet d'examiner les valeurs d'un type somme

```
let int_of_booléen x =
  match x with Vrai -> 0 | Faux -> 1;;
```

Raccourci (notation équivalente)

```
let int_of_booléen = function Vrai -> 0 | Faux -> 1;;
```

Le type option

Les références sont toujours initialisées en Ocaml. Quelques fois il est nécessaire de créer des références qui seront définies plus tard

Ce traitement doit être explicite

Prédéfini par le système

```
type 'a option = Some of 'a | None
```

```
let statut = ref None;;
```

Plus tard, on pourra définir la valeur :

```
statut := Some 3;;
```

La lecture du statut doit aussi considérer le cas où celui-ci est indéfini

```
match !statut with Some x -> x | None -> 0;;
```

Listes

Les listes sont simplement un type somme récursif

Types récursifs

```
type 'a liste =
     Vide
   | Cellule of 'a * 'a liste;;
```

Définition alternative (utilisant un enregistrement)

```
type 'a liste = Vide | Cellule of 'a cell
and 'a cell = { tête : 'a; reste : 'a liste };;
```

Listes modifiables en place (les champs sont mutables)

```
type 'a liste = Vide | Cellule of 'a cell
and 'a cell = { mutable hd : 'a; mutable tl : 'a liste };;
```

Les vraies listes (bibliothèque `List`)

Ce sont les listes qu'il faut utiliser !

Le système prédéfinit :

```
type 'a list = [] | (::) of 'a * 'a list
```

Raccourci Les deux formes suivantes sont équivalentes :

```
let l = 1 :: 2 :: 3 :: 4 :: [];;
let l = [1 ; 2 ; 3 ; 4];;
```

Exemple : la fonction inverse

Elle utilise un argument supplémentaire pour accumuler la liste déjà retournée :

```
let rec inverse_aux déjà_retournée à_retourner =
  match à_retourner with
  | [] -> déjà_retournée
  | h :: t -> inverse_aux (h :: déjà_retournée) t;;
let inverse l = inverse_aux [] l;;
```

Le jeu de carte

Il combine types somme et types enregistrement :

```
type carte = Carte of ordinaire | Joker
and ordinaire = { couleur : couleur ; figure : figure }
and couleur = Coeur | Carreau | Pique | Trèfle
and figure = As | Roi | Reine | Valet | Simple of int;;
```

```
let valet_de_pique =
  Carte { figure = Valet ; couleur = Pique };;
```

```
val valet_de_pique : carte =
  Carte { couleur = Pique ; figure = Valet }
```

```
let carte n s = Carte { figure = n ; couleur = s }
```

```
val carte : figure -> couleur -> carte = <fun>
```

```
let roi = carte Roi;;
```

```
val roi : couleur -> carte = <fun>
```

Filtrage en profondeur

```
let valeur = function
  | Carte { figure = As } -> 14
  | Carte { figure = Roi } -> 13
  | Carte { figure = Reine } -> 12
  | Carte { figure = Valet } -> 11
  | Carte { figure = Simple k } -> k
  | Joker -> 0;;
```

Un motif peut capturer plusieurs cas

```
let petite = function
  | Carte { figure = Simple _ } -> true
  | _ -> false;;
```

Sémantique du filtrage

Lorsqu'un argument est passé à un ensemble de clauses

$$p_1 \texttt{ -> } e_1 \mid \ldots \mid p_n \texttt{ -> } e_n$$

- la première clause qui filtre l'argument est exécutée, les autres ignorées
- si aucune clause ne filtre l'argument, une exception est levée

Si la définition est incomplète, ceci est indiqué à la compilation

```
let simple = function Carte {figure = Simple k} -> k;;
```

```
Characters 13-52 :
Warning : this pattern-matching is not exhaustive.
Here is an example of a value that is not matched :
Joker
val simple : carte -> int = <fun>
```

Usage Il est préférable d'éviter les définitions incomplètes

Les motifs (patterns)

Ils peuvent

- être emboîtés, contenir des produits et des sommes
- ne pas énumérer tous les champs des enregistrements
- n'imposer aucune contrainte (représentés par le caractère _)
- lier une valeur intermédiaire avec mot clé `as`

```
function Carte ({ figure = Simple _ } as z) -> z.couleur;;
```

```
Warning : this pattern-matching is not exhaustive.
Joker
- : carte -> couleur = <fun>
```

Une variable ne peut pas être liée deux fois dans le même motif

```
function Carte z, z -> true;;
```

```
Characters 18-19 :
This variable is bound several times in this matching
```