UNIVERSITÉ TOULOUSE III PAUL SABATIER

Rédigé avec LaTeX
Version du 5 novembre 2013

Programmation fonctionnelle 1

Caml

L3 Informatique
Semestre 5

Cours donné par Christine Maurel
Rédigé par Antoine de Roquemaurel

2013

Table des matières

Chapitre 1

La programmation fonctionnelle

1.1 Différents paradigmes de programmation

- Impératif : C, Java, Ada, ...
- Objet : Java, C++, ...
- Fonctionnel : Lisp, Shceme, 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 partis d'une base théorique avec le λ calcul en 1936,c'est un langage sûr. C'était d'abord non typé [1], 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'execution

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

Chapitre 2

Syntaxe de base

2.1 Action

```
# expression ;;
-: valeur : type
#
```

Listing 2.1 – Syntaxe de base

- Lire l'expression jusqu'au ;;
- Typer
  - Si ko ⇒ Message d'erreur
  - Si ok ⇒ Évaluation ⇒ « Réduire, calculer » ⇒ 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

2.3.1 Conditions

```
# 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

2.3.2 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 filtre [1].

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

Listing 2.3 – Syntaxe du filtrage

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

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

Les exemples ci-dessous utilisent des fonctions, celles-ci sont détaillées dans le chapitre 3.

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

Listing 2.4 – Exemple filtrage

Écrire la fonction d'implication.

A	B	A→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 2.5 – Exemple filtrage – Implication

2.4 Variables

Un définition peut être de plusieurs type :

1. ou pattern

– Globale
– Locale
– Simultanée

2.4.1 Définition globale

```
# let variable = expression;;
```

Listing 2.6 – Définition de variable

L'interpréteur va évaluer la valeur et donner un type à la variable, il effectue une liaison `<var,val>`, 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

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

Listing 2.7 – Définition de variable

La définition et temporaire

1. Évaluer l'expression dans l'environnement ourant
2. Ajouter à l'environnement courantla nouvelle. Liaison `var,val1`
3. Évaluer l'expression 2 dans ce nouvel environnement augmenté ⇒ Résultat
4. Restituer environnement de départ

2.4.3 Définitions simultanées

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

Listing 2.8 – Définition de variable

2.5 Exceptions

Dans le cas où on ne veut pas rattraper une exception, celles-ci peuvent s'effectuer simplement à l'aide de `failwith` suivi du message d'erreur.

Chapitre 3

Fonctions

3.1 Définition

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

Listing 3.1 – Sytaxe d'une définition de fonction

R Il existe une différence entre `fun` et `function`. `function` permet d'alléger l'écriture en cas de pattern matching. En effet

```
fun x -> match x with (* contenu du match ... *);;
```

est équivalent à l'écriture suivante :

```
function (* contenu du match ... *);;
```

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

Listing 3.2 – Sytaxe d'une définition de fonction

R Il est possible de tracer une fonction, ceci- s'effectue à l'aide de la commande suiante, cette trace est succinte mais permet de savoir ce qui se passe lors de l'exécution de la fonction :
`#trace fonction`
Afin d'enlever la trace, il suffit d'utiliser `#untrace fonction`
Cela peut s'avérer particulièrement utile pour les fonctions récursives, cf 3.5.

3.2 Application

3.2.1 Valeur d'une fonction dans un environnement Γ

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

3.2.2 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 $< \Gamma, x, corps >$ 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 3.3 – Exemple d'utilisation de fonctions

3.3 Fonction à n paramètres

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

Listing 3.4 – Sytaxe d'une définition de fonction à n paramètres

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

3.4 Application partielle

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

Listing 3.5 – Application partielle

3.5 Fonctions récursives sur les entiers

Une récursive implique qu'il y ai une référence au nom de la fonction dans le corps de cette même fonction. Systématiquement un cas d'arrêt de la fonction doit être présent, ceci afin que la fonction se termine, éventuellement des cas d'erreurs peuvent être gérés.

Fonction qui étant donné un entier n, calcule sa factorielle c'est-à-dire $n!$ en sachant que :

$$\begin{aligned} n &\leq 0 \\ \\ 0 &\neq 1 \\ n &\neq n \times (n-1)! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n \end{aligned}$$

```
let rec fact = fun n ->
  if n = 0 then 1 (* arrêt *)
  else n * fact(n-1);; (* cas général*)
```

Listing 3.6 – Exemple de la fonction factorielle en récursif

La solution en utilisant le filtrage :

```
let rec fact = fun n ->
  match n with
     0 -1 1
    |p -> p*fact(p-1);;
```

Listing 3.7 – Exemple de la fonction factorielle avec filtrage

Les vérifications du domaine d'application doivent se faire en dehors de la fonction récursives. En effet, il est inutile de tester le cas d'erreur à chaque appel récursif. Ceci peut s'effectuer de la manière suivante :

```
let fact = fun n ->
  let rec calcul = fun x ->
    if x = 0 then 1 (* cas d'arrêt *)
    else x * calcul(x-1) (* cas général *)
  in if n < 0 then failwith "erreur nombre négatif"
    else calcul n;;
```

Listing 3.8 – Exemple de la fonction factorielle avec un cas d'erreur

R Il faut faire attention, dans ce cas il est donc impossible de tracer le cas d'erreur, en effet la fonction n'est pas visible de l'extérieur.

3.5.1 Fonction mutuellement récursives

Il est également possible d'avoir deux fonctions que l'on appelle mutuellement récursives. C'est-à-dire que les fonctions s'appellent les une les autres, la première fonction appelle la seconde et la seconde appelle la première.

Écriture de la fonction pair avec deux fonctions mutuellement récursives.

```
fonction pair
  si n = 0 alors true
  sinon impair(n-1)
```

```
-- Avec la fonction impair comme ceci :
fonction impair
  si n = 0 alors false
  sinon pair(n-1)
```

Listing 3.9 – Algorithme de la récursivité mutuelle pour pair

Ce qui se traduirait de la façon suivante en caml, les deux fonctions doivent être déclarés en même temps, pour ceci on utilise le `and`.

```
# let rec pair = fun n ->
  if n = 0 then true
  else impair(n-1)
and impair = fun n ->
  if n = 0 then false
  else pair (n-1);;
```

Listing 3.10 – Récursivité mutuelle pour pair

Cas d'arrêt	Diminution taille du problème
$n = 0$	$n - 1$
$n = 1$	$n - 2$
$a = b$	$a - b$
$n < 10$	$\frac{n}{10}$

Chapitre 4

Structures de données

4.1 Nuplets

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

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

Listing 4.1 – 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 4.2 – Exemple n-uplets

Ils permettent de mettre des informations hétérogènes, cependant la dimension doit être connue à l'avance. L'avantage étant l'accès aux informations par filtrage.

Il faut que N soit raisonnable, pour un grand nombre d'informations ce n'est pas adapté

4.2 Listes

Elles permettent de rassembler N informations avec un N quelconque, cependant les informations doivent être homogènes.

On définie une liste par induction : la liste est vide, ou l'ajout d'un élément.

En Caml le type est list, noté `'a list`, correspondant à une liste dont tous les éléments sont de type a. Le constructeur d'une liste vide se fait avec `[]`. L'ajout d'un élément se fait systématiquement en tête, à l'aide de l'opérateur `::`, `e::l` ajoute l'élément e en tête de la liste.

```
e1::e2::...::en::[]
[e1;e2;...;e3]
```

```
#[];;
- : 'a list = []
#let ex = 1::[];;
val ex : int list = [1]
#let ex1 = 1 ::2::3::[];;
val ex1 : int list = [1;2;3]
#let ex2 = 0::ex1
val ex2 : int list = [0;1;2;3]
#let ex3 = [1,2]::({3,4]::[]);;
[[(1,2)];[(3,4)]]
let ex4 = (1,tue)::[(2,false)}
ex4 : (int * bool) lt = [(1,true);(2,false)]
```

Listing 4.3 – Exemple de listes

On peut utiliser le filtrage sur les listes avec les filtrage `[]` et `::l` permettant respectivement de savoir si une liste est vide ou d'accéder aux informations : tête(e), reste(l)

4.2.1 Fonctions récursives sur les listes

Le cas d'arrêt est 1 ou plusieurs élément, le cas général correspond souvent à diminuer la taille de la liste.

```
let rec dernier = function
     e::[] -> e
    |e::l -> dernier l
    |_ -> failwith "erreur liste vide";;

let rec longueur = function
     [] -> 0
    |e::l -> (longueur l)+1 ;;

(* somme des élémets d'un liste
 * int list -> int
 *)
let rec somme = function
     [] -> 0
    |t::q -> t + somme q ;;
```

Listing 4.4 – Fonctions sur les listes

4.2.1.1 La fonction append

La fonction **append** est une fonction qui concatène deux listes, elle pourrait être implémenter comme ceci :

```
(* Concatène deux liste, version naive *)
let rec append = fun l1 l2
    match (l1,l2) with
     ([],[]) -> []
    |([]::e, t::q) -> t::q
    |([t1:::q1, []) -> t1::q1
    |(t1::q1, t2::q2) -> t1::(append q1 (t2::q2));;

```

```
(* Bonne version de append *)
let rec append = fun l1 l2 ->
    match l1 with
     [] -> l2
    |t::q -> t::(append q l2);;
```

Listing 4.5 – Exemple d'implémentation de `append`

> R Cette fonction est déjà disponible dans le langage OCaml, pour cela on peut utiliser l'opérateur `@` : `l1@l2`

4.2.2 La fonction `reverse`

La fonction `reverse` retourne la liste construite à l'envers.

```
(* Renvoie la liste constuite à l'envers *)
let rec reverse = function
     [] -> []
    |t::q ->  (reverse q)@t::[] ;;
```

Listing 4.6 – Fonction `reverse`

4.2.3 La fonction `nbOcc`

Fonction qui compte le nombre d'occurrence d'un élément dans une liste.

```
(* compte le nombe d'occurences de l'élément dans la liste *)
let rec nbOcc e l ->
    match l with
     [] -> 0
    |t::q -> if t = e then (nbOcc e q)+1 else nbOccc e q;;

(* Le e ne bouge pas durant l'appel récursif. Il peut être intéressant donc de
 * ne pas le passe systématiquement *)
let nbOccBis = fun e ->
    let rec compte = fun l ->
        match with
             [] -> 0
            |t::q -> if t = e then 1 + compte q else compte q
    in compte;;
```

Listing 4.7 – Fonction `nbOcc`

Annexe

A

Liste des codes sources

Annexe B

Exercices

B.1 Fonctions

B.1.1 Récursivité sur les entiers

B.1.1.1 Parité

Donner la définition récursive de la fonction pair qui retourne un booléen si n est pair :

```
si n = 0 alors true
si n = 1 alors false
si n > 1 alors pair(n-2)
si n < 0 alors erreur
```

```
# let pair = fun n ->
  let rec verifMult2 = fun x ->
    match x with
       0 -> true
      |1 -> false
      |p -> verifMult2(p-2)
  in if n < 0 then verifMult2(-n)
    else verifMult2 n;;
int -> bool = <fun>
```

Listing B.1 – Exercice – Fonction pair en récursif

B.1.1.2 `sommeCarres`

Écriture d'une fonction qui effectue la somme des carrés : $1^2 + 2^2 + 3^2 + \cdots + n^2$

```
# let sommeCarres = fun n ->
  let rec funRecCarres = fun x ->
    if x = 0 then 0
    else (x*x) + funRecCarres(x-1)
  in if n < 0 then funRecCarres (-n)
    else funRecCarres n;;
```

Listing B.2 – Exercice – Fonction pair en récursif

B.1.1.3 `sommeFonctions`

```
# let sommeFonctions = fun f -> fun n ->
  let rec calcul = fun x ->
    if x = 0 then 0
    else (f x) + calcul(x-1)
  in if n < 0 then calcul(-n)
    else calcul n;;
```

Listing B.3 – Exercice – Fonction pair en récursif

B.1.2 PGCD

Écrire la fonction pgcd tel que `pgcd a b` est égale au lus grand diviseur de a et de b.

Par soustraction successive, le pgcd de a et b est le pgcd du plus petit des 2 et de la valeur absolue de leur différence $a > 0$ et $b > 0$.

```
  si a = b alors a
  si a < b alrs pgcd a (b-a)
  si a > b alors pgcd (a-b) b
```

Listing B.4 – Exercices – Algorithme pgcd

```
let pgcd = fun a -> fun b ->
  let rec trait = fun x -> fun y -> (* x > 0 et y > 0*)
    if x = y then x (* cas d'arrêt *)
    else if x < y then trait x (y-x) (* Appel récursif *)
    else trait (x-y) y (* Appel récursif *)
  in if(a > 0) && (b > 0) then trait a b
     else failwith "PGCD, entiers négatifs ou nuls";;
```

Listing B.5 – Exercice – Fonction pgcd

B.1.3 Exercices diverses sur les fonctions

B.1.3.1 dernierChiffre

B.1.3.2 Son argument privé de son dernier chiffre

B.1.3.3 nombre d'occurence d'un chiffre

Compte le nombre d'occurrence d'un chiffre dans l'écriture décimale d'un entier

B.2 Structures de données

B.2.1 Liste

```
- : int list = [1; 2; 3]
# 1::[2;3];r
- : int list = [1; 2; 3]
```

```
# 1::(1*2)::[2+1];;
- : int list = [1; 2; 3]
# 1::2::3::[];;
- : int list = [1; 2; 3]
# (2=3-1)::(1<2)::false::[];;
- : bool list = [true; true; false]
# 1.5::(2.5::(3::[]));;
Error: This expression has type int but an expression was expected of type float
# [1,2,3];;
- : (int * int * int) list = [(1, 2, 3)]
# [[1];[2];[3;4];[]];;
- : int list list = [[1]; [2]; [3; 4]; []]
# [[1];[2.5];[3;4];[]];;
 Error: This expression has type float but an expression was expected of type int
# [1;true];;
 Error: This expression has type bool but an expression was expected of type int
# [1,true];;
- : (int * bool) list = [(1, true)]
# [[1;2];[];3;4];;
 Error: This expression has type int but an expression was expected of type ↩
     int list
# [1]::[];;
 - : int list list = [[1]]
# []::[];;
 - : 'a list list = [[]]
# [1]::[[2;3];[4]];;
 - : int list list = [[1]; [2; 3]; [4]]
```

Listing B.6 – Exercice – listes

```
let elem2Sur3= function
    un::deux::trois::[]->deux;;

let elem2sur3Bis = fun l ->
    match l with
        _::deux::_::[]->deux;;

let elem2sur3Ter = function
    _::deux::_::[]->deux
    |_ -> failwith "erreur";;

let elem2 = function
    _::deux::_::_->deux;;

let access = function
    (_::deux::_,_::_) -> deux;;

let accessBis = function
    (_,deux)::_::[] -> deux;;
```

Listing B.7 – Exercice – fonctions sur les listes