

Programmation Fonctionnelle (PF)

INFO4

Chapître 1 : Introduction à OCaml

Expressions, types composés, récursivité

Jean-François Monin



Qui ?

Enseignants

- ▶ Cours : Jean-François Monin
- ▶ TD, TP :
Erwan Jahier & Jean-François Monin (& Benjamin Wack)
- ▶ (échanges ponctuels)

Les moyens

Matériel

- ▶ Linux + OCaml + Tuareg + emacs
- ▶ Pages Web (outils, planning, documents...)
 - `https://ltpf.gricad-pages.univ-grenoble-alpes.fr/`
 - `http://www-verimag.imag.fr/~monin/Enseignement/LTPF/PF`

Ressources recommandées

- ▶ `http://ocaml.org/`
- ▶ Demander de l'aide aux enseignants

Bibliographie

- ▶ Xavier Leroy et al. Objective Caml manual <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>
- ▶ The book Developing Applications With Objective Caml (by Emmanuel Chailloux, Pascal Manoury and Bruno Pagano)
- ▶ Approche fonctionnelle de la programmation, Michel Mauny , Guy Cousineau
- ▶ Pierre Weis and Xavier Leroy. Le langage Caml. Dunod, 1999.
- ▶ Apprentissage de la programmation avec OCaml Catherine Dubois & Valérie Ménissier-Morain
- ▶ Philippe Nardel. Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml. Vuibert, 2005
- ▶ Louis Gacogne. Programmation par l'exemple en Caml. Ellipse, 2004

UE : matière combinée avec LT

Évaluation par compétences

travail personnel + contrôles (détails à venir)

(Mini-)projet final en fin de semestre

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 *récurrence structurelle*.
- ▶ Concevoir des algorithmes efficaces et corrects
- ▶ Analyse syntaxique
- ▶ Quelques calculs de *complexité*.

Outils

Langage

- ▶ Programmation fonctionnelle en Objective Caml.
(Caml = Categorical Abstract Machine Language)
- ▶ Liaison avec LT : Coq + sémantique des langages de programmation

Plan

Présentation du cours

Motivations

Expressions et types de base

Définition d'un identificateur

Types composés

Sommes et filtrage

Types somme rékursifs

Plan

Présentation du cours

Motivations

Expressions et types de base

Définition d'un identificateur

Types composés

Sommes et filtrage

Types somme rékursifs

Différents paradigmes de programmation

1. Impératif

Fortran (1954), COBOL (1960), C (1970)...

2. Fonctionnel

Lisp (1959), Haskell (1990), Objective CAML (1996)...

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 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 : les variables sont de véritables savonnettes.

Comprendre un programme

Pour raisonner sur un programme

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

Problème : les variables sont de véritables savonnettes.

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.

Programmation fonctionnelle

Centrée sur la notion d'expression

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

- ▶ Déterminés une fois pour toutes.
- ▶ La valeur ne dépend pas de l'ordre d'évaluation.

Description d'un algorithme

- ▶ Un algorithme est une *fonction* d'un ensemble de valeurs dans un autre.
- ▶ 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**)

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.

Passer 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 typiquement 1/10 du code C équivalent
- ▶ Gestion mémoire automatique

Prix à payer

Effort

- ▶ Changement de perspective
- ▶ Moins adapté à l'embarqué ou à la programmation système de bas niveau (un peu comme Java)

Support

- ▶ Communauté très active dans le « libre »
- ▶ Bibliothèques en quantité moyenne mais d'excellente qualité

Un « vrai » langage

Caml et ses avatars développés depuis $\simeq 35$ ans, principalement dans des laboratoires d'informatique français.

Applications

- ▶ Domaine de prédilection : traitement symbolique
 - ▶ analyse de programmes : ASTREE (CNRS & INRIA)
 - ▶ aide à la démonstration : Coq (INRIA)
 - ▶ compilation
- ▶ Programmation système : SLAM (Microsoft)
- ▶ Programmation réseau : MLdonkey (peer to peer multiplateformes, premier client open source d'eDonkey)
- ▶ Milieux scientifiques, domaine bancaire (Jane Street)...

Un exemple : **Unison** est passé de Java à OCaml en cours de développement pour gagner en robustesse.

Langage très influent

- ▶ Java
- ▶ C++
- ▶ Rust
- ▶ Scala
- ▶ ...

Caractéristiques de OCaml

Expressivité

- ▶ Typage statique avec inférence de types et polymorphisme
- ▶ Types définissables et filtrage
- ▶ Gestions des exceptions

Développement

- ▶ Gestion de la mémoire
- ▶ Modules paramétrables (foncteur)
- ▶ Compilateur natif et bytecode
- ▶ Système de classe évolué (objets)
- ▶ Open source

Programme du Cours

1. Bases de OCaml
2. Programmation et preuve par récurrence structurelle
3. Analyse syntaxique
4. Modularité
5. Mécanismes de typage
6. Lambda Calcul
7. Structures et traits impératifs

Plan

Présentation du cours

Motivations

Expressions et types de base

Définition d'un identificateur

Types composés

Sommes et filtrage

Types somme rékursifs

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

- ▶ “**if** b **then** e_1 **else** e_2 ”
- ▶ **Pas une instruction** mais une *expression* dont la valeur dépend de b

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

Expression conditionnelle

- ▶ “**if** b **then** e_1 **else** e_2 ”
- ▶ **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

- ▶ “**if** b **then** e_1 **else** e_2 ”
- ▶ **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

- ▶ “**if** b **then** e_1 **else** e_2 ”
- ▶ **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

- ▶ “**if** b **then** e_1 **else** e_2 ”
- ▶ **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

- ▶ “**if** b **then** e_1 **else** e_2 ”
- ▶ **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 $e_1 \ \&\& \ e_2$ l'expression e_2 n'est pas évaluée si e_1 vaut false

Dans $e_1 \ || \ e_2$ l'expression e_2 n'est pas évaluée si e_1 vaut vrai

Plan

Présentation du cours

Motivations

Expressions et types de base

Définition d'un identificateur

Types composés

Sommes et filtrage

Types somme rékursifs

Identificateurs

Définition d'un identificateur : **let**

let associe un *identificateur* à une *valeur* calculée par une *expression*; cette association est *définitive*

```
# let x = 53;;
```

```
val x : int = 53
```

```
# let y = x - 11;;
```

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

Identificateurs

Définition d'un identificateur : **let**

let associe un *identificateur* à une *valeur* calculée par une *expression*; cette association est *définitive*

```
# let x = 53;;
```

```
val x : int = 53
```

```
# let y = x - 11;;
```

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

Identificateurs

Définition d'un identificateur : **let**

let associe un *identificateur* à une *valeur* calculée par une *expression*; cette association est *définitive*

```
# let x = 53;;
```

```
val x : int = 53
```

```
# let y = x - 11;;
```

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

Explications

- ▶ **let** y = x - 11;;
- ▶ x est associé à la valeur 53.
- ▶ donc x - 11 a la valeur de 53 - 11.
- ▶ **val** y : **int** = 42

Pas d'état, pas de variables

Redéfinition d'un identificateur

Un identificateur est un nom donné à une valeur.

Exemple

```
# let x = 53 ;;  
val x : int = 53  
# let y = x - 11 ;;  
val y : int = 42  
# let x = 2 ;;  
val x : int = 2  
# y ;;  
- : int = ?
```

Redéfinition d'un identificateur

Un identificateur est un nom donné à une valeur.

Exemple

```
# let x = 53 ;;  
val x : int = 53  
# let y = x - 11 ;;  
val y : int = 42  
# let x = 2 ;;  
val x : int = 2  
# y ;;  
- : int = ? -9 ou 42
```

Redéfinition d'un identificateur

Un identificateur est un nom donné à une valeur.

Exemple

```
# let x = 53 ;;  
val x : int = 53  
# let y = x - 11 ;;  
val y : int = 42  
# let x = 2 ;;  
val x : int = 2  
# y ;;  
- : int = ? -9 ou 42
```

Redéfinition d'un identificateur

Un identificateur est un nom donné à une valeur.

Exemple

```
# let x = 53 ;;  
val x : int = 53  
# let y = x - 11 ;;  
val y : int = 42  
# let x = 2 ;;  
val x : int = 2  
# y ;;  
- : int = ? -9 ou 42
```

On peut même écrire `let x = x + 1`
mais signification différente de l'affectation en C !

Let local

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

let $a = \text{expr1}$

expr2

\Rightarrow a reste disponible

Définition locale

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

let $a = \text{expr1}$ **in** expr2

\Rightarrow expr1 écrite et **évaluée une seule fois**

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

\Rightarrow Nb : **let** $a = \text{expr1}$ **in** expr2 est une expression

Plan

Présentation du cours

Motivations

Expressions et types de base

Définition d'un identificateur

Types composés

Sommes et filtrage

Types somme rékursifs

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 récurifs sont naturellement adaptés à la manipulation de **structures récurives**.

Deux grands procédés de composition

- ▶ produits de types
existent nativement dans tous les langages de programmation
 - ▶ **sommes** de types
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) ;;

Besoins contradictoires

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

Besoins contradictoires

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 solution : union (casse le typage)

Solution compliquée : types avec discriminants

*Solution OCaml : union **disjointe** = **somme***

Types somme

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 type d'origine (ex. int, float, float × float).

Syntaxe : les constructeurs débutent par une majuscule.

Sommes : exemples dégénérés

Énumération

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

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

Syntax error

```
# type extremite = 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

Expressions et types de base

Définition d'un identificateur

Types composés

Sommes et filtrage

Types somme rékursifs

Exemples de type somme

```
type legume = Haricot | Carotte | Courge
type fleur = Rose | Hortensia
type fruit = Poire | Banane
type couleur = Rouge | Jaune | Blanc
type plante =
  | Legume of legume
  | Fleur of fleur * couleur
  | Fruitier of fruit
type objet =
  | Plante 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 =
  | Legume of legume
  | Fleur of fleur * couleur
  | Fruitier of fruit
type objet =
  | Plante of plante
  | ...
```

Représentation graphique (au tableau)

Arbres

Type somme et filtrage

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  
| Plante (Fleur (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

Expressions et types de base

Définition d'un identificateur

Types composés

Sommes et filtrage

Types somme rékursifs

Type somme rékursif : listes d'entiers

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

```
match l with  
  | Nil → true  
  | Cons (x, l) → false
```

```
match l with  
  | [] → 0  
  | x :: l → 1 + longueur l
```