

PF chapitre 1 : Introduction à OCaml

Expressions, liaisons, types composés, récursivité

Jean-François Monin & Pierre Corbineau



Plan

Présentation du cours

Motivations

Programmation impérative

Programmation fonctionnelle

Expressions et types de base (nouveau 0)

Liaison d'un identificateur (nouveau 1 et 2)

Types composés (nouveau 3)

Sommes et filtrage (nouveau 4)

Types somme récursifs

Qui ?

Enseignants

- ▶ Cours : Pierre Corbineau
- ▶ TD : Pierre Corbineau & Erwan Jahier
- ▶ TP : Erwan Jahier & Martin Bodin & Pierre Corbineau

Supports

Matériel

- ▶ Linux + OCaml + emacs + Tuareg
- ▶ Pages Web (outils, planning, documents...)
<https://ltpf.gricad-pages.univ-grenoble-alpes.fr/>

Ressources recommandées

- ▶ <http://ocaml.org/>
- ▶ Demander de l'aide aux enseignants
Mettre tous les enseignants en copies de vos questions.

Bibliographie

- ▶ Plus de 15 bonnes références sur <https://ocaml.org/books>
- ▶ Approche fonctionnelle de la programmation, Michel Mauny et Guy Cousineau, Ediscience international, 1995
- ▶ Attention : on suit une route un peu particulière, en lien avec LT (Langages et Traducteurs)

UE : matière combinée avec LT

Évaluation par compétences

travail personnel + contrôles (détails à venir)

(Mini-)projet final en fin de semestre

Oraux **individuels**

Objectifs du cours

Compétences

- ▶ Utiliser le paradigme fonctionnel
→ les **lambdas** présents en Java, Python, C++, Javascript...
- ▶ Typage
- ▶ Utiliser la récursivité et les structures de données correspondantes
- ▶ Raisonnement par *récence structurelle*
- ▶ Concevoir des algorithmes efficaces et corrects
- ▶ Analyse lexicale et syntaxique

Outils

Langage

- ▶ Programmation fonctionnelle en Objective Caml
(Caml = Categorical Abstract Machine Language)
- ▶ Liaison avec LT (Coq + sémantique des langages de programmation) :
 - ▶ Coq intègre un langage fonctionnel
 - ▶ preuves mathématiques de programmes OCaml assistées par ordinateur
- ▶ Historique : <https://ocaml.org/about#history>

Plan

Présentation du cours

Motivations

Programmation impérative

Programmation fonctionnelle

Expressions et types de base (nouveau 0)

Liaison d'un identificateur (nouveau 1 et 2)

Types composés (nouveau 3)

Sommes et filtrage (nouveau 4)

Types somme récursifs

Plan

Présentation du cours

Motivations

Programmation impérative

Programmation fonctionnelle

Expressions et types de base (nouveau 0)

Liaison d'un identificateur (nouveau 1 et 2)

Types composés (nouveau 3)

Sommes et filtrage (nouveau 4)

Types somme récursifs

Différents paradigmes de programmation

1. Impératif
Fortran (1954), COBOL (1960), C (1970)...
2. Fonctionnel
Lisp (1959), Haskell (1990), CAML/OCaml (1987)...
3. Logique
Prolog (1972)

En réalité

Les langages modernes **combinent** plusieurs paradigmes

Ex 1 : **lambdas** en Python, C++, Java etc.

Ex 2 : traits impératifs et objets en OCaml ; **monades**

Programmation impérative

Centrée sur la notion d'affectation

$$x := E$$

- ▶ x est un *emplacement mémoire*
- ▶ E est une *expression* :
sa valeur v est **temporairement** associée à x .

x est aussi traditionnellement appelée une *variable*.

Remarque : calcul de E

- ▶ idéalement, expression pure : facile à gérer
- ▶ parfois, effets de bord : pénible !

$$x ++ = x ++ + x ++$$

(des versions différentes du compilateur C donnent des résultats différents)

Modèle de calcul

État

Définit les données représentées en mémoire à un instant donné.

L'état peut être très complexe, utiliser des **pointeurs**

Les ingrédients des algorithmes

- ▶ Une instruction indique une transition d'état.
- ▶ Combinaisons d'instructions (if, boucles...)

Pour concevoir et comprendre un programme

Il faut étudier l'effet de toutes les instructions sur toutes les parties de l'état dans toutes les situations atteignables.

Problème : la valeur des variables est difficile à saisir (savonnette)

Concevoir et comprendre un programme

Raisonner sur un programme impératif est compliqué

Il faut étudier l'effet de toutes les instructions sur toutes les parties de l'état dans toutes les situations atteignables.

Problème : les valeur des variables est difficile à saisir (savonnette)

Exemple

$x := 3$

Donne $x = 3$

$x := x + 1$

Donne quoi ?

On peut survivre (ex. logique de Hoare)

Mais c'est compliqué. Il est plus simple d'**éviter les états mutables**.

Programmation fonctionnelle

Centrée sur la notion d'expression

Toute expression a une *valeur* et un *type*

- ▶ déterminés une fois pour toutes
- ▶ indépendants de l'ordre d'évaluation

Description d'un algorithme

- ▶ Au moyen d'appels de **fonctions** qui sont **gérées comme des données ordinaires**
- ▶ Une fonction est souvent décrite elle-même à partir de fonctions (analyse descendante).
- ▶ Utilisation intensive de la **récursivité**
- ▶ Manipulation aisée des données grâce au **filtrage** (English : **pattern-matching**)

Concevoir et comprendre un programme fonctionnel

Raisonnement facilité

- ▶ On ne se préoccupe que de la *valeur* et du *type*.
- ▶ Raisonnement *par cas* sur les différentes valeurs possibles
+ raisonnement *par récurrence*.
- ▶ Raisonnement *équationnel* :
toute expression peut-être remplacée par une expression égale.

Passe plus facilement à l'échelle

Y compris pour des preuves formelles en présence de données complexes

Mythes et réalités sur la programmation fonctionnelle

Ce que l'on évite

- ▶ Pas de contrôle : chemins d'exécution indifférents
- ▶ Pas de pointeurs : pas de corruption des valeurs

Ce que l'on ne perd pas

- ▶ Efficacité : **Interpréteur**, compilateur → code-octet ou **natif**
- ▶ Possibilité d'utiliser des traits impératifs (à bon escient)

Ce que l'on gagne

- ▶ Taille du code source typiquement 1/10 du code C équivalent
- ▶ Gestion mémoire automatique
inventée pour LISP (1959),
popularisée dans les années 90 par... Java !

Quelques applications

- ▶ Domaine de prédilection : traitement symbolique
 - ▶ analyse de programmes : ASTREE
 - ▶ aide à la démonstration : Coq
 - ▶ compilation
- ▶ Négoce électronique (trading) Jane Street
- ▶ Synchronisateur de fichiers **Unison**
- ▶ Beaucoup d'autres

Plan

Présentation du cours

Motivations

Programmation impérative

Programmation fonctionnelle

Expressions et types de base (nouveaué 0)

Liaison d'un identificateur (nouveaués 1 et 2)

Types composés (nouveaué 3)

Sommes et filtrage (nouveaué 4)

Types somme récursifs

Types de base

- ▶ `int` 42 -12 opérateurs : + - * / mod
- ▶ `float` -3.1 0. 1.3e-4 opérateurs : +. -. *. /.
- ▶ `bool` true false opérateurs : && || not

- ▶ et aussi `char`, `string`... cf doc. OCaml

Types de base

- ▶ `int` 42 -12 opérateurs : + - * / mod
- ▶ `float` -3.1 0. 1.3e-4 opérateurs : +. -. *. /.
- ▶ `bool` true false opérateurs : && || not

- ▶ et aussi `char`, `string`... cf doc. OCaml

Expression = assemblage de constantes et d'opérateurs
qui respecte les **contraintes de type**

Chaque opérateur est typé (arguments attendus et valeur calculée).

Expressions à base d'opérateurs

- ▶ $2 + 3$
- ▶ $2. +. 3.$
- ▶ $(2 + 5) * 3$
- ▶ $2 = 5$
- ▶ $2 < 5$
- ▶ $\text{true} \ \&\& \ \text{false}$
- ▶ $(2 < 5) \ \&\& \ 2 = 5$

Expression conditionnelle (nouveau 0)

- ▶ `if b then e1 else e2` NB. en langage C `b ? e1 : e2`
- ▶ Pas une instruction mais une *expression*
dont la valeur dépend de *b*

Exemple : `(if 5 < 9 then 21 else 3) * 2;;`

Expression conditionnelle (nouveau 0)

- ▶ `if b then e1 else e2` NB. en langage C `b ? e1 : e2`
- ▶ Pas une instruction mais une *expression*
dont la valeur dépend de *b*

Exemple : `(if 5 < 9 then 21 else 3) * 2;;`

- ▶ `(if 5 < 9 then 21 else 3) * 2`

Expression conditionnelle (nouveau 0)

- ▶ `if b then e1 else e2` NB. en langage C `b ? e1 : e2`
- ▶ Pas une instruction mais une *expression*
dont la valeur dépend de *b*

Exemple : `(if 5 < 9 then 21 else 3) * 2;;`

- ▶ `(if 5 < 9 then 21 else 3) * 2`
- ▶ `(if true then 21 else 3) * 2`

Expression conditionnelle (nouveau 0)

- ▶ `if b then e1 else e2` NB. en langage C `b ? e1 : e2`
- ▶ Pas une instruction mais une *expression*
dont la valeur dépend de *b*

Exemple : `(if 5 < 9 then 21 else 3) * 2;;`

- ▶ `(if 5 < 9 then 21 else 3) * 2`
- ▶ `(if true then 21 else 3) * 2`
- ▶ `21 * 2`

Expression conditionnelle (nouveau 0)

- ▶ `if b then e1 else e2` NB. en langage C `b ? e1 : e2`
- ▶ Pas une instruction mais une *expression*
dont la valeur dépend de *b*

Exemple : `(if 5 < 9 then 21 else 3) * 2;;`

- ▶ `(if 5 < 9 then 21 else 3) * 2`
- ▶ `(if true then 21 else 3) * 2`
- ▶ `21 * 2`
- ▶ `42`

Expression conditionnelle (nouveau 0)

- ▶ `if b then e1 else e2` NB. en langage C `b ? e1 : e2`
- ▶ Pas une instruction mais une *expression* dont la valeur dépend de *b*

Exemple : `(if 5 < 9 then 21 else 3) * 2;;`

- ▶ `(if 5 < 9 then 21 else 3) * 2`
- ▶ `(if true then 21 else 3) * 2`
- ▶ `21 * 2`
- ▶ `42`

Les opérateurs booléens sont **paresseux** :

Dans `e1 && e2` l'expression `e2` n'est pas évaluée si `e1` vaut `false`

Dans `e1 || e2` l'expression `e2` n'est pas évaluée si `e1` vaut `vrai`

Plan

Présentation du cours

Motivations

Programmation impérative

Programmation fonctionnelle

Expressions et types de base (nouveau^té 0)

Liaison d'un identificateur (nouveau^ts 1 et 2)

Types composés (nouveau^té 3)

Sommes et filtrage (nouveau^té 4)

Types somme récursifs

Identificateurs, liaison (nouveau 1)

Liaison d'un identificateur : **let** *ident* = *expr*

let lie (ou associe)

- ▶ un *identificateur*
- ▶ à une *valeur* calculée par une *expression* ;
- ▶ cette association est *définitive*

```
# let x = 52 ;;
```

```
val x : int = 52
```

```
# let y = x - 10 ;;
```

```
val y : int = ?
```

Identificateurs, liaison (nouveau 1)

Liaison d'un identificateur : **let** *ident* = *expr*

let lie (ou associe)

- ▶ un *identificateur*
- ▶ à une *valeur* calculée par une *expression* ;
- ▶ cette association est *définitive*

```
# let x = 52 ;;
```

```
val x : int = 52
```

```
# let y = x - 10 ;;
```

```
val y : int = 42
```

Liaison d'un identificateur : **let** ident = expr

- ▶ un *identificateur*
- ▶ à une *valeur* calculée par une *expression* ;
- ▶ cette association est *définitive*

```
val x : int = 52
```

```
val y : int = 42
```

- ▶ x est associé à la valeur 52.
- ▶ donc $x - 10$ a la valeur calculée par $52 - 10$, c.à.d. 42

Pas d'état, pas de variables

Redéfinition d'un identificateur

De quel **x** parle-t-on ?

Exemple

```
# let x = 52 ;;
```

```
val x : int = 52
```

```
# let y = x - 10 ;;
```

```
val y : int = 42
```

Redéfinition d'un identificateur

De quel **x** parle-t-on ?

Exemple

```
# let x = 52 ;;  
val x : int = 52  
# let y = x - 10 ;;  
val y : int = 42  
# let x = 11 ;;  
val x : int = 11  
# 2 * x + y ;;  
- : int = ?
```

Redéfinition d'un identificateur

De quel **x** parle-t-on ?

Exemple

```
# let x = 52 ;;
```

```
val x : int = 52
```

```
# let y = x - 10 ;;
```

```
val y : int = 42
```

```
# let x = 11 ;;
```

```
val x : int = 11
```

```
# 2 * x + y ;;
```

```
- : int = ? 146 ou 23 ou 64
```

Redéfinition d'un identificateur

De quel **x** parle-t-on ?

Exemple

```
# let x = 52 ;;
```

```
val x : int = 52
```

```
# let y = x - 10 ;;
```

```
val y : int = 42
```

```
# let x = 11 ;;
```

```
val x : int = 11
```

```
# 2 * x + y ;;
```

```
- : int = ? 146 ou 23 ou 64
```

On peut même écrire **let** $x = x + 1$

mais signification différente de l'affectation en C !

Let local

Le **let** précédent correspond à une définition **globale** qui lie un nom à la valeur d'une expression

let *a* = *expr1*

expr2

⇒ *a* reste disponible

Définition locale

Une expression peut elle même commencer par une définition locale : identificateur visible uniquement dans cette expression

let *a* = *expr1* **in** *expr2*

⇒ *expr1* écrite et **évaluée une seule fois**

⇒ Ce *a* est disponible seulement dans *expr2*, puis est **oublié**
évite des collisions

⇒ Nb : **let** *a* = *expr1* **in** *expr2* est une expression

Définition d'une fonction (avant goût)

C'est un **let** ordinaire (non-nouveauté 2)

let *f* = ... (* la fonction qui ajoute 7 *)

Définition d'une fonction (avant goût)

C'est un **let** ordinaire (non-nouveauté 2)

let *f* = ... (* la fonction qui ajoute 7 *)

Utilisation : application d'une fonction

let huit = *f* 1

Nouveauté 2.1 : pas de parenthèses pour *f* appliqué à un argument

Définition d'une fonction (avant goût)

C'est un **let** ordinaire (non-nouveauté 2)

let **f** = **fun** **x** → 7 + **x** (* la fonction qui ajoute 7 *)

Utilisation : application d'une fonction

let huit = **f** 1

Nouveauté 2.1 : pas de parenthèses pour **f** appliqué à un argument

Nouveauté 2.2 : expression d'une valeur fonctionnelle

Définition d'une fonction (avant goût)

C'est un **let** ordinaire (non-nouveauté 2)

let **f** = **fun** **x** → 7 + **x** (* la fonction qui ajoute 7 *)

Utilisation : application d'une fonction

let huit = **f** 1

Nouveauté 2.1 : pas de parenthèses pour **f** appliqué à un argument

Nouveauté 2.2 : expression d'une valeur fonctionnelle

Nouveauté 2.3 : une valeur fonctionnelle peut résulter d'un calcul

Exemple : “le troisième élément de cette liste de fonctions”

Définition d'une fonction (avant goût)

C'est un **let** ordinaire (non-nouveauté 2)

let *f* = **fun** *x* → 7 + *x* (* la fonction qui ajoute 7 *)

Utilisation : application d'une fonction

let huit = *f* 1

Nouveauté 2.1 : pas de parenthèses pour *f* appliqué à un argument

Nouveauté 2.2 : expression d'une valeur fonctionnelle

Nouveauté 2.3 : une valeur fonctionnelle peut résulter d'un calcul

Exemple : “le troisième élément de cette liste de fonctions”

NOUVEAUTÉ 2

Les fonctions sont gérées comme des expressions ordinaires

Plan

Présentation du cours

Motivations

Programmation impérative

Programmation fonctionnelle

Expressions et types de base (nouveau 0)

Liaison d'un identificateur (nouveau 1 et 2)

Types composés (nouveau 3)

Sommes et filtrage (nouveau 4)

Types somme récursifs

Pourquoi composer de nouveaux types ?

- ▶ Tous les langages (même fonctionnels !) offrent à peu près les mêmes **actions** algorithmiques (branchement, itération...).
- ▶ L'expressivité d'un langage vient plutôt des **structures de données** qu'il permet de représenter et des facilités de manipulation qu'il offre.
- ▶ Les algorithmes récursifs sont naturellement adaptés à la manipulation de **structures récursives**.

Deux grands procédés de composition

- ▶ produits de types
existent nativement dans tous les langages de programmation
- ▶ **sommes** de types (nouveau 3)
existent nativement en programmation fonctionnelle typée
Un des points essentiels de PF

Produits : n -uplets

- ▶ Définition d'un type *produit* :
type complexe = float * float ;;
- ▶ Valeurs notées entre parenthèses :
let e_i_pi_sur_2 = (0., 1.) ;;
- ▶ Accéder aux composantes du n -uplet :
let (re, im) = e_i_pi_sur_2 **in** sqrt (re*.re +. im*.im) ;;

Problème du mélange propre

Essence du typage

Les arguments fournis à une fonction doivent avoir un sens :
ne pas mélanger entiers, chaînes, booléens, n-uplets, fonctions...

Mais on a souvent besoin de mélanger

- ▶ protocoles
- ▶ lexèmes, structures grammaticales

Problème du mélange propre

Essence du typage

Les arguments fournis à une fonction doivent avoir un sens :
ne pas mélanger entiers, chaînes, booléens, n-uplets, fonctions...

Mais on a souvent besoin de mélanger

- ▶ protocoles
- ▶ lexèmes, structures grammaticales

Fausse bonne idée : union (casse le typage)

Solution compliquée : types avec discriminants

*Bonne solution : union **disjointe** = **somme***

Types somme (nouveau 3)

Exemple

```
type nombre =
```

```
  Ent of int | Reel of float | Cplx of float × float
```

Ent, Reel, Cplx sont les *constructeurs* du type.

Valeurs : Ent(3), Reel(2.5), Cplx(0., 1.)

Étant donnée une valeur, l'examen du constructeur permet de déterminer le cas d'origine avec le type contenu (pour nombre : int ou bien float ou bien float × float).

Syntaxe OCaml : les constructeurs débutent par une majuscule.

Sommes : exemples dégénérés

Énumération

```
# type couleur = Rouge | Vert | Bleu | Jaune  
# type extremité = top | bottom ;;
```

Sommes : exemples dégénérés

Énumération

```
# type couleur = Rouge | Vert | Bleu | Jaune
```

```
# type extremite = top | bottom;;
```

Syntax error

```
# type extremite = Top | Bottom;;
```

Sommes : exemples dégénérés

Énumération

```
# type couleur = Rouge | Vert | Bleu | Jaune
```

```
# type extremité = top | bottom;;
```

Syntax error

```
# type extremité = Top | Bottom;;
```

Booléens

```
# type bool = true | false
```

NB. Exception à la règle : ce sont les seuls constructeurs qui débutent par une minuscule.

Plan

Présentation du cours

Motivations

Programmation impérative

Programmation fonctionnelle

Expressions et types de base (nouveau 0)

Liaison d'un identificateur (nouveau 1 et 2)

Types composés (nouveau 3)

Sommes et filtrage (nouveau 4)

Types somme récursifs

Exemples de type somme

```
type legume = Haricot | Carotte | Courge
type fleur = Rose | Hortensia
type fruit = Poire | Banane
type couleur = Rouge | Jaune | Blanc
type plante =
  | Leg of legume
  | Flr of fleur * couleur
  | Arb of fruit
type objet =
  | Plt of plante
  | ...
```

Exemples de type somme

```
type legume = Haricot | Carotte | Courge
type fleur = Rose | Hortensia
type fruit = Poire | Banane
type couleur = Rouge | Jaune | Blanc
type plante =
  | Leg of legume
  | Flr of fleur * couleur
  | Arb of fruit
type objet =
  | Plt of plante
  | ...
```

Représentation graphique (au tableau)

Arbres

Type somme et filtrage (nouveau 4)

Un type définit exhaustivement les valeurs possibles après réduction
Le filtrage couvre toutes ces valeurs par des motifs

Motif

Arbre à trous, où chaque trou représente un sous-arbre quelconque
(du type approprié)

```
match o with
| Plt (Flr (f, c)) -> ... f ... c ...
| ...
```

Représentation graphique

Motif de filtrage arborescent

Exemple : booléens

```
match b with true → e1 | false → e2
```

est juste une autre syntaxe pour

```
if b then e1 else e2
```

Plan

Présentation du cours

Motivations

Programmation impérative

Programmation fonctionnelle

Expressions et types de base (nouveau 0)

Liaison d'un identificateur (nouveau 1 et 2)

Types composés (nouveau 3)

Sommes et filtrage (nouveau 4)

Types somme récursifs

Type somme récursif

```
type arbent =  
  | FV  
  | N of arbent * int * arbent
```

```
match a with  
  | FV → true  
  | N (g, x, d) → false
```

```
match a with  
  | FV → 0  
  | N (g, x, d) → taille g + 1 + taille d
```

Type somme rékursif

```
type arbent =  
  | FV  
  | N of arbent * int * arbent
```

```
match a with  
  | FV → true  
  | N (g, x, d) → false
```

```
match a with  
  | FV → 0  
  | N (g, x, d) → taille g + 1 + taille d
```

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