



Programmation fonctionnelle 1

Caml



L3 Informatique
Semestre 5

Cours donné par Christine MAUREL
Rédigé par Antoine de ROQUEMAUREL

2013

Table des matières

1	La programmation fonctionnelle	3
1.1	Différents paradigmes de programmation	3
1.2	Le fonctionnel	3
2	Syntaxe	4
2.1	Action	4
2.2	Types de base	4
2.3	Structures de contrôles	4
2.4	Variables	4
2.5	Fonctions	5
2.6	Nuplets	7
2.7	Filtrage (pattern matching)	7
A	Liste des codes sources	9

La programmation fonctionnelle

1.1 Différents paradigmes de programmation

- Impératif : C, Java, Ada, ...
- Objet : Java, C++, ...
- Fonctionnel : Lisp, Scheme, ML, Caml, Haskell, ...
- Déclarative ou logique : Prolog

1.2 Le fonctionnel

L'outil de base de la programmation fonctionnel est les fonctions. On peut les définir, les appliquer et les composer. Il n'y a pas d'affectation en fonctionnel.

Le fonctionnel est parti d'une base théorique avec le λ calcul en 1936, c'est un langage sûr. C'était d'abord non typé¹, les langages typés sont arrivés ensuite avec la famille Ocaml vers les années 2000.

Un langage fonctionnel typé possède plusieurs propriétés.

Inférence de type On ne déclare pas le type expressément.

Vérification de type Vérifier à la compilation, pas de risque de problème lors de l'exécution

Polymorphe

Syntaxe simple Syntaxe non verbeuse, sémantique solide, environnement de développement solide, mise au point facilitée et programmation sûre

1.2.1 Mode de compilation

Le Caml peut être soit compilé soit interprété, l'avantage de la compilation étant l'efficacité et l'interprétation « convivial ». Historiquement ceux-ci étaient uniquement compilés.

1. Comme le lisp ou, Scheme

Syntaxe

2.1 Action

```

1 | # expression ;;
   | -: valeur : type
3 | #

```

Listing 2.1 – Syntaxe de base

- Lire l'expression jusqu'au ; ;
- Typer
 - Si ko \Rightarrow Message d'erreur
 - Si ok \Rightarrow Évaluation \Rightarrow « Réduire, calculer » \Rightarrow Résultat / Valeur

2.2 Types de base

Type	Mot clé	Opération	Comparaison	Exemple
Entiers(\mathbb{Z})	int	+, -, *, /, mod	=, >, <, >=, <=, <>	2013
Flottants	float	+, -, *, /, sqrt, **	Polymorphe	2013.0
Chaines	string	"_", ^	Polymorphe	"coucou"
Caractères	char	'_'	Polymorphe	'c'
Booléens	bool	true, false, &&, , not	Polymorphe	

2.3 Structures de contrôles

```

1 | # if condition then action else alternative ;;

```

Listing 2.2 – Syntaxe de la condition



- La condition doit être un booléen.
- L'action et l'alternative doit être du même type

2.4 Variables

Un définition peut être de plusieurs type :

- Globale
- Locale
- Simultanée

2.4.1 Définition globale

```
1 | # let variable = expression;;
```

Listing 2.3 – Définition de variable

L'interpréteur va évaluer la valeur et donner un type à la variable, il effectue une liaison $\langle \text{var}, \text{val} \rangle$, ceci peut aussi s'appeler une fermeture.

On ajoute la liaison à l'environnement, un environnement est donc un ensemble ordonné de liaisons.

2.4.2 Définition Locale

```
1 | # let variable = expression 1
   | in expression2 ;;
```

Listing 2.4 – Définition de variable

La définition est temporaire

1. Évaluer l'expression dans l'environnement courant
2. Ajouter à l'environnement courant la nouvelle. Liaison $\text{var}, \text{val1}$
3. Évaluer l'expression 2 dans ce nouvel environnement augmenté \Rightarrow Résultat
4. Restituer environnement de départ

2.4.3 Définitions simultanées

```
2 | # let var1 = expression1
   | and var2 = expression2
   | and var3 = expression3;;
```

Listing 2.5 – Définition de variable

2.5 Fonctions

2.5.1 Définition

```
1 | # fun param -> corps;;
   | # function param -> corps;;
```

Listing 2.6 – Syntaxe d'une définition de fonction



Il existe une différence entre `fun` et `function`, cette différence sera vu plus tard

```
# fun x -> x + 1;;
int -> int = <fun>
# let succ = func x -> x + 1;;
succ: int -> int = <fun>
```

Listing 2.7 – Syntaxe d’une définition de fonction

2.5.2 Application

2.5.2.1 Valeur d’une fonction dans un environnement Γ

Évaluer une fonction `fun x -> corps` dans Γ nous donne la fermeture suivante $\langle \Gamma, x, corps \rangle$

2.5.2.2 Application d’une fonction à un argument dans Γ_2

- Évaluer `f` dans Γ_1
- Évaluer `a` dans Γ_1 Soit `v` la valeur de `a` dans Γ_1
- Soit $\langle \Gamma, x, corps \rangle$ la valeur de `f` dans Γ_1
- On « branche `x` et `v` » et on évalue le corps de la fonction dans l’environnement où `x` est lié à `v` a été ajouté à Γ
- Résultat

```
# let x = 2013;;
val x : int = 2013
# let y = x + 10
  and z = x * 10;;
val y : int = 2013 z = int : 20130
# let f = func x -> x + z + y;;
val f: int -> int = <fun>
# f(y+1);;
```

Listing 2.8 – Exemple d’utilisation de fonctions

2.5.3 Fonction à n paramètres

```
# fun x1 -> fun x2 -> fun x3 -> ... -> fun xn -> corps;;
# fun x1 x2 x3 ... xn -> corps;;
```

Listing 2.9 – Syntaxe d’une définition de fonction à n paramètres

Une fonction à N paramètre fonctionne à l'aide de N fonctions à 1 paramètre.

2.5.4 Application partielle

```
# let creerPredPGQ = fun x -> fun y -> x > y;;
# let plusGrandQue10 = fun creePredPGQ 10;
# plusGrandQue10 5;;
- : bool = false
```

Listing 2.10 – Application partielle

2.6 Nuplets

Permet de rassembler eds informations dont on connait à l'avance le nombre et le type éventuellement hétérogène.

```
(x1, x2, ..., xn)
tx1 * tx2 * ... * txn
```

Listing 2.11 – Syntaxe de n-uplets

```
# (1, 2, 3);;
-: int * int * int = 1, 2, 3
# (1, (2,3));;
int * (int * int)
# let first = fun (x,y) -> x ;;
val first = ('a*'b) -> 'a = <fun>
# let consCple = fun x -> fun y -> (x, y);;
val consCple = 'a -> 'b -> ('a*'b) = <fun>
```

Listing 2.12 – Exemple n-uplets

2.7 Filtrage (pattern matching)


Filtre ou motif, permet d'exprimer la syntaxe d'une donnée. On écrit la fonction par cas, c'est-à-dire on filtre la donnée avec un filtre¹.

```
match expr with
pat1 -> expr1
| pat2 -> expr2
| pat31|pat32|pat33 -> expr3 (* un des pattern retourne expr3 *)
| patn -> exprn
| _ -> not b;; (* default *)
```

Listing 2.13 – Syntaxe du filtrage

On examine en séquence et essaye de filtrer successivement l'expression avec le pattern i, le premier à marcher sera appliqué.

1. ou pattern

 Les pattern doivent tous être de même type afin que cela fonctionne.

```
# let nand = fun a -> fun b ->
match a with false -> true
      | _ -> not b;;
```

Listing 2.14 – Exemple filtrage

Écrire la fonction d'implication.

A	B	$A \rightarrow B$
T	T	T
T	F	F
F	T	T
F	F	T

```
# let impl = fun a -> fun b ->
match (a,b) with
  (true,true) -> true
|(true,false) -> false
|(false,true) -> true
|(false,false) -> true;;
(* Autre manière plus élégante *)
# let imp = fun a -> fun b ->
match (a,b) with
  (true, false) -> false
  | _ -> true;;
```

Listing 2.15 – Exemple filtrage – Implication

Liste des codes sources

2.1	Syntaxe de base	4
2.2	Syntaxe de la condition	4
2.3	Définition de variable	5
2.4	Définition de variable	5
2.5	Définition de variable	5
2.6	Syntaxe d'une définition de fonction	5
2.7	Syntaxe d'une définition de fonction	6
2.8	Exemple d'utilisation de fonctions	6
2.9	Syntaxe d'une définition de fonction à n paramètres	6
2.10	Application partielle	7
2.11	Syntaxe de n-uplets	7
2.12	Exemple n-uplets	7
2.13	Syntaxe du filtrage	7
2.14	Exemple filtrage	8
2.15	Exemple filtrage – Implication	8