Université de Nice Sophia-Antipolis
Master STIC 1ère année *Informatique*

Année 2006-2007

Paradigmes et Langages de Programmation

Introduction au langage CAML 

Laurence PIERRE
(Laurence.Pierre@unice.fr)

La *programmation fonctionnelle* a été étudiée en Licence d'Informatique par l'intermédiaire du langage Scheme, nous ne reviendrons donc que brièvement sur les principes de base. Ce cours (de 3h) se concentre sur les spécificités du langage CAML, comme l'inférence de types, les définitions par filtrage,…

PLAN DU COURS

I. Pour débuter... 3

I.1 Premières manipulations

I.2 Quelques mots sur les types de base

I.3 Types composites

II. Fonctions 9

II.1 Définition de fonctions

II.1.1 Fonctions à un paramètre
II.1.2 Fonctions à plusieurs paramètres
II.1.3 Portée des variables

II.2 Fonctions d'ordre supérieur

II.3 Fonctions polymorphes

II.4 Fonctions récursives

III. Filtrage de motif 19

III.1 Déclaration de types utilisateur

III.1.1 Enregistrements (types produit)
III.1.2 Types somme

III.2 Filtrage

III.2.1 Principes et syntaxe
III.2.2 Filtrage et fonctions récursives

Quelques compléments 30

Structures modifiables
Entrées/sorties
Chargement, compilation
Programmation modulaire, compilation séparée

I. POUR DEBUTER...

Le langage ML ("Meta-Language") fut développé dans les années 1970, dans l'équipe de Robin Milner à l'Université d'Edinburgh. A l'origine, l'objectif était la mise en oeuvre d'un système de démonstration automatique, LCF ("Logic of Computable Functions") permettant de faire du raisonnement formel sur des programmes. Le langage CAML ("Categorical Abstract Machine Language"), comme Standard ML (SML), est l'un des dialectes de ML.

CAML est développé à l'INRIA (`http://caml.inria.fr`) depuis le milieu des années 1980. Caml Light est une version allégée du langage CAML d'origine, et son successeur Objective Caml, créé en 1996, propose notamment des aspects de programmation orientée-objet. Cette brève introduction ne présentera que certains aspects essentiels du langage (non objets), en s'appuyant sur OCaml.

ML (et donc CAML) est principalement un *langage de programmation fonctionnelle*, qui a subi l'influence de Lisp et s'appuie tout comme lui sur une mise en oeuvre du lambda-calcul. On y trouve donc les caractéristiques de cette famille de langages : fonctions de première classe (peuvent être passées en argument ou retournées comme valeur d'une fonction, être la valeur d'une variable,...), utilisation de la récursion, non-utilisation (autant que possible !) d'effets de bords,... C'est également un *langage fortement typé*, proposant un puissant mécanisme d'inférence de types et permettant de manipuler des fonctions polymorphes.

I.1 Premières manipulations

Avant d'apprendre à réellement programmer en Caml, et à compiler nos programmes, nous allons commencer à découvrir la syntaxe du langage avec quelques manipulations d'expressions sous l'interpréteur.

Une fois sous l'interpréteur OCaml, le prompt est un #, toute expression doit être terminée par `;;` (double point-virgule). La sortie de l'interpréteur se fait par `#quit`. On peut faire évaluer n'importe quelle expression respectant la syntaxe. La réponse

de l'interpréteur donne le nom de la variable si l'expression définit une variable (on a un – sinon), son type, et sa valeur. Si l'on procède à une *définition globale de variable* (`let`), la variable est utilisable dans les expressions écrites ultérieurement. Des déclarations globales simultanées peuvent être faites grâce au `and`, mais les symboles ne sont connus qu'à la fin de toutes les déclarations. Un regroupement de déclarations globales provoque une évaluation séquentielle.

Exemple :

```
mamachine% ocaml
        Objective Caml version 3.08.3

# 3 + 8 ;;
- : int = 11

# let x = 4 * 11 ;;
val x : int = 44

# x - 23 ;;
- : int = 21

# let y = 5 and w = 8 ;;
val y : int = 5
val w : int = 8

# let a = 12 and b = a + 1 ;;   (* décl. simultanées *)
Unbound value a

# let a = 12 let b = a + 1 ;;   (* regroupement *)
val a : int = 12
val b : int = 13
```

Attention à faire la différence entre les définitions *globales* et l'utilisation de *variables locales* à une expression (`let..in`). Attention aussi à faire la différence entre la *définition d'une variable* et l'évaluation d'une expression contenant *l'opérateur de comparaison* =.

Exemple :

```
# let x = 4 * 11 ;;
val x : int = 44

# let x = 13 in x + 5 ;;   (* remarquer le masquage *)
- : int = 18

# x ;;
- : int = 44

# x = 13 ;;
- : bool = false
```

```
# let x = 8 and y = 23 in y - x + 1 ;;
- : int = 16
```

Enfin, une *déclaration locale* est elle-même une expression, elle peut donc être utilisée pour construire d'autres expressions.

Exemple :

```
# (let a = 3 in a * a) + 10 ;;
- : int = 19
# let a = 3 in let b = a + 78 ;;
Syntax error
# let b = (let a = 3 in a + 78) ;;
val b : int = 81
# a ;;
Unbound value a
# b ;;
- : int = 81
# let a = 3 in (let c = a + 78 in c + 1) ;;
- : int = 82
# c ;;
Unbound value c
# let a = 3 in (let c = 78 in a + c) ;;
- : int = 81
# c ;;
Unbound value c
```

I.2 Quelques mots sur les types de base

Nous allons ici faire un rapide tour d'horizon des types de base et de leurs opérateurs. On pourra par exemple trouver un petit récapitulatif à ce sujet à `http://pauillac.inria.fr/ocaml/htmlman/manual003.html`.

Les nombres *entiers et flottants*, les *caractères* et *chaînes de caractères*, et les *booléens* sont prédéfinis.

On distingue les opérateurs pour nombres entiers des opérateurs pour nombres flottants : `+ - *` et `/` pour les entiers, et `+. -. *.` et `/.` pour les nombres flottants. Les fonctions de conversion `float_of_int` et `int_of_float`

peuvent permettre de passer d'un type à l'autre (nécessaire notamment pour les comparaisons !).

Exemple :

```
# 4 + 7 ;;
- : int = 11

# 4.2 +. 3 ;;
This expression has type int but is here used
with type float

# 4.2 +. 3. ;;
- : float = 7.2

# 13 / 3 ;;
- : int = 4

# 13. /. 3. ;;
- : float = 4.33333333333333304

# 6 = 6. ;;
This expression has type float but is here used
with type int

# float_of_int 6 = 6.0 ;;
- : bool = true

# int_of_float 7.8 ;;
- : int = 7

# floor 7.8 ;;
- : float = 7.
```

Les types `char` et `string` correspondent respectivement aux caractères et chaînes de caractères.
La notation `.[ ]` permet l'accès aux caractères d'une chaîne. Diverses fonctions de conversion sont disponibles : `int_of_char`, `char_of_int`, `int_of_string`, `string_of_int`, `float_of_string` et `string_of_float`. Enfin, l'opérateur `^` permet de concaténer des chaînes de caractères.

Exemple :

```
# int_of_char 'Z' ;;
- : int = 90

# "Bonjour" ;;
- : string = "Bonjour"

# let s = "Bonjour " ^ "Jim" ;;
```

```
val s : string = "Bonjour Jim"
# int_of_string "2005" ;;
- : int = 2005
# float_of_string "78.3" ;;
- : float = 78.3
# s.[2] ;;
- : char = 'n'
```

On dispose des opérateurs de comparaison = (égal), <> (différent), et < > <= >=, et des connecteurs logiques && (et), || (ou) et not (négation).

Exemple :

```
# 7.2 > 3.56 && "Bonjour" <> "bonjour" ;;
- : bool = true
# let a = 'X' ;;
val a : char = 'X'
# a = 'Y' || a >= 'W' ;;
- : bool = true
```

I.3 Types composites

Les types composites *n-uplets* et *listes* peuvent être construits à partir d'éléments des types de base présentés ci-dessus. Caml donne aussi la possibilité de manipuler des *tableaux* (vecteurs), dont nous ne parlerons pas ici.

Les *n-uplets* sont formés d'éléments, qui peuvent être de types différents, séparés par des virgules. Les fonctions `fst` et `snd` permettent d'accéder respectivement au 1er et 2ème élement d'un couple (2-uplet).

Exemple :

```
# let montuple = 4 , 8.8, 'V' ;;
val montuple : int * float * char = (4, 8.8, 'V')
# let montuple2 = 4 , 8.8, (9 , 6.7), 'V' ;;
val montuple2 : int * float * (int * float) * char =
   (4, 8.8, (9, 6.7), 'V')
# let couple = "elt1", 2 ;;
val couple : string * int = ("elt1", 2)
```

```
# fst couple ;;
- : string = "elt1"

# snd couple ;;
- : int = 2
```

Les *listes* sont des suites de valeurs *de même type*, de longueur non prédéfinie. Les éléments sont placés entre `[ ]` et sont séparés par des points-virgules. L'ajout d'un élément en tête de liste (le `cons` de Lisp) se fait par `::`, la tête et la queue de liste (`car` et `cdr` de Lisp) peuvent être extraites par `List.hd` et `List.tl`. La concaténation de deux listes est réalisée par l'opérateur `@`.

Remarque : `hd` et `tl` sont des fonctions de la bibliothèque `List`. Ces fonctions, comme toutes les fonctions de bibliothèques, peuvent être utilisées grâce à une notation pointée comme indiqué ci-dessus, mais il est également possible d'ouvrir la bibliothèque (`open`) dans l'environnement courant.

Exemple :

```
# let maliste = [ 48 ; 92 ; 3 ; 17 ] ;;
val maliste : int list = [48; 92; 3; 17]

# [] ;;      (* liste vide *)
- : 'a list = []

# 25 :: maliste ;;
- : int list = [25; 48; 92; 3; 17]

# List.hd maliste ;;
- : int = 48

# List.tl maliste ;;
- : int list = [92; 3; 17]

# maliste @ [ 8 ; 5 ] ;;
- : int list = [48; 92; 3; 17; 8; 5]

# open List ;;

# hd maliste ;;
- : int = 48

# tl maliste ;;
- : int list = [92; 3; 17]
```

II. FONCTIONS

Nous allons tout d'abord étudier la syntaxe des définitions de fonctions en Caml et leur utilisation, puis nous verrons comment définir et utiliser des fonctions d'ordre supérieur, et des fonctions récursives.

II.1 Définition de fonctions

II.1.1 Fonctions à un paramètre

La syntaxe d'une *expression fonctionnelle à un paramètre* est la suivante :

```
function paramètre -> expression
```

Comme précédemment, le système *infère le type* de cette expression, par exemple `int -> float` est le type de fonction prenant un entier en paramètre et renvoyant un réel.

L'*application* d'une fonction à un argument se fait en faisant suivre la fonction de l'argument, avec ou sans parenthèses.

Exemple :

```
# function x -> 3 * x ;;
- : int -> int = <fun>
# (function x -> 3 * x) 7 ;;
- : int = 21
# (function x -> 3 * x)(18) ;;
- : int = 54
```

Bien entendu, manipuler ainsi une fonction anonyme n'est pas très pratique. On peut *nommer les fonctions*, grâce à `let`, comme nous l'avons vu pour la définition de variables dans la section I.1.

Exemple :

```
# let triple = function x -> 3 * x ;;
val triple : int -> int = <fun>
```

```
# triple 25 ;;
- : int = 75
# triple(9) ;;
- : int = 27
```

Une syntaxe alternative `let nom paramètre = expression` permet de simplifier l'écriture des fonctions.

Exemple :

```
# let triple x = 3 * x ;;
val triple : int -> int = <fun>
# triple 5 ;;
- : int = 15
```

II.1.2 Fonctions à plusieurs paramètres

La première possibilité pour définir une *expression fonctionnelle à plusieurs paramètres* consiste à utiliser la même syntaxe que dans le cas des expressions fonctionnelles à un paramètre :

`function` p_1 `->` `function` p_2 `->` `...` `expression`

Comme précédemment, l'*application* de la fonction se fait en faisant suivre la fonction des arguments.

Exemple :

```
# function x -> function y -> x * y ;;
- : int -> int -> int = <fun>
# (function x -> function y -> x * y) 7 13 ;;
- : int = 91
# function x -> function y -> function z ->
                                        x * y + z ;;
- : int -> int -> int -> int = <fun>
# (function x -> function y -> function z -> x * y + z)
                                              4 6 10 ;;
- : int = 34
```

Une syntaxe plus compacte `fun` p_1 p_2 `...` `->` `expression` est également possible.

Exemple :

```
# fun x y -> x * y ;;
- : int -> int -> int = <fun>

# (fun x y -> x * y) 5 9 ;;
- : int = 45

# fun x y z -> x * y + z ;;
- : int -> int -> int -> int = <fun>

# (fun x y z -> x * y + z) 2 8 1 ;;
- : int = 17
```

Tout comme les fonctions à un paramètre, on peut *nommer les fonctions* à plusieurs paramètres grâce à `let`. Les deux syntaxes vues ci-dessus sont encore valables.

Exemple :

```
# let produit = fun x y -> x * y ;;
val produit : int -> int -> int = <fun>

# produit 9 5 ;;
- : int = 45

# let poly x y z = x * y + z ;;
val poly : int -> int -> int -> int = <fun>

# poly 3 9 4 ;;
- : int = 31
```

Il est à noter que les fonctions ainsi définies se présentent sous *forme curryfiée*, l'application peut se faire argument par argument (application partielle). Notamment, dans l'exemple ci-dessous, `produit 9` est une fonction de type `int -> int`, qui est appliquée ultérieurement à `5`.

Exemple :

```
# let mult9 = produit 9 ;;
val mult9 : int -> int = <fun>

# mult9 5 ;;
- : int = 45

# let fun1 = poly 10 4 ;;
val fun1 : int -> int = <fun>

# fun1 7 ;;
```

```
- : int = 47
# let fun2 = fun x y -> poly x y 1 ;;
val fun2 : int -> int -> int = <fun>
# fun2 6 3 ;;
- : int = 19
```

Attention, il est aussi possible de donner les *paramètres entre parenthèses, séparés par des virgules*, mais le type de la fonction n'est alors plus le même. On définit ainsi une version non curryfiée, il n'y a *en fait qu'un seul paramètre*, qui est un n-uplet. Il n'est bien sûr plus possible de procéder à une application argument par argument.

Exemple :

```
# let poly(x, y, z) = x * y + z ;;
val poly : int * int * int -> int = <fun>
# let fun1 = poly 10 4 ;;
This function is applied to too many arguments, maybe
you forgot a `;'
# poly 3 9 4 ;;
This function is applied to too many arguments, maybe
you forgot a `;'
# poly(3, 9, 4) ;;
- : int = 31
```

II.1.3 Portée des variables

Les expressions fonctionnelles peuvent faire intervenir des *variables locales* (construction `let ..in`).

Exemple :

```
# let polynome x =
    let a = 8 and b = 3 and c = 1
    in a * x * x + b * x + c ;;
val polynome : int -> int = <fun>
# polynome 2 ;;
- : int = 39
# a ;;
Unbound value a
```

Pour que l'évaluation d'une expression soit possible, il faut que toutes les variables qui y apparaissent soient définies. Les variables apparaissant dans une expression fonctionnelle peuvent être, hormis les paramètres et les variables locales, des variables de l'environnement.

En ce qui concerne ces variables, Caml utilise une *liaison statique* : l'environnement utilisé pour exécuter l'application d'une fonction est *celui de sa déclaration*, et pas l'environnement courant au moment de l'application (i.e. pas de portée dynamique).

Exemple :

```
# let produit x = a * x ;;
Unbound value a

# let a = 12 ;;
val a : int = 12

# let produit x = a * x ;;
val produit : int -> int = <fun>

# produit 4 ;;
- : int = 48

# let a = 3 ;;
val a : int = 3

# a ;;
- : int = 3

# produit 5 ;;
- : int = 60
```

II.2 Fonctions d'ordre supérieur

Les paramètres et résultats des fonctions Caml peuvent eux-mêmes être des fonctions. Les fonctions prenant en paramètre ou renvoyant en résultat des fonctions sont appelées des *fonctions d'ordre supérieur*.

Par exemple, la fonction `List.map` est une fonction d'ordre supérieur très utilisée, elle permet d'appliquer une fonction à tous les éléments d'une liste.

Exemple :

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

```
# List.map succ [ 48 ; 92 ; 3 ; 17 ] ;;
- : int list = [49; 93; 4; 18]
# let applique_et_add f x = f x + x ;;
val applique_et_add : (int -> int) -> int -> int = <fun>
# applique_et_add succ 15 ;;
- : int = 31
```

Dans l'exemple de la fonction `applique_et_add`, remarquons l'inférence de type provoquée par la présence de l'opérateur `+`.

II.3 Fonctions polymorphes

Jusqu'à présent, nous n'avons rencontré que des fonctions ne s'appliquant qu'à un seul type d'objets (monomorphes). Caml donne la possibilité de définir des fonctions *polymorphes* : certains des paramètres et/ou la valeur de retour sont d'un type quelconque. Des *variables de type* `'a`, `'b`, ... sont alors utilisées, elles représentent n'importe quel type, et seront instanciées par le type du paramètre effectif lors de l'application de la fonction.

Exemple : reprenons l'exemple ci-dessus, sans la présence du `+`

```
# let mon_applique f x = f x ;;
val mon_applique : ('a -> 'b) -> 'a -> 'b = <fun>
# mon_applique succ 147 ;;
- : int = 148
```

ici la fonction `succ`, de type `int -> int`, est utilisée, `'a` et `'b` sont donc instanciés par `int`, et le résultat est de type `'b` = `int`.

```
# mon_applique succ 56.2 ;;
This expression has type float but is here used with
type int
```

ici l'inférence de type échoue, `'a` ne peut pas simultanément être de type `int` et `float`.

Les exemples ci-dessous illustrent également le travail réalisé par le mécanisme d'inférence de types.

Exemple :

```
# let compose f g x = f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
                                                = <fun>

# compose succ List.hd [ 45; 7 ; 88 ; 13 ] ;;
- : int = 46

# List.hd ;;
- : 'a list -> 'a = <fun>

# let app2 f x y = f x y ;;
val app2 : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

# let somme l x = (List.hd l) + x ;;
val somme : int list -> int -> int = <fun>

# app2 somme [ 45; 7 ; 88 ; 13 ] 1 ;;
- : int = 46

# let couple x y = x , y ;;
val couple : 'a -> 'b -> 'a * 'b = <fun>

# couple 5.34 "flottant" ;;
- : float * string = (5.34, "flottant")

# app2 couple (List.hd [ 45; 7 ; 88 ; 13 ]) 90.4 ;;
- : int * float = (45, 90.4)
```

Il est toutefois possible de *contraindre une expression à avoir un type donné*, la syntaxe utilisée est `(expression : type)`. Ceci permet notamment de rendre visible le type des paramètres d'une fonction (la contrainte de type n'est pas indispensable, mais rend le code plus explicite), ou de limiter le contexte de l'inférence de type.

Exemple :

```
# let add (x: int) (y: int) = x + y ;;
val add : int -> int -> int = <fun>

# let couple (x: float) y = x , y ;;
val couple : float -> 'a -> float * 'a = <fun>

# couple 5.34 "flottant" ;;
- : float * string = (5.34, "flottant")

# couple (List.hd [ 45; 7 ; 88 ; 13 ]) 90.4 ;;
This expression has type int but is here used with
type float

# let monhd l = List.hd l ;;
val monhd : 'a list -> 'a = <fun>
```

```
# monhd [ 8.3 ; 4. ; 29.1 ] ;;
- : float = 8.3

# let monhd_int (l : int list) = List.hd l ;;
val monhd_int : int list -> int = <fun>

# monhd_int [ 8.3 ; 4. ; 29.1 ] ;;
This expression has type float but is here used with
type int

# monhd_int [ 87 ; 19 ; 22 ] ;;
- : int = 87
```

II.4 Fonctions récursives

La construction `let rec` permet des *définitions récursives* (attention, `let` ne le permet pas), la syntaxe est :

```
let rec nomfct p1 p2 ... = expression ;;
```

Une fonction récursive peut être écrite de différentes façons (du moment que son arrêt est garanti !). La section suivante traitera du filtrage, et son utilisation dans le contexte de définitions récursives sera étudiée. Les quelques exemples donnés ci-dessous ne font appel qu'à la méthode classique de définition de fonctions récursives dans un langage fonctionnel, ils font usage de la *structure de contrôle conditionnelle* `if expr1 then expr2 else expr3`.

La construction `and` permet de définir des *fonctions mutuellement récursives*.

La directive `#trace` permet de *suivre les appels récursifs* lors de l'application d'une fonction récursive (annulation par `#untrace`).

Exemples :

```
# let rec factorielle x =
    if x = 0 then 1 else x * factorielle(x-1) ;;
val factorielle : int -> int = <fun>

# factorielle 10 ;;
- : int = 3628800

# #trace factorielle ;;
factorielle is now traced.

# factorielle 5 ;;
factorielle <-- 5
```

```
factorielle <-- 4
factorielle <-- 3
factorielle <-- 2
factorielle <-- 1
factorielle <-- 0
factorielle --> 1
factorielle --> 1
factorielle --> 2
factorielle --> 6
factorielle --> 24
factorielle --> 120
- : int = 120

# #untrace factorielle ;;
factorielle is no longer traced.

# let fact n =
     let rec factorielle x =
       if x = 0 then 1 else x * factorielle(x-1) in
  if n >= 0 then factorielle n else 0 ;;
val fact : int -> int = <fun>

# fact 8 ;;
- : int = 40320

# fact (-4) ;;
- : int = 0

# let rec longueur l =
    if l = [] then 0 else 1 + longueur (List.tl l) ;;
val longueur : 'a list -> int = <fun>

# #trace longueur ;;
longueur is now traced.

# longueur [ 45; 7 ; 88 ; 13 ] ;;
longueur <-- [<poly>; <poly>; <poly>; <poly>]
longueur <-- [<poly>; <poly>; <poly>]
longueur <-- [<poly>; <poly>]
longueur <-- [<poly>]
longueur <-- []
longueur --> 0
longueur --> 1
longueur --> 2
longueur --> 3
longueur --> 4
- : int = 4

# let rec pair x =
     if x = 0 then true else impair (x - 1)
  and impair x =
     if x = 0 then false else pair (x - 1) ;;
val pair : int -> bool = <fun>
val impair : int -> bool = <fun>

# pair 853 ;;
```

```
- : bool = false

# impair 51 ;;
- : bool = true

# #trace pair ;;
pair is now traced.

# #trace impair ;;
impair is now traced.

# pair 5 ;;
pair <-- 5
impair <-- 4
pair <-- 3
impair <-- 2
pair <-- 1
impair <-- 0
impair --> false
pair --> false
impair --> false
pair --> false
impair --> false
pair --> false
- : bool = false
```

III. FILTRAGE DE MOTIF

Une caractéristique du langage Caml est de permettre le *filtrage de motif* (aussi appelé *pattern matching*). Nous allons étudier comment définir des fonctions en utilisant le filtrage sur les paramètres, en particulier dans le cas des fonctions récursives.
Au préalable, nous discutons rapidement la déclaration de types, notamment les types somme que nous recontrerons fréquemment dans les définitions avec filtrage.

III.1 Déclaration de types utilisateur

La déclaration d'un type se fait de la façon suivante :

```
type nom = definition_type ;;
```

Les déclarations de type sont par défaut récursives (pas de distinction nécessaire, comme dans le cas de `let` et `let rec`). Des types mutuellement récursifs peuvent être déclarés par :

```
type nom1 = definition_type1
     and nom2 = definition_type2
     and ... ;;
```

On distingue les types *produit* et types *somme*. Les types peuvent être *paramétrés* (paramètres de type `'a`, `'b`, ...).

III.1.1 Enregistrements (types produit)

La notion d'enregistrement est similaire à celle de "struct" en C. Un enregistrement contient des *champs*, chaque champ a un *nom* et un *type*. La déclaration d'un tel type se fait par :

```
type nom = { champ1 : type1; champ2 : type2 ...} ;;
```

Une *valeur de ce type* est créée en affectant une valeur à chaque champ, les affectations étant faites dans un ordre quelconque. Classiquement, une *notation pointée* permet l'accès aux champs.

Exemples :

```
# type point = { abscisse: float ; ordonnee: float } ;;
type point = { abscisse : float; ordonnee : float; }

# let point1 = { abscisse = 8.4 ; ordonnee = 23.9 } ;;
val point1 : point = {abscisse = 8.4; ordonnee = 23.9}

# let point2 = { ordonnee = 11. ; abscisse = 56.2 } ;;
val point2 : point = {abscisse = 56.2; ordonnee = 11.}

# point1 ;;
- : point = {abscisse = 8.4; ordonnee = 23.9}

# point2 ;;
- : point = {abscisse = 56.2; ordonnee = 11.}

# point1.ordonnee ;;
- : float = 23.9

# type ('a , 'b) enreg = { cle: 'a ; valeur: 'b } ;;
type ('a, 'b) enreg = { cle : 'a; valeur : 'b; }

# let x = { cle = "passwd" ; valeur = 1234 } ;;
val x : (string, int) enreg =
                          {cle = "passwd"; valeur = 1234}
```

Notons qu'il est également possible de définir des types *produits cartésiens* avec *constructeur* (utilisation du mot-clé `of`). Dans ce cas, les différentes composantes ne sont pas nommées, on peut définir des fonctions pour y accéder.

Exemple :

```
# type cpoint = ConstrPoint of float * float ;;
type cpoint = ConstrPoint of float * float

# let point1 = ConstrPoint(5.3, 2.0) ;;
val point1 : cpoint = ConstrPoint (5.3, 2.0)

# let abscisse (ConstrPoint(x, y)) = x ;;
val abscisse : cpoint -> float = <fun>

# let ordonnee (ConstrPoint(x, y)) = y ;;
val ordonnee : cpoint -> float = <fun>

# abscisse point1 ;;
- : float = 5.3

# ordonnee point1 ;;
- : float = 2.0
```

III.1.2 Types somme

Les types somme ("ou") caractérisent des données comprenant des alternatives. Les différents membres de la somme, séparés par des |, sont discriminés par des *constructeurs*, qui peuvent avoir des arguments (utilisation du mot-clé `of` suivi du type). On prendra l'habitude de faire commencer les identificateurs de constructeurs par une majuscule, pour des raisons de compatibilité avec Objective Caml.

La déclaration d'un tel type se fait de la façon suivante :

```
type nom = | Const1   (* constructeur constant *)
           | Const2 of arg (* constructeur à un argument *)
           | Const3 of arg1 * arg2 * ...
                          (* const. à plusieurs arguments *)
           | ...   ;;
```

La *création d'une valeur* d'un type somme se fait par *application d'un des constructeurs* à une valeur du type approprié.

Exemples :

```
# type couleur = | Vert | Bleu | Violet ;;
type couleur = Vert | Bleu | Violet

# Bleu ;;
- : couleur = Bleu

# let c = Violet ;;
val c : couleur = Violet

# type valeur =
    | Entier of int
    | Reel of float
    | LePoint of point
    | Couple of int * int ;;
type valeur =
    Entier of int
  | Reel of float
  | LePoint of point
  | Couple of int * int

# Entier 42 ;;
- : valeur = Entier 42

# Reel 83.6 ;;
- : valeur = Reel 83.6

# LePoint {abscisse = 56.2; ordonnee = 11.0} ;;
```

```
- : valeur = LePoint {abscisse = 56.2; ordonnee = 11.0}
# let c = Couple (32 , 4) ;;
val c : valeur = Couple (32, 4)
```

Nous avons dit que les déclarations de type sont par défaut récursives. Cela permet en particulier de définir des types somme dont les arguments de certains constructeurs peuvent être de ce même type.

Exemple :

```
# type mon_type_entier =
      | Zero
      | Plus1 of mon_type_entier ;;
type mon_type_entier = Zero | Plus1 of mon_type_entier
# let x = Plus1 (Plus1 (Plus1 Zero)) ;;
val x : mon_type_entier = Plus1 (Plus1 (Plus1 Zero))
```

cette définition est tout simplement fidèle à la caractérisation des entiers naturels dans l'arithmétique de Peano (élément de base 0 et successeur).

III.2 Filtrage

Nous avons vu dans la section II diverses variantes de définitions de fonctions, et nous avons étudié la définition de fonctions récursives telle qu'elle est réalisée assez classiquement dans tout langage.

Nous allons ici présenter la définition de fonction faisant usage du filtrage de motif (qui peut s'apparenter à une définition par cas), et revenir sur la définition de fonctions récursives dans ce contexte.

III.2.1 Principes et syntaxe

Le filtrage de motif sert à *reconnaître la forme* syntaxique d'une expression. Sa syntaxe est la suivante :

```
match expression with
    | motif1 -> expression1
    | motif2 -> expression2 ...
```

Les différents motifs sont examinés séquentiellement, l'expression correspondant au premier motif reconnu est évaluée. Les motifs doivent être du même type, et les différentes expressions doivent également être du même type.

Le symbole `_` est appelé *motif universel*, il filtre toutes les valeurs possibles, on peut notamment l'utiliser comme dernière alternative.

Plusieurs motifs peuvent être combinés par un `|`, mais chaque motif doit alors être une constante. Un *intervalle de caractères* peut être représenté par `'c1'..'c2'`.

Le filtrage de motif pourra être utilisé dans les *définitions de fonctions*, l'expression située entre `match` et `with` servant dans ce cas à *filtrer la valeur d'un ou plusieurs paramètres*. Attention, l'ensemble des cas possibles pour les valeurs filtrées doit être considéré, éventuellement en ayant recours au motif universel.

Exemples :

```
# let couleur2chaine c = match c with
      | Vert -> "la couleur est Vert"
      | Bleu -> "la couleur est Bleu"
      | Violet -> "la couleur est Violet" ;;
val couleur2chaine : couleur -> string = <fun>

# let coul1 = Bleu ;;
val coul1 : couleur = Bleu

# couleur2chaine coul1 ;;
- : string = "la couleur est Bleu"

# let etlogique a b = match (a, b) with
      | (false, false) -> false
      | (false, true) -> false
      | (true, false) -> false
      | (true, true) -> true ;;
val etlogique : bool -> bool -> bool = <fun>

# etlogique false true ;;
- : bool = false

# let etlogiqueV2 a b = match (a, b) with
      | (false, _) -> false
      | (true, x) -> x ;;
val etlogiqueV2 : bool -> bool -> bool = <fun>

# etlogiqueV2 true false ;;
- : bool = false

# let etlogiqueV3 a b = match (a, b) with
```

```
      | (true, true) -> true
      |      _       -> false ;;
val etlogiqueV3 : bool -> bool -> bool = <fun>

# etlogiqueV3 false false ;;
- : bool = false

# let decode x = match x with
      | '0'..'9' -> "chiffre"
      | 'a' | 'A' -> "lettre a minuscule ou majuscule"
      |    _   -> "lettre autre que a" ;;
val decode : char -> string = <fun>

# decode '7' ;;
- : string = "chiffre"

# decode 'X' ;;
- : string = "lettre autre que a"

# decode 'A' ;;
- : string = "lettre a minuscule ou majuscule"
```

L'exemple de la fonction `couleur2chaine` ci-dessus nous montre un cas de *filtrage sur un type somme*. Ce genre de filtrage est fréquent, la règle générale suivant laquelle l'ensemble des cas possibles doit être prévu s'applique bien entendu.

Exemple :

```
# type valeur =
    | Entier of int
    | Reel of float
    | LePoint of point
    | Couple of int * int ;;
type valeur =
    Entier of int
  | Reel of float
  | LePoint of point
  | Couple of int * int

# let estnul v = match v with
    | Entier 0 | Reel 0.
    | LePoint { abscisse=0. ; ordonnee=0. }
    | Couple (0,0) -> true
    |  _ -> false ;;
val estnul : valeur -> bool = <fun>

# estnul (Couple (0, 5)) ;;
- : bool = false

# estnul (LePoint { abscisse=0. ; ordonnee=0.1 }) ;;
- : bool = false
```

```
# estnul (Entier 0) ;;
- : bool = true
```

Lors du filtrage, le mot-clé `as` permet, si nécessaire, de *nommer le motif* ou une partie du motif.

Notons également qu'il est possible de réaliser du *filtrage avec gardes*, dans lequel une expression conditionnelle peut être évaluée juste après le filtrage d'un motif, et conditionne l'évaluation de l'expression associée à ce motif. On utilise alors la syntaxe :

```
match expression with
    | motif1 when condition1 -> expression1
    | motif2 when condition2 -> expression2 ...
```

Exemples :

```
# type monome = { coeff : float ; degre : int } ;;
type monome = { coeff : float; degre : int; }

# let m1 = { coeff = 3.9 ; degre = 4 } ;;
val m1 : monome = {coeff = 3.9; degre = 4}

# let m2 = { coeff = -. 17.5 ; degre = 2 } ;;
val m2 : monome = {coeff = -17.5; degre = 2}

# let m3 = { coeff = 0. ; degre = 3 } ;;
val m3 : monome = {coeff = 0.; degre = 3}

# let tostring m =
    string_of_float m.coeff ^ " * x^"
                            ^ string_of_int m.degre ;;
val tostring : monome -> string = <fun>

# let estnul m = match m with
     | { coeff = 0. ; degre = d } -> "Monome nul"
     | { coeff = c ; degre = d } as x ->
                    (tostring x) ^ " n'est pas nul" ;;
val estnul : monome -> string = <fun>

# estnul m1 ;;
- : string = "3.9 * x^4 n'est pas nul"

# estnul m3 ;;
- : string = "Monome nul"

# let valabs m = match m with
      | { coeff = c ; degre = d } when c < 0.
                      -> { coeff = -. c ; degre = d }
      | { coeff = c ; degre = d } as x -> x ;;
val valabs : monome -> monome = <fun>
```

```
# valabs m1 ;;
- : monome = {coeff = 3.9; degre = 4}
# valabs m2 ;;
- : monome = {coeff = 17.5; degre = 2}
```

Enfin remarquons que, pour le filtrage de paramètres, une *syntaxe allégée* est souvent préférée à la syntaxe que nous avons utilisée jusqu'ici :

```
function
    | motif1 -> expression1
    | motif2 -> expression2 ...
```

Toutefois l'utilisation de `match` sera parfois plus naturelle, par exemple dans des cas de filtrage sur le premier paramètre seulement.

Exemples :

```
# let couleur2chaine = function
      | Vert -> "la couleur est Vert"
      | Bleu -> "la couleur est Bleu"
      | Violet -> "la couleur est Violet" ;;
val couleur2chaine : couleur -> string = <fun>
# let coul1 = Vert ;;
val coul1 : couleur = Vert
# couleur2chaine coul1 ;;
- : string = "la couleur est Vert"
# let etlogiqueV2 = function
     | (false, _) -> false
     | (true, x) -> x ;;
val etlogiqueV2 : bool * bool -> bool = <fun>
# etlogiqueV2 (false, true) ;;
- : bool = false
# let decode = function
      | '0'..'9' -> "chiffre"
      | 'a' | 'A' -> "lettre a minuscule ou majuscule"
      |  _  -> "lettre autre que a" ;;
val decode : char -> string = <fun>
# decode 'J' ;;
- : string = "lettre autre que a"
# let valabsV2 = function      (* definition alternative *)
    { coeff = c ; degre = d } as x ->
        if c < 0. then { coeff = -. c ; degre = d }
                  else x ;;
```

```
val valabsV2 : monome -> monome = <fun>

# valabsV2 m2 ;;
- : monome = {coeff = 17.5; degre = 2}

# type point = ConstrPoint of float * float ;;
type point = ConstrPoint of float * float

# let abscisse = function
          ConstrPoint(x, y) -> x ;;
val abscisse : point -> float = <fun>

# let ordonnee = function
          ConstrPoint(x, y) -> y ;;
val ordonnee : point -> float = <fun>

# let p = ConstrPoint(5.12, 23.9) ;;
val p : point = ConstrPoint (5.12, 23.9)

# abscisse p ;;
- : float = 5.12

# let simplifmult x y = match x with
      | 0 -> 0
      | 1 -> y
      | _ -> x * y ;;
val simplifmult : int -> int -> int = <fun>

# simplifmult 1 6 ;;
- : int = 6

# simplifmult 8 4 ;;
- : int = 32

# let simplifmult = function
     | 0 -> (function y -> 0)
     | 1 -> (function y -> y)
     | _ as x -> (function y -> x * y) ;;
val simplifmult : int -> int -> int = <fun>

# simplifmult 0 9 ;;
- : int = 0

# simplifmult 90 2 ;;
- : int = 180
```

III.2.2 Filtrage et fonctions récursives

Le filtrage peut également être utilisé dans un contexte de *fonctions récursives*, avec la construction `let rec` vue à la section II.4. Le principe est le même pour la construction par cas, mais la forme est différente.

Par exemple, dans le cas de fonctions sur entiers naturels, on reconnaîtra souvent le motif 0, puis les autres.

Exemples :

```
# let rec factorielle = function
      | 0 -> 1
      | _ as x -> x * factorielle(x-1) ;;
val factorielle : int -> int = <fun>

# factorielle 7 ;;
- : int = 5040

# let rec pair = function
      | 0 -> true
      | _ as x -> impair(x-1)
  and impair = function
      | 0 -> false
      | x -> pair(x-1) ;;
val pair : int -> bool = <fun>
val impair : int -> bool = <fun>

# pair 5 ;;
- : bool = false

# impair 73 ;;
- : bool = true

# let rec pair = function    (* def. alternative *)
    | 0 -> true
    | 1 -> false
    | _ as x -> pair(x-2) ;;
val pair : int -> bool = <fun>

# pair 12 ;;
- : bool = true

# pair 49 ;;
- : bool = false

# let rec fibonacci = function
      | 0 -> 1
      | 1 -> 1
      | n -> fibonacci(n-1) + fibonacci(n-2) ;;
val fibonacci : int -> int = <fun>

# fibonacci 10 ;;
- : int = 89
```

Enfin, cette forme de fonctions récursives avec filtrage est très répandue dans le cas de *fonctions sur listes*. La plupart des définitions consistent à filtrer le motif correspondant à la *liste vide* `[]`, puis le motif associé à une *liste de tête* ***x*** *et de queue* *l*, c'est à dire `x::l`.

Exemples :

```
# let rec longueur = function
      | [] -> 0
      | x::l -> 1 + longueur(l) ;;
val longueur : 'a list -> int = <fun>

# longueur [ 45; 7 ; 88 ; 13 ] ;;
- : int = 4

# let rec fois2 = function
      | [] -> []
      | tete::queue -> (2 * tete)::fois2(queue) ;;
val fois2 : int list -> int list = <fun>

# fois2 [ 45; 7 ; 88 ; 13 ] ;;
- : int list = [90; 14; 176; 26]
```

Remarque : la possibilité de définition de types somme permet de définir par des constructeurs des structures récursives plus élaborées que les listes (arbres, graphes,...). Les fonctions sur ces structures mettent en oeuvre un filtrage qui suit un raisonnement similaire au filtrage ci-dessus.

QUELQUES COMPLEMENTS

Dans cette section annexe, nous présentons rapidement quelques compléments de nature un peu plus technique : quelques mots sur les structures modifiables et effets de bord, entrées/sorties conversationnelles, chargement de fichiers et compilation.

Structures modifiables

Parmi les structures de données que nous avons présentées, certaines sont ou peuvent être modifiables. Disons quelques mots à ce sujet, dans le cas des chaînes de caractères et des enregistrements.

Les *chaînes de caractères* (section I.2) sont des structures physiquement modifiables (`String.set` ou `<-`), on rappelle que la notation `.[ ]` permet l'accès aux caractères de la chaîne.

Exemple :

```
# let s = "Bonjour " ^ "Jim" ;;
val s : string = "Bonjour Jim"

# s.[0] <- 'b' ;;
- : unit = ()

# String.set s 8 'j' ;;
- : unit = ()

# s ;;
- : string = "bonjour jim"
```

Par défaut, les champs des *enregistrements* (section III.1.1) ne sont pas modifiables ("mutable"). Le mot-clé `mutable` permet de les rendre modifiables.

Exemple :

```
# type point = { abscisse: float ; ordonnee: float } ;;
type point = { abscisse : float; ordonnee : float; }

# let point1 = { abscisse = 8.4 ; ordonnee = 23.9 } ;;
val point1 : point = {abscisse = 8.4; ordonnee = 23.9}
```

```
# point1.abscisse <- 0. ;;
The record field label abscisse is not mutable

# type point = { mutable abscisse: float ;
                 ordonnee: float } ;;
type point = { mutable abscisse : float;
               ordonnee : float; }

# let point2 = { abscisse = 4.62 ; ordonnee = 10.3 } ;;
val point2 : point = {abscisse = 4.62; ordonnee = 10.3}

# point2.abscisse <- 1. ;;
- : unit = ()

# point2.ordonnee <- 22.8 ;;
The record field label ordonnee is not mutable

# point2 ;;
- : point = {abscisse = 1.; ordonnee = 10.3}
```

Entrées/sorties

Il est possible de réaliser des *affichages sur la sortie standard* (`std_out`) au moyen des fonctions `print_string`, `print_int`, `print_float`,... La fonction `print_newline` affiche un retour à la ligne. La *saisie d'une chaîne de caractères* sur l'entrée standard (`std_in`) se fait grâce à la fonction `read_line`.

Exemple :

```
# let masaisie_chaine () =
    print_string "   prompt> ";
    read_line();;
val masaisie_chaine : unit -> string = <fun>

# let chainelue = masaisie_chaine() ;;
   prompt> salut
val chainelue : string = "salut"

# chainelue ;;
- : string = "salut"

# let masaisie_float () =
    print_string "   prompt> ";
    float_of_string (read_line()) ;;
val masaisie_float : unit -> float = <fun>

# let floatlu = masaisie_float() ;;
   prompt> 6.31
val floatlu : float = 6.31
```

Chargement, compilation

Plutôt que de saisir vos fonctions interactivement sous l'interpréteur, votre programme peut être placé dans un fichier, qu'il vous suffit de *charger sous l'interpréteur* grâce à la directive `#use`.

Exemple :

```
mamachine% cat facto.ml
let fact n =
   let rec factorielle x =
     if x = 0 then 1 else x * factorielle(x-1) in
if n >= 0 then factorielle n else 0 ;;
mamchine% ocaml
      Objective Caml version 3.08.3

# #use "facto.ml" ;;
val fact : int -> int = <fun>
# fact 7 ;;
- : int = 5040
#
```

Au lieu d'exécuter vos programmes sous l'interpréteur Caml, vous pouvez les *compiler* puis les exécuter sous l'interpréteur de commandes (le shell). Une compilation avec `ocamlc` produit un exécutable pour la machine virtuelle OCaml (code portable), tandis qu'une compilation avec `ocamlopt` produit du code natif optimisé.

Attention, ici vous n'êtes plus dans la boucle d'évaluation/affichage de l'interpréteur Caml. Il faut donc prévoir les entrées/sorties conversationnelles nécessaires au bon fonctionnement de votre programme, en particulier ne pas oublier de faire explicitement afficher les résultats.

Exemple :

```
mamachine% cat facto.ml
let fact n =
   let rec factorielle x =
     if x = 0 then 1 else x * factorielle(x-1) in
if n >= 0 then factorielle n else 0 ;;
```

```
print_string "on calcule fact(9) :" ;; print_newline() ;;

print_string "resultat = " ;;
print_int (fact 9) ;;  print_newline() ;;
```

```
mamchine% ocamlc -o facto facto.ml
```

```
mamchine% ocamlrun facto
```

```
on calcule fact(9) :
resultat = 362880
```

```
mamchine% ocamlopt -o factoopt facto.ml
```

```
mamchine% ./factoopt
```

```
on calcule fact(9) :
resultat = 362880
```

Programmation modulaire, compilation séparée

Tout comme en C par exemple, votre programme peut être *modulaire*, c'est à dire que les différentes fonctions peuvent être placées dans des *unités de compilation* différentes. Une unité de compilation est composée de deux fichiers, le fichier d'implantation (`.ml`) et le fichier, optionnel, d'interface (`.mli`). La compilation d'un fichier `monfic.ml` produit un module `Monfic`, ses fonctions peuvent être utilisées dans un autre fichier par notation pointée ou en utilisant `open`.

La compilation avec l'option `-c` produit un fichier de code objet `.cmo` et un fichier d'interface compilée `.cmi`. L'exécutable final sera produit par l'édition de liens des fichiers `.cmo`. Cela peut vous conduire tout naturellement à élaborer des `Makefile` pour vos applications Caml...

Exemple :

```
mamachine% ls -l
total 4224
-rw-r--r--  1 moi  staff      150 Sep 27 09:20 cnp.ml
-rw-r--r--  1 moi  staff      129 Sep 27 09:15 facto.ml
```

```
mamachine% cat facto.ml
```

```
let fact n =
   let rec factorielle x =
     if x = 0 then 1 else x * factorielle(x-1) in
if n >= 0 then factorielle n else 0 ;;
```

```
mamachine% cat cnp.ml
open Facto ;;

let comb n p =
    ((fact n) * (fact p)) / (fact (n - p)) ;;

print_string "C 6 2 = " ;;
print_int (comb 6 2) ;;
print_newline() ;;
mamachine% ocamlc -c facto.ml
mamachine% ocamlc -c cnp.ml
mamachine% ls -l
total 3256
-rw-r--r--  1 moi  staff     220 Sep 27 09:28 cnp.cmi
-rw-r--r--  1 moi  staff     444 Sep 27 09:28 cnp.cmo
-rw-r--r--  1 moi  staff     150 Sep 27 09:20 cnp.ml
-rw-r--r--  1 moi  staff     173 Sep 27 09:28 facto.cmi
-rw-r--r--  1 moi  staff     291 Sep 27 09:28 facto.cmo
-rw-r--r--  1 moi  staff     129 Sep 27 09:15 facto.ml
mamachine% ocamlc -o moncnp facto.cmo cnp.cmo
mamachine% ocamlrun moncnp
C 6 2 = 60
```

Les *interfaces de modules* peuvent être utilisées pour déclarer les identificateurs du module que l'on veut rendre visibles à l'extérieur. Les identificateurs définis dans le module mais non déclarés dans l'interface ne seront pas visibles. Ce sont des fichiers `.mli` dans lesquels les déclarations d'identificateurs sont introduites par le mot-clé `val` et précisent le nom et le type de l'identificateur (on pourra aussi y placer des commentaires sur sa nature et son utilisation !...). Ces fichiers sont compilés avant les fichiers `.ml`, leur compilation donne naissance aux fichiers `.cmi`.

Exemple :

```
mamachine% ls -l
total 4224
-rw-r--r--  1 moi  staff     150 Sep 27 09:20 cnp.ml
-rw-r--r--  1 moi  staff     129 Sep 27 09:15 facto.ml
-rw-r--r--  1 moi  staff      51 Sep 27 09:49 facto.mli
mamachine% cat facto.mli
(* Fonction factorielle *)
val fact: int -> int ;;
mamachine% ocamlc -c facto.mli
```

```
mamachine% ls -l
total 2724
-rw-r--r--  1 moi  staff      150 Sep 27 09:20 cnp.ml
-rw-r--r--  1 moi  staff      173 Sep 27 10:01 facto.cmi
-rw-r--r--  1 moi  staff      129 Sep 27 09:15 facto.ml
-rw-r--r--  1 moi  staff       51 Sep 27 09:49 facto.mli

mamachine% ocamlc -c facto.ml

mamachine% ls -l
total 3224
-rw-r--r--  1 moi  staff      150 Sep 27 09:20 cnp.ml
-rw-r--r--  1 moi  staff      173 Sep 27 10:01 facto.cmi
-rw-r--r--  1 moi  staff      295 Sep 27 10:02 facto.cmo
-rw-r--r--  1 moi  staff      129 Sep 27 09:15 facto.ml
-rw-r--r--  1 moi  staff       51 Sep 27 09:49 facto.mli

mamachine% ocamlc -c cnp.ml

mamachine% ocamlc -o moncnp facto.cmo cnp.cmo

mamachine% ocamlrun moncnp
C 6 2 = 60
```

Parmi les sources de ce cours...

✓ "Le langage Caml", P.Weis et X.Leroy, Dunod.

✓ http://caml.inria.fr

✓ http://www.bath.ac.uk/~cs1cb/ML

✓ http://burks.brighton.ac.uk/burks/language/ml/

✓ http:/www-calfor.lip6.fr/~vmm/Enseignement/Licence_info/Programmation/Cours/

✓ http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/