**Expression Logique et Fonctionnelle . . . Évidemment**

# Cours n° 1 : Introduction à la programmation fonctionnelle

## 1 Introduction

La *programmation fonctionnelle* est un style de programmation dont les concepts fondamentaux sont les notions de *valeur*, de *fonction* et d'*application* d'une fonction à un valeur.

Le terme *fonction* est à prendre ici au sens mathématique du terme, c'est-à-dire une relation entre deux ensembles $A$ et $B$ telle que tout élément de l'ensemble $A$ soit en relation avec au plus un élément de l'ensemble $B$. Si on nomme $f$ cette fonction, et $x$ un élément de $A$ on note habituellement $f(x)$ l'élement de $B$ associé à $x$ et on dit que $f(x)$ est l'*image* de $x$ par $f$ si cette image est définie.

**Exemple 1 :**
Voici quelques exemples de fonctions

1. une fonction numérique
$$\begin{array}{rccc} \texttt{f} : & \mathbb{R} & \longrightarrow & \mathbb{R} \\ & x & \longmapsto & f(x) = \frac{\sqrt{x^2+1}}{x-1} \end{array} .$$
Les ensembles $A$ et $B$ sont ici tous deux égaux à l'ensemble $\mathbb{R}$ des nombres réels. Tout réel $x \neq 1$ a une image par `f`.
2. la fonction qui à un mot sur un alphabet $\mathcal{A}$ associe sa longueur
$$\begin{array}{rccc} \texttt{length} : & \mathcal{A}^* & \longrightarrow & \mathbb{N} \\ & u & \longmapsto & |u| \end{array} .$$
L'ensemble $A$ est l'ensemble $\mathcal{A}^*$ des mots sur l'alphabet $\mathcal{A}$, et l'ensemble $B$ est l'ensemble $\mathbb{N}$ des entiers naturels. Tout mot a une image par cette fonction.
3. la fonction qui à deux entiers associe leur somme
$$\begin{array}{rccc} \texttt{add} : & \mathbb{Z} \times \mathbb{Z} & \longrightarrow & \mathbb{Z} \\ & (n_1, n_2) & \longmapsto & n_1 + n_2 \end{array} .$$
$A$ est l'ensemble $\mathbb{Z} \times \mathbb{Z}$ des couples de nombres entiers, et $B$ est l'ensemble $\mathbb{Z}$ des entiers. Tout couple d'entiers possède une image par la fonction `add`.

Beaucoup de langages de programmation ont intégré la notion de fonctions dans leurs constructions, et permettent donc de programmer des fonctions. C'est le cas du langage C par exemple, dans lequel la première des trois fonctions s'écrit

```
double f(double x) {
  return sqrt(x*x + 1)/(x - 1) ;
}
```

Une fois de telles fonctions écrites, il est possible de les appliquer à des valeurs dans une expression du programme. Pour reprendre notre exemple en C qui précède, si plus loin dans le programme on trouve l'expression `f(3)`, cela signifie que l'on *applique* la fonction `f` au nombre 3, afin d'en obtenir la valeur associée (approximativement 1.581139). De ce point de vue, C peut à la rigueur être considéré comme un langage fonctionnel.

**Transparence référentielle :** Du point de vue mathématique, l'image d'un élément $x \in A$ d'une fonction $f : A \to B$ est un élément $y \in B$ bien précis (si elle existe). Autrement dit, $f(x)$ est égal à $y$, et parler de $f(x)$ ou de $y$ revient au même. Replacé dans un contexte de programmation, cela signifie que normalement écrire $f(x)$, ou bien écrire $y$ dans le programme doit revenir au même.

Cependant, ce n'est pas le cas de fonctions couramment écrites en programmation. En voici un exemple (toujours en C).

```
int i = 0 ;

int g(int j) {
  i += j ;
  return i ;
}
```

Au premier appel à la fonction `g` avec l'entier 1 passé en paramètre, l'expression `g(1)` est évaluée en 1. Au second appel, toujours avec l'entier 1, l'expression `g(1)` est évaluée en 2. Il apparaît donc que la valeur de `g(1)` n'est pas précisée une fois pour toute par une relation fonctionnelle au sens mathématique du terme. Cette valeur dépend du contexte d'évaluation. Cela provient de l'apparition d'une variable globale `i` dont la valeur est changée à chaque appel à `g`. Cette propriété de la fonction `g` rend donc impossible la possibilité de remplacer dans un programme l'expression `f(i)` par une quelconque valeur. Pire, l'expression `f(i) - f(i)` ne peut pas être remplacée par 0 [1].

> La *transparence référentielle* est la propriété d'un langage (ou d'un programme) qui fait que toute variable, ainsi que tout appel à une fonction, peut être remplacée par sa valeur sans changer le comportement du programme.

Une expression fonctionnelle d'un programme doit satisfaire cette propriété de transparence référentielle [2].

Bien entendu, la transparence référentielle n'est pas respectée si les variables d'un programme peuvent changer de valeurs, ce qui est le cas dans les langages d'expression impérative.

**Les fonctions, des valeurs au même titre que les autres :** Dans leur travaux les mathématiciens manipulent des fonctions, et les combinent pour en produire de nouvelles : somme de deux fonctions, composition de fonctions, dérivation, etc... Les informaticiens, eux, manipulent des programmes et les combinent en les assemblant pour produire de nouveaux programmes. Mais ces combinaisons de programmes ne sont le plus souvent pas des constructions de programmes décrits dans le langage lui-même car les programmes ne sont pas des données au même titre que les entiers, les booléens, etc...

Par exemple, les nombres entiers sont des données de base existant dans la plupart des langages de programmation. Il n'est pas besoin de nommer un entier pour pouvoir l'utiliser. En C, comme dans beaucoup d'autres langages de programmation, l'expression `3 + 1` désigne (le plus souvent) l'entier 4 obtenu en additionnant les entiers 3 et 1. Nous n'avons pas besoin de nommer les deux entiers de départ, et il n'est pas nécessaire de nommer l'entier résultant de la somme.

En revanche, dans les langages non fonctionnels, les fonctions doivent obligatoirement être nommées. La syntaxe des langages l'oblige. C'est le cas des deux fonctions précédentes écrites en C. Le nommage est obligatoire

- pour les définir,
- et pour les appliquer.

Une caractéristique des langages fonctionnels est que les fonctions sont des données au même titre que toute autre données, qui peuvent être construites et utilisées comme toute autre données. En particulier, il n'est pas nécessaire de les nommer, tout comme on ne nomme pas nécessairement l'entier 1 dans une instruction d'incrémentation d'un compteur, et elles peuvent être passées en paramètres à une fonction et/ou en être le résultat.

> Dans un langage fonctionnel, les fonctions doivent être des données au même titre que toutes autres données.

Distinction entre programmation impérative et programmation fonctionnelle :

**Paradigme impératif.** Celui-ci repose sur les notions d'*état d'une machine* (i.e. état de la mémoire), d'*instructions*, de *séquence* (ordonnancement) et d'*itération*. Il nécessite d'avoir une connaissance minime du modèle de machine sur laquelle on programme (modèle de *Von Neumann*).

1. ce qui interdit donc tout travail d'optimisation
2. Cela interdit donc des fonctions comme les générateurs de nombres pseudo-aléatoires, ou bien les fonctions de lectures de données

**Paradigme fonctionnel.** Celui-ci repose sur les notions *valeurs*, *expressions* et *fonctions*. L'exécution d'un programme est l'*évaluation* d'une ou plusieurs expressions, expressions qui sont souvent des applications de fonctions à des valeurs passées en paramètre [3].

# 2 Quelques langages fonctionnels

Il est parfois difficile d'établir une frontière bien nette entre langages fonctionnels et langages non fonctionnels, parce que nombre de langages fonctionnels présentent des traits qui ne le sont pas (c'est le cas de CAML par exemple), et nombre de langages non fonctionnels possèdent des aspects qui le sont. Néanmoins, certains langages sont universellement connus comme étant essentiellement des langages fonctionnels. En voici quelques uns.

1. LISP, l'ancêtre de tous les langages fonctionnels, créé en 1958 par John McCarthy. Très utilisé en intelligence artificielle.
2. SCHEME, dérivé de LISP, créé en 1975 par Gerald Jay Sussman et Guy L. Steele.
3. La famille des langages ML (Meta Language), issue des travaux de Robin Milner de la fin des années 70, dont les deux principaux représentants aujourd'hui sont SML (Standard ML) et CAML (dans sa version OCAML).
4. HASKELL, langage fonctionnel pur. Créé en 1990. Une de ses particularités est d'être un langage à évaluation *paresseuse* (lazy).

Ces langages se distinguent selon certaines caractéristiques :

**Langages fonctionnels pur vs. impurs.** Un langage fonctionnel sans effet de bord est dit langage *fonctionnel pur*. Par exemple, dans de tels langages il n'y a pas d'opération d'affection. Les langages purs restent assez rares, citons HASKELL parmi eux. LISP, SCHEME, la famille des langages ML sont impurs.

**Typage statique vs. dynamique.** Les arguments d'une fonction doivent avoir un type compatible avec celui de la fonction pour que l'évaluation d'une application puisse être réalisée. La vérification du type, ou typage, peut se faire à l'exécution du programme, on parle alors de *typage dynamique*. C'est le cas de langages comme LISP ou SCHEME. Inversement, le typage peut avoir lieu avant l'exécution du programme, lors de la compilation, le typage est alors qualifié de *statique*. C'est le cas des langages de la famille ML.

**Vérification vs. inférence de types.** La plupart des langages de programmation impératifs typés s'appuie sur un *typage explicite* : le programmeur donne toutes les informations de types liées aux déclarations d'identificateurs. Le compilateur, ou machine d'exécution, s'appuie alors sur cette information pour *vérifier* (statiquement ou dynamiquement) que les identificateurs sont utilisés conformément au type déclaré. La plupart des langages fonctionnels, dont OCAML, libèrent le programmeur des annotations de types. Les langages de la famille ML font de l'*inférence de types* par l'analyse du code dans la phase de compilation.

Le langage utilisé dans ce cours est Objective-Caml (OCAML), dialecte de la famille ML. OCAML est développé à l'INRIA [4] et librement distribué (`http://caml.inria.fr`). Ce langage est très riche et offre
- un noyau fonctionnel (Core ML) ;
- des structures de contrôle impératives, ainsi que des types de données mutables ;
- un système de *modules* ;
- la possibilité de programmation par *objets*.

Les modules et les objets d'OCAML ne seront pas abordés dans ce cours et nous nous limiteront à l'apprentissage de CAML et de son «noyau théorique» Core ML.

# 3 CAML : petit tour d'horizon du langage

## 3.1 Organisation d'un programme

Un programme en CAML est une suite de *phrases*. Ces phrases peuvent être

1. des *directives*,

3. On dit aussi *programmation applicative*
4. Institut National de Recherche en Informatique et Automatique

2. des *déclarations* de variables ou de types,
3. des *expressions* à évaluer.

Les phrases de ce programme peuvent

1. être écrites une à une en dialoguant avec l'interpréteur du langage,
2. ou bien être toutes écrites une fois pour toute à l'aide d'un éditeur de textes, puis être évaluées en une seule fois.

L'exécution d'un programme consiste en l'évaluation de toutes les phrases.

## 3.2 Premier dialogue avec l'interpréteur OCaml

Ci-dessous quelques exemples de phrases évaluées par l'interprèteur du langage OCaml.
Le symbole # est le prompt de la boucle d'interaction, ce qui suit est la phrase à évaluer. Cette phrase est suivie d'un double point-virgule (`;;`).
Les lignes débutant par le symbole – sont les résultats de l'évaluation de la phrase.

```
# 132 ;;
- : int = 132
# 2*(45+10) ;;
- : int = 110
# 7/4 ;;
- : int = 1
# 7 mod 4 ;;
- : int = 3
# 2. ;;
- : float = 2.
# 2. +. 3. ;;
- : float = 5.
# 7./.4. ;;
- : float = 1.75
# float_of_int 2 ;;
- : float = 2.
# true ;;
- : bool = true
# true & false ;;
- : bool = false
# 1 = 2-1 ;;
- : bool = true
# "Hello" ;;
- : string = "Hello"
# "Hello " ^ "le monde " ;;
- : string = "Hello le monde "
# 'A' ;;
- : char = 'A'
# int_of_char 'B' ;;
- : int = 66
```

Comme vous pouvez le constater, les réponses de l'interpréteur précisent à chaque fois le *type* de l'expression à évaluer suivi de sa *valeur*.

## 3.3 Types de base

Caml, comme tous les langages de la famille ML, est un langage fortement typé, à typage statique. Avant d'évaluer une phrase, le type de l'expression est calculé, c'est l'*inférence de type* : contrairement à des langages comme C, Pascal, Ada ou Java, le programmeur n'a pas besoin de préciser le type de ces expressions, variables ou fonctions. Le type d'une expression est calculé à partir de ses composants. Si l'inférence de type échoue, l'expression n'est pas évaluée. Si en revanche elle réussit, l'expression peu être évaluée.
Voici les types de base du langage :

`int` : les entiers. L'intervalle du type `int` dépend de l'architecture de la machine utilisée. C'est l'intervalle $[\![-2^{30}, 2^{30}-1]\!]$ sur les machines 32 bits. Deux variables prédéfinies () donnent les valeurs de la plus petite et de la plus grande valeur de type `int`.

```
# min_int ;;
- : int = -1073741824
# max_int ;;
- : int = 1073741823
```

| Opérateur | signification |
|---|---|
| `+` | addition |
| `-` | soustraction |
| `*` | multiplication |
| `/` | division entière |
| **`mod`** | reste de la division entière |

`float` : les nombres flottants. Ils respectent la norme IEEE 754. OCaml distingue les entiers des flottants : `2` est de type `int`, tandis que `2.` est de type `float`, et ces deux valeurs ne sont pas égales. Même les opérateurs arithmétiques sur les flottants sont différents de ceux sur les entiers.

| Opérateur | signification |
|---|---|
| `+.` | addition |
| `-.` | soustraction |
| `*.` | multiplication |
| `/.` | division |
| `**` | exponentiation |

On ne peut pas additionner un `int` et un `float`

```
# 2 + 3. ;;
Characters 4-6:
  2 + 3. ;;
      ^^
This expression has type float but is here used with type int
# 2. +. 3 ;;
Characters 6-7:
  2. +. 3 ;;
        ^
This expression has type int but is here used with type float
```

| Fonction | signification |
|---|---|
| `float_of_int` | conversion d'un `int` en `float` |
| `ceil` | partie entière supérieure (plafond) |
| `floor` | partie entière inférieure (plancher) |
| `sqrt` | racine carrée |
| `exp` | exponentielle |
| `log` | logarithme népérien |
| `log10` | logarithme décimal |
| ... | ... |

`string` : les chaînes de caractères (longueur limitée à $2^{64}-6$).

| Opérateur | signification |
|---|---|
| `^` | concaténation |

| Fonction | signification |
|---|---|
| `string_of_int` | conversion d'un `int` en un `string` |
| `int_of_string` | conversion d'un `string` en un `int` |
| `string_of_float` | conversion d'un `float` en un `string` |
| `float_of_string` | conversion d'un `string` en un `float` |
| `String.length` | longueur |

`char` : les caractères. Au nombre de 256, les 128 premiers suivent le code ASCII.

| Fonction | signification |
|---|---|
| `char_of_int` | conversion d'un `int` en un `char` |
| `int_of_char` | conversion d'un `char` en un `int` |

`bool` : les booléens pouvant prendre deux valeurs **`true`** et **`false`**.

| Opérateur | signification |
|---|---|
| `not` | négation |
| `&&` ou `&` | et séquentiel |
| `\|\|` ou **`or`** | ou séquentiel |

Les opérateurs de comparaison sont *polymorphes*, ils sont utilisables pour comparer deux `int`, ou deux `float`, ou deux `string`,.... Mais ils ne permettent pas la comparaison d'un `int` et un `float`.

| Opérateur | signification |
|---|---|
| `=` | égalité (structurelle) |
| `<>` | négation de `=` |
| `<` | inférieur |
| `<=` | inférieur ou égal |
| `>` | supérieur |
| `>=` | supérieur ou égal |

`unit` : le type `unit` est un type ne possédant qu'une seule valeur notée `()`. Ce type est utilisé dans la partie impérative du langage OCaml, et par les différentes procédures d'impression comme par exemple la procédure `Printf.printf`.

## 3.4 Types $n$-uplets

À partir des types de base, il y a plusieurs façons de construire des données plus élaborées. L'un de ces moyens est de former des $n$-uplets.

En Caml, on construit des $n$-uplets en juxtaposant des expressions séparées par des virgules. Le tout pouvant être mis entre parenthèses.

En voici deux exemples.

```
# (1,-3) ;;
- : int * int = (1, -3)
#("ELFE", true, 5. *. 3.) ;;
- : string * bool * float = ("ELFE", true, 15.)
```

Le premier est un couple de deux entiers, le second un triplet dont la première composante est une chaîne de caractères, la deuxième un booléen, et la troisième un flottant. Comme on le constate, les types des différentes composantes ne sont pas nécessairement les mêmes.

Les expressions qui suivent montrent des constructions de couples de types différents. La tentative échouée de comparaison prouve cette différence.

```
# (("ELFE",true), 5. *. 3.) ;;
- : (string * bool) * float = (("ELFE", true), 15.)
# ("ELFE",(true, 5. *. 3.)) ;;
- : string * (bool * float) = ("ELFE", (true, 15.))
# ("ELFE",(true, 5. *. 3.)) = (("ELFE", true), 5. *. 3.) ;;
Error: This expression has type (string * bool) * float
       but an expression was expected of type string * (bool * float)
```

Les $n$-uplets sont construits à l'aide du constructeur virgule. Les types des composantes peuvent être hétérogènes. Les composantes peuvent elles-mêmes être des $n$-uplets. Le type général d'un $n$-uplet est

$$a_1 * a_2 * \ldots * a_n.$$

## 3.5 Type liste

Les listes de Caml sont des structures de données linéaires non mutables permettant le regroupement d'un nombre quelconque de valeurs de même type.

La construction d'une liste peut se faire par énumération de ses éléments :

```
# [3; 1; 4; 1; 5; 9; 2] ;;
- : int list = [3; 1; 4; 1; 5; 9; 2]
# [true; false]
- : bool list = [true; false]
```

```
# [[3]; [1; 4]; [1; 5; 9; 2]] ;;
- : int list list = [[3]; [1; 4]; [1; 5; 9; 2]]
```

Une liste peut être vide

```
# [] ;;
- : 'a list = []
```

Le constructeur `::` permet la construction d'une liste par ajout d'un élément en tête d'une liste.

```
# 3::[1; 4] ;;
- : int list = [3; 1; 4]
# 3::[] ;;
- : int list = [3]
```

Une liste non vide en programmation fonctionnelle doit souvent être considérée comme un couple (*tête*, *reste*), où *tête* est le premier élément de la liste, et *reste* le reste de la liste. L'accès à chacune de ces deux composantes d'une liste non vide peut se faire en CAML avec les fonctions `List.hd` et `List.tl`.

```
# List.hd [3; 1; 4] ;;
- : int = 3
# List.tl [3; 1; 4] ;;
- : int list = [1; 4]
# List.hd [[3]; [1; 4]; [1; 5; 9; 2]] ;;
- : int list = [3]
# List.tl [[3]; [1; 4]; [1; 5; 9; 2]] ;;
- : int int list = [[1; 4]; [1; 5; 9; 2]]
```

Les listes sont construites à partir du constructeur `::` qui construit une nouvelle liste en ajoutant un élément en tête d'une liste existante. Les éléments d'une liste sont homogènes du point de vue du type. Le type général d'une liste est

`'a list`.

La sélection de l'élément de tête se fait avec la fonction `List.hd` et celle du reste avec la fonction `List.tl`.

## 3.6 Déclaration de variables

En programmation fonctionnelle, une *variable* est une liaison entre un nom et une valeur. Les variables peuvent être *globales*, et elles sont alors connues de toutes les expressions qui suivent la déclaration, ou bien *locales* à une expression, et dans ce cas elles ne sont connues que dans l'expression pour laquelle elles ont été déclarées.

L'ensemble des variables connues d'une expression est appelé *environnement* de l'expression.

La syntaxe d'une déclaration globale est de la forme

```
let <nom> = <expr>
```

où `<nom>` est le nom de la variable (en CAML les noms de variables doivent obligatoirement commencer par une lettre minuscule), et `<expr>` une expression décrivant la valeur de la variable.

```
# let n = 12 ;;
val n : int = 12
# n ;;
- : int = 12
# 2 * n;;
- : int = 24
# let m = 2 * n ;;
val m : int = 24
# m + n;;
- : int = 36
# m + p;;
Characters 2-3:
m+p;;
  ^
Unbound value p
```

Pour être typable, et donc évaluable, toutes les variables d'une expression doivent être définies dans l'environnement de l'expression, sous peine d'avoir le message d'erreur `Unbound value`.

On peut aussi déclarer simultanément plusieurs variables.

```
let <nom1> = <expr1>
and <nom2> = <expr2>
  ...
and <nomn> = <exprn>
```

## 3.7 Déclaration locale de variables

Syntaxe des déclarations locales

```
let <nom> = <expr1> in
   <expr2>
```

Cette forme syntaxique permet de déclarer la variable nommée `<nom>` dans l'environnement d'évaluation de l'expression `<expr2>`. En dehors de l'environnement de cette expression, la variable `<nom>` n'existe pas.

Il est possible de faire des déclarations simultanées de variables locales à une expression.

```
let <nom1> = <expr1>
and <nom2> = <expr2>
  ...
and <nomn> = <exprn> in
   <expr>
```

```
# let pi = 3.141592
  and r = 2.0 in
    2. *. pi *. r ;;
- : float = 12.566368
# pi ;;
Characters 0-2:
pi ;;
^^
Unbound value pi
```

## 3.8 Expression conditionnelle

La syntaxe d'une expression conditionnelle est

```
  if <expr1> then <expr2> else <expr3>
```

dans laquelle

- `<expr1>` est une expression booléenne (type `bool`),
- `<expr2>` et `<expr3>` ont le même type.

```
# if 1=1 then "egaux" else "non egaux" ;;
- : string = "egaux"
# if 1=2 then "egaux" else "non egaux" ;;
- : string = "non egaux"
# if 1=2 then 0 else "non egaux" ;;
Characters 19-30:
if 1=2 then 0 else "non egaux" ;;
                   ^^^^^^^^^^^
This expression has type string but is here used with type int
```

**Remarques :**

- en CAML, la structure de contrôle conditionnelle est une expression et non une instruction, c'est la raison pour laquelle les deux expressions des parties **then** et **else** doivent être du même type.
- en CAML, il est possible d'omettre la partie **else** d'une expression conditionnelle. Dans ce cas, elle est considérée comme étant la valeur `()` (type `unit`), et la partie **then** doit elle aussi être du type `unit`. De part son type `unit` imposé, cette forme d'expression conditionnelle ne se rencontre en général pas en programmation fonctionnelle pure.

## 3.9 Fonctions

En CAML, comme dans tous les langages fonctionnels, les fonctions sont des valeurs comme toutes les autres. Une variable peut donc avoir une fonction pour valeur.

```
# let racine_carree = sqrt ;;
val racine_carree : float -> float = <fun>
# racine_carree 2. ;;
- : float = 1.41421356237309515
```

### 3.9.1 Type d'une valeur fonctionnelle

Le type d'une fonction est déterminé par le type de son (ou ses) argument(s) et celui des valeurs qu'elle renvoie. Ainsi le type de la fonction `sqrt` est `float -> float` puisqu'elle prend un `float` pour paramètre et retourne un `float`.

### 3.9.2 Les valeurs fonctionnelles

Le premier mot-clé permettant de construire une fonction est **function** :.

```
function <param> -> <expr>
```

où <param> est le nom donné au paramètre formel, et <expr> est l'expression qui définit cette fonction.

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

Dans l'exemple ci–dessus, l'évaluation de la phrase donne comme résultat une valeur fonctionnelle de type `int` $\rightarrow$ `int`. CAML signale les valeurs fonctionnelles en imprimant **<fun>** après le type.

Cette fonction qui vient d'être construite n'est pas nommée : c'est une *fonction anonyme*.

Il est bien entendu possible de nommer une fonction en déclarant une variable (globale ou non).

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

Une autre façon équivalente pour déclarer la même fonction consiste à écrire

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

Cette syntaxe est du *sucre syntaxique*.

### 3.9.3 Application d'une fonction

Pour appliquer une fonction à une valeur <arg>, il suffit d'écrire le nom de la fonction suivi de l'expression

```
<fonct> <arg>
```

```
# carre 2 ;;
- : int = 4
# carre (2 + 2) ;;
- : int = 16
```

Attention aux parenthèses qui permettent de contrôler la priorité des opérateurs

```
# carre 2 + 2 ;;
- : int = 6
```

L'application d'une fonction à une valeur est prioritaire à toute autre opération.

Attention aussi aux nombres négatifs écrits litéralement

```
# carre -1 ;;
  ^^^^^
Error: This expression has type int -> int
       but an expression was expected of type int
```

Erreur qui provient d'une interprétation du signe - comme étant le symbole de soustraction. CAML refuse de soustraire 1 à la fonction `carre` car cette fonction n'est pas une valeur de type `int`.

### 3.9.4 Application d'une fonction anonyme

```
# (function x -> x * x) 2 ;;
- : int = 4
```

### 3.9.5 Fonctions à plusieurs paramètres

En programmation fonctionnelle, deux points de vue sont concevables pour les fonctions à plusieurs paramètres :

1. comme des fonctions qui s'appliquent à des $n$-uplets,
2. ou bien comme des fonctions qui s'appliquent à leur premier paramètre et qui renvoient une fonction s'appliquant sur les paramètres restant.

Par exemple la fonction d'addition de deux nombres entiers peut être définie en CAML sous la forme

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

Le type de cette fonction ainsi déclarée montre qu'elle s'applique à des couples d'entiers. Tout appel à cette fonction doit donc se faire sur un couple d'entiers.

```
# add (3,-6) ;;
- : int = -3
```

Cette façon de programmer la fonction `add` est celle couramment employée dans la plupart des langages. Mais en programmation fonctionnelle, puisque les fonctions sont des valeurs comme les autres, on peut a priori concevoir une autre approche dans laquelle l'application de la fonction ne se fait pas sur tous ses arguments.

Une fonction à plusieurs paramètres peut être vue comme une fonction qui à son premier paramètre associe une fonction qui au second paramètre associe ... une fonction qui au dernier paramètre associe une expression.

```
function x1 -> (function x2 -> (... -> (function xn -> <expr>)...))
```

Ainsi par exemple, on peut définir d'une autre façon la fonction d'addition de deux entiers.

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

L'enchaînement des expressions **function** `x -> ...` pouvant s'avérer fastidieux, le langage permet une autre formulation équivalente à l'aide du mot-clé **fun**.

```
fun x1 x2 ... xn -> <expr>
```

Par exemple,

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

Dans cet exemple, on peut noter que le type d'une fonction à plusieurs arguments se note

`type_arg_1 -> type_arg_2 -> ... -> type_arg_n -> type_resultat`

Un appel à une fonction à plusieurs paramètres se fait en notant successivement le nom de la fonction et ses arguments.

```
# add' 3 (-6) ;;
- : int = -3
```

Les deux fonctions `add` et `add'` sont deux réalisations différentes de la même fonction mathématique. On dit de la seconde qu'elle est déclarée sous forme *curryfiée* et de la première qu'elle l'est sous forme *dcurryfiée*[5].

Le langage CAML autorise l'*application partielle* d'une fonction à plusieurs arguments définie sous forme curryfiée. C'est à dire l'application à un nombre d'arguments inférieur à celui prévu par sa construction. Cette possibilité n'existe pas pour les fonctions définies sous forme décurryfiée.

Lorsque le nombre de paramètres effectifs passés à l'appel d'une fonction est inférieur au nombre de paramètres formels, la valeur du résultat est une fonction

```
# add' 8 ;;
- : int -> int = <fun>
```

Cette fonction est la fonction qui à un entier associe cet entier augmenté de 8. On peut bien évidemment la nommer

```
# let add_8 = add' 8 ;;
val add_8 : int -> int = <fun>
# add_8 11 ;;
- : int = 19
```

Il n'est pas possible d'utiliser le mot-clé **function** sous la forme suivante pour définir une fonction à plusieurs variables.

```
# function x y -> x+y ;;
Characters 11-12:
   function x y -> x+y ;;
              ^
Syntax error
```

Ce mot-clé ne permet que la définition de fonctions unaires.

Si on veut utiliser le mot-clé **function** pour définir `add'`, on écrira

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

On comprend immédiatement l'intérêt du raccourci d'écriture proposé par **fun**.

## 3.10 Déclarations récursives

Sans variables mutables, il n'est pas possible d'effectuer des calculs itératifs avec des boucles **for** et **while**[6].

En programmation fonctionnelle où ne règnent que fonctions et application de fonctions, les calculs itérés ne se réalisent que par l'expression récursive.

Envisageons la définition de la fonction calculant la factorielle de l'entier passé en paramètre. Une expression récursive classique de $n!$ est

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \quad \forall n \in \mathbb{N}^* \end{aligned}$$

On peut être tenté de traduire cette formulation par la déclaration

```
# let fact n =
    if n=0 then 1 else n*fact(n-1) ;;
```

mais on a tort car on obtient le message

```
Characters 37-41:
  if n=0 then 1 else n*fact(n-1) ;;
                       ^^^^
Unbound value fact
```

5. Les qualificatifs « curryfié » et « décurryfié » sont construits à partir du nom du mathématicien américain Haskell Curry dont les travaux ont contribué aux fondements de la programmation logique.

6. Bien entendu, le langage CAML permettant l'expression impérative des programmes dispose de ces structures de contrôle.

qui indique que la variable `fact` apparaissant dans l'expression définissant la fonction `fact` n'est pas liée dans l'environnement de déclaration. On tente de définir `fact` en l'utilisant. On tourne en rond.

Pour remédier à cela, il faut utiliser la forme syntaxique **`let rec`**

```
let rec <nom> = <expr>
```

dans laquelle toute référence à la variable `<nom>` dans l'expression `<expr>` est une référence à la variable en cours de déclaration.

```
# let rec fact = function n ->
  if n = 0 then 1 else n * fact (n - 1) ;;
val fact : int -> int = <fun>
# fact 5 ;;
- : int = 120
```

Une autre façon de définit la fonction `fact` est d'utiliser le filtrage de motifs :

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n - 1)
```