Les bases de la programmation en CAML

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Qu'est ce que CAML ?

CAML est un langage de programmation développé par l'INRIA depuis 1985 ; il se range dans la catégorie des langages fonctionnels, mais se prête aussi à la programmation impérative.

Qu'est ce que CAML ?

CAML est un langage de programmation développé par l'INRIA depuis 1985 ; il se range dans la catégorie des langages fonctionnels, mais se prête aussi à la programmation impérative.

La programmation impérative repose sur la notion de machine à états.

La programmation fonctionnelle met en avant la définition et l'évaluation de fonctions.

Qu'est ce que CAML ?

CAML est un langage de programmation développé par l'INRIA depuis 1985 ; il se range dans la catégorie des langages fonctionnels, mais se prête aussi à la programmation impérative.

La programmation impérative repose sur la notion de machine à états.

La programmation fonctionnelle met en avant la définition et l'évaluation de fonctions.

Exemple : calcul de 5! dans un style impératif.

```
let n = ref (1) in
   for i = 2 to 5 do n := i * !n done ;
   !n ;;
```

Qu'est ce que CAML ?

CAML est un langage de programmation développé par l'INRIA depuis 1985 ; il se range dans la catégorie des langages fonctionnels, mais se prête aussi à la programmation impérative.

La programmation impérative repose sur la notion de machine à états.

La programmation fonctionnelle met en avant la définition et l'évaluation de fonctions.

Exemple : calcul de 5! dans un style impératif.

```
let n = ref (1) in
   for i = 2 to 5 do n := i * !n done ;
   !n ;;
```

Exemple : calcul de 5! dans un style fonctionnel.

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n - 1) in
fact 5 ;;
```

Comment programmer en CAML ?

CAML offre non seulement un compilateur qui transforme des fichiers de code source en code compilé exécutable par la machine, mais aussi un système interactif qui permet de dialoguer avec CAML : en réponse au signe d'invite (le caractère #), l'utilisateur rédige une phrase en CAML qui se termine par l'indication de fin de phrase (les caractères **;;**).

Comment programmer en CAML ?

CAML offre non seulement un compilateur qui transforme des fichiers de code source en code compilé exécutable par la machine, mais aussi un système interactif qui permet de dialoguer avec CAML : en réponse au signe d'invite (le caractère #), l'utilisateur rédige une phrase en CAML qui se termine par l'indication de fin de phrase (les caractères **;;**).

```
>        Caml Light version 0.75

#
```

Comment programmer en CAML ?

CAML offre non seulement un compilateur qui transforme des fichiers de code source en code compilé exécutable par la machine, mais aussi un système interactif qui permet de dialoguer avec CAML : en réponse au signe d'invite (le caractère #), l'utilisateur rédige une phrase en CAML qui se termine par l'indication de fin de phrase (les caractères **;;**).

```
>        Caml Light version 0.75

# let rec fact = function  (* définition d'une fonction *)
    | 0 -> 1
    | n -> n * fact (n-1) ;;
fact : int -> int = <fun>
#
```

Comment programmer en CAML ?

CAML offre non seulement un compilateur qui transforme des fichiers de code source en code compilé exécutable par la machine, mais aussi un système interactif qui permet de dialoguer avec CAML : en réponse au signe d'invite (le caractère #), l'utilisateur rédige une phrase en CAML qui se termine par l'indication de fin de phrase (les caractères **;;**).

```
>        Caml Light version 0.75

# let rec fact = function  (* définition d'une fonction *)
    | 0 -> 1
    | n -> n * fact (n-1) ;;
fact : int -> int = <fun>
# fact 5 ;;                (* application d'une fonction *)
- : int = 120
#
```

Comment programmer en CAML ?

CAML offre non seulement un compilateur qui transforme des fichiers de code source en code compilé exécutable par la machine, mais aussi un système interactif qui permet de dialoguer avec CAML : en réponse au signe d'invite (le caractère #), l'utilisateur rédige une phrase en CAML qui se termine par l'indication de fin de phrase (les caractères **;;**).

```
>        Caml Light version 0.75

# let rec fact = function  (* définition d'une fonction *)
    | 0 -> 1
    | n -> n * fact (n-1) ;;
fact : int -> int = <fun>
# fact 5 ;;                (* application d'une fonction *)
- : int = 120
# fact 2.3 ;;                    (* une erreur de typage *)
Toplevel input:
>fact 2.3 ;;
>     ^^^
This expression has type float, but is used with type int.
#
```

Typage fort, typage faible

Tout objet informatique possède un *type*. Le typage du langage CAML est dit fort car il possède les particularités suivantes :

- le typage d'une fonction est réalisé au moment de sa définition ;
- les conversions implicites de type sont formellement interdites.

Typage fort, typage faible

Tout objet informatique possède un *type*. Le typage du langage CAML est dit fort car il possède les particularités suivantes :

- le typage d'une fonction est réalisé au moment de sa définition ;
- les conversions implicites de type sont formellement interdites.

À l'inverse, le typage du langage PYTHON est faible :

- une même fonction peut s'appliquer à des objets de type différent (par exemple `len`) ;
- la conversion implicite de type est possible : `1.0 + 2 = 3.0`

Typage fort, typage faible

Tout objet informatique possède un *type*. Le typage du langage CAML est dit fort car il possède les particularités suivantes :

- le typage d'une fonction est réalisé au moment de sa définition ;
- les conversions implicites de type sont formellement interdites.

À l'inverse, le typage du langage PYTHON est faible :

- une même fonction peut s'appliquer à des objets de type différent (par exemple `len`) ;
- la conversion implicite de type est possible : `1.0 + 2 = 3.0`

En CAML, il faut réaliser explicitement la conversion :

```
# 1.0 + 2 ;;
Toplevel input:
> 1.0 + 2 ;;
> ^^^
This expression has type float, but is used with type int.
# 1.0 +. float_of_int 2 ;;
- : float = 3.0
```

Typage fort, typage faible

Tout objet informatique possède un *type*. Le typage du langage CAML est dit fort car il possède les particularités suivantes :

- le typage d'une fonction est réalisé au moment de sa définition ;
- les conversions implicites de type sont formellement interdites.

À l'inverse, le typage du langage PYTHON est faible :

- une même fonction peut s'appliquer à des objets de type différent (par exemple `len`) ;
- la conversion implicite de type est possible : `1.0 + 2 = 3.0`

En CAML, il faut réaliser explicitement la conversion :

```
# 1.0 + 2 ;;
Toplevel input:
> 1.0 + 2 ;;
> ^^^
This expression has type float, but is used with type int.
# 1.0 +. float_of_int 2 ;;
- : float = 3.0
```

CAML n'autorise pas non plus la surcharge d'opérateur : l'addition des entiers se note `+`, l'addition des flottants `+.`, etc.

Définitions globales et locales

Attribuer un nom à une valeur à l'aide de l'instruction **`let`** est une *définition*.

```
# let n = 12 ;;
n : int = 12
# let f = function x -> x * x ;;
f : int -> int = <fun>
# f n ;;
- : int = 144
```

Définitions globales et locales

Attribuer un nom à une valeur à l'aide de l'instruction **let** est une *définition*.

```
# let n = 12 ;;
n : int = 12
# let f = function x -> x * x ;;
f : int -> int = <fun>
# f n ;;
- : int = 144
```

La syntaxe «**let ... = ... in**» permet de définir temporairement un nom pour la seule durée du calcul en question.

```
# let n = 11 in f n ;;
- : int = 121
# n ;;
- : int = 12
```

Définitions globales et locales

Attribuer un nom à une valeur à l'aide de l'instruction **let** est une *définition*.

```
# let n = 12 ;;
n : int = 12
# let f = function x -> x * x ;;
f : int -> int = <fun>
# f n ;;
- : int = 144
```

Le mot **and** permet des définitions multiples :

```
# let n = 9 and f = function x -> x * x * x ;;
n : int = 9
f : int -> int = <fun>
```

Définitions globales et locales

Attribuer un nom à une valeur à l'aide de l'instruction **`let`** est une *définition*.

```
# let n = 12 ;;
n : int = 12
# let f = function x -> x * x ;;
f : int -> int = <fun>
# f n ;;
- : int = 144
```

Les valeurs ne deviennent visibles qu'après toutes les déclarations réalisées :

```
# let a = 5 and f = function x -> a * x ;;
Toplevel input:
> let a = 5 and f = function x -> a * x ;;
>                                 ^
The value identifier a is unbound.
```

Définitions globales et locales

Attribuer un nom à une valeur à l'aide de l'instruction **let** est une *définition*.

```
# let n = 12 ;;
n : int = 12
# let f = function x -> x * x ;;
f : int -> int = <fun>
# f n ;;
- : int = 144
```

Les valeurs ne sont visibles qu'après les déclarations réalisées :

```
# let a = 5 and f = function x -> a * x ;;
Toplevel input:
> let a = 5 and f = function x -> a * x ;;
>                                 ^
The value identifier a is unbound.
```

Il faudrait écrire :

```
# let f = let a = 5 in function x -> a * x ;;
f : int -> int = <fun>
```

Fonctions

On définit une fonction par la syntaxe **`let f arg = expr`**.

```
# let f x = x + 1 ;;
f : int -> int = <fun>
# f 2 ;;
- : int = 3
```

$f\ x$ est équivalent à $f(x)$

Fonctions

On définit une fonction par la syntaxe **`let f arg = expr`**.

```
# let f x = x + 1 ;;
f : int -> int = <fun>
# f 2 ;;
- : int = 3
```

$f\ x$ est équivalent à $f(x)$

```
# f 2 * 3 ;;
- : int = 9
# (f 2) * 3 ;;
- : int = 9
# f (2 * 3) ;;
- : int = 7
```

$f\ x\ y$ est équivalent à $(f\ x)\ y$

Fonctions à plusieurs arguments

On utilise la syntaxe **`let f args = expr`**, où **`args`** désigne les arguments, séparés par un espace.

```
# let g x y = x + y ;;
g : int -> int -> int = <fun>
# g 2 3 ;;
- : int = 5
```

Une telle fonction est dite *curryfiée*.

Fonctions à plusieurs arguments

On utilise la syntaxe **`let f args = expr`**, où **`args`** désigne les arguments, séparés par un espace.

```
# let g x y = x + y ;;
g : int -> int -> int = <fun>
# g 2 3 ;;
- : int = 5
```

Une telle fonction est dite *curryfiée*.

La version non curryfiée de cette fonction s'écrit :

```
# let h (x, y) = x + y ;;
h : int * int -> int = <fun>
# h (2, 3) ;;
- : int = 5
```

Fonctions à plusieurs arguments

On utilise la syntaxe **`let f args = expr`**, où **`args`** désigne les arguments, séparés par un espace.

```
# let g x y = x + y ;;
g : int -> int -> int = <fun>
# g 2 3 ;;
- : int = 5
```

Une telle fonction est dite *curryfiée*.

La version non curryfiée de cette fonction s'écrit :

```
# let h (x, y) = x + y ;;
h : int * int -> int = <fun>
# h (2, 3) ;;
- : int = 5
```

Du point de vue mathématique, c'est la différence entre :

$$\left(\begin{array}{ccc} \mathbb{Z} & \longrightarrow & \mathscr{F}(\mathbb{Z},\mathbb{Z}) \\ x & \longmapsto & \left(\begin{array}{ccc} \mathbb{Z} & \longrightarrow & \mathbb{Z} \\ y & \longmapsto & x+y \end{array}\right) \end{array}\right) \quad \text{et} \quad \left(\begin{array}{ccc} \mathbb{Z}^2 & \longrightarrow & \mathbb{Z} \\ (x,y) & \longmapsto & x+y \end{array}\right)$$

Fonctions à plusieurs arguments

On utilise la syntaxe **`let f args = expr`**, où **`args`** désigne les arguments, séparés par un espace.

```
# let g x y = x + y ;;
g : int -> int -> int = <fun>
# g 2 3 ;;
- : int = 5
```

Une telle fonction est dite *curryfiée*.

On aurait pu définir la fonction *f* en écrivant : **`let f x = g 1 x`**, ou encore **`let f x = (g 1) x`**. Le langage CAML autorisant la simplification à droite, il nous est alors loisible de définir *f* de la façon suivante :

```
# let f = g 1 ;;
f : int -> int = <fun>
```

Cette simplification n'est pas possible avec la fonction non curryfiée.

Fonctions anonymes

On peut construire une valeur fonctionnelle anonyme en suivant la syntaxe **`function x -> expr`** :

```
# let f = function x -> x + 1 ;;
f : int -> int = <fun>
# let g = function x -> function y -> x + y ;;
g : int -> int -> int = <fun>
# let h = function (x, y) -> x + y ;;
h : int * int -> int = <fun>
```

Fonctions anonymes

On peut construire une valeur fonctionnelle anonyme en suivant la syntaxe **`function x -> expr`** :

```
# let f = function x -> x + 1 ;;
f : int -> int = <fun>
# let g = function x -> function y -> x + y ;;
g : int -> int -> int = <fun>
# let h = function (x, y) -> x + y ;;
h : int * int -> int = <fun>
```

`g 1` est donc équivalent à : **`function y -> 1 + y`**.

Fonctions anonymes

On peut construire une valeur fonctionnelle anonyme en suivant la syntaxe **`function x -> expr`** :

```
# let f = function x -> x + 1 ;;
f : int -> int = <fun>
# let g = function x -> function y -> x + y ;;
g : int -> int -> int = <fun>
# let h = function (x, y) -> x + y ;;
h : int * int -> int = <fun>
```

`g 1` est donc équivalent à : **`function y -> 1 + y`**.

Le typage est associatif à droite :

int -> int -> int est équivalent à *int -> (int -> int)*

Fonctions anonymes

On peut construire une valeur fonctionnelle anonyme en suivant la syntaxe **`function x -> expr`** :

```
# let f = function x -> x + 1 ;;
f : int -> int = <fun>
# let g = function x -> function y -> x + y ;;
g : int -> int -> int = <fun>
# let h = function (x, y) -> x + y ;;
h : int * int -> int = <fun>
```

`g 1` est donc équivalent à : **`function y -> 1 + y`**.

Le typage est associatif à droite :

int -> int -> int est équivalent à *int -> (int -> int)*

`function` ne définit qu'une fonction à une variable. On utilise **`fun`** pour définir une fonction anonyme à plusieurs variables :

`fun a b ...` est équivalent à **`function a -> function b -> ...`**

On peut donc aussi écrire : **`let g = fun x y -> x + y`**.

Le vide

Le type *unit* ne comporte qu'une seule valeur, notée ( ).

Un langage fonctionnel tel que CAML ne fait que définir et appliquer des fonctions. Voilà pourquoi les fonctions à effet de bord sont en général de type *... -> unit* :

```
# print_string "Hello World" ;;
Hello World- : unit = ()
```

Il existe de même **print_int** de type *int -> unit*, **print_float**, de type *float -> unit* et **print_char**, de type *char -> unit*.

Le vide

Le type *unit* ne comporte qu'une seule valeur, notée ( ).

Un langage fonctionnel tel que CAML ne fait que définir et appliquer des fonctions. Voilà pourquoi les fonctions à effet de bord sont en général de type *... -> unit* :

```
# print_string "Hello World" ;;
Hello World- : unit = ()
```

Il existe de même **print_int** de type *int -> unit*, **print_float**, de type *float -> unit* et **print_char**, de type *char -> unit*.

Autre exemple, la fonction **print_newline** est de type *unit -> unit* et a pour effet de passer à la ligne. Il ne faut donc pas oublier son argument :

```
# print_newline () ;;

- : unit = ()
```

Le vide

Le type *unit* ne comporte qu'une seule valeur, notée ( ).

Un langage fonctionnel tel que CAML ne fait que définir et appliquer des fonctions. Voilà pourquoi les fonctions à effet de bord sont en général de type *... -> unit* :

```
# print_string "Hello World" ;;
Hello World- : unit = ()
```

Il existe de même **print_int** de type *int -> unit*, **print_float**, de type *float -> unit* et **print_char**, de type *char -> unit*.

Autre exemple, la fonction **print_newline** est de type *unit -> unit* et a pour effet de passer à la ligne. Il ne faut donc pas oublier son argument :

```
# print_newline () ;;

- : unit = ()
# print_newline ;;
- : unit -> unit = <fun>
```

Les entiers

Les éléments de type *int* sont les entiers de l'intervalle $[\![-2^{62}, 2^{62}-1]\!]$, les calculs s'effectuant modulo 2^{63} par complémentation à deux :

```
# max_int ;;
- : int = 4611686018427387903
# min_int ;;
- : int = -4611686018427387904
# max_int + 1 ;;
- : int = -4611686018427387904
```

Les entiers

Les éléments de type *int* sont les entiers de l'intervalle $[\![-2^{62}, 2^{62}-1]\!]$, les calculs s'effectuant modulo 2^{63} par complémentation à deux :

```
# max_int ;;
- : int = 4611686018427387903
# min_int ;;
- : int = -4611686018427387904
# max_int + 1 ;;
- : int = -4611686018427387904
```

Opérations usuelles : **+**, **−**, *****, **/**, **mod**.

```
# 1 + 3 mod 2 ;;        (*  1 + (3 mod 2)  *)
- : int = 2
# 2 * 5 / 3 ;;          (*  (2 * 5) / 3    *)
- : int = 3
# 5 / 3 * 2 ;;          (*  (5 / 3) * 2    *)
- : int = 2
```

Les réels

Les éléments de type *float* sont composés d'un signe, une mantisse et un exposant :

```
# let pi = 3.141592653 and date = 1.789e3 ;;
pi : float = 3.141592653
date : float = 1789.0
```

Les réels

Les éléments de type *float* sont composés d'un signe, une mantisse et un exposant :

```
# let pi = 3.141592653 and date = 1.789e3 ;;
pi : float = 3.141592653
date : float = 1789.0
```

Opérations usuelles : **+.**, **-.**, ***.**, **/.**, ******, **sqrt**, **exp**, **log**, **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, ...

```
# let a = 3.141592654 in tan (a /. 4.) ;;
- : float = 1.00000000021
```

Les réels

Les éléments de type *float* sont composés d'un signe, une mantisse et un exposant :

```
# let pi = 3.141592653 and date = 1.789e3 ;;
pi : float = 3.141592653
date : float = 1789.0
```

Opérations usuelles : **+.**, **-.**, ***.**, **/.**, ******, **sqrt**, **exp**, **log**, **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, ...

```
# let a = 3.141592654 in tan (a /. 4.) ;;
- : float = 1.00000000021
```

Attention aux erreurs de typage :

```
# let a = 3.141592653 in tan (a /. 4) ;;
Toplevel input:
>let a = 3.141592653 in tan (a /. 4) ;;
>                                    ^
This expression has type int, but is used with type float.
```

Caractères et chaînes de caractères

Les caractères sont les éléments du type *char* ; les chaînes de caractères sont les éléments de type *string* :

```
# string_of_char 'a' ;;
- : string = "a"
# let s = "Caml" in s.[1] ;;
- : char = 'a'
# "liberté, " ^ "égalité, " ^ "fraternité" ;;
- : string = "liberté, égalité, fraternité"
```

Caractères et chaînes de caractères

Les caractères sont les éléments du type *char* ; les chaînes de caractères sont les éléments de type *string* :

```
# string_of_char 'a' ;;
- : string = "a"
# let s = "Caml" in s.[1] ;;
- : char = 'a'
# "liberté, " ^ "égalité, " ^ "fraternité" ;;
- : string = "liberté, égalité, fraternité"
```

Attention : contrairement à PYTHON les éléments de type *string* sont des données mutables ; on peut en modifier leur valeur :

```
# let mot = "Caml" ;;
mot : string = "Caml"
# mot.[3] <- 'p' ;;
- : unit = ()
# mot ;;
- : string = "Camp"
```

Il s'agit d'une structure analogue aux tableaux (ou vecteurs).

Les booléens

Le type *bool* comporte deux valeurs : **true** et **false**, et les opérateurs logiques : **not**, **&&**, **||**.
Les deux derniers opérateurs fonctionnent suivant le principe de l'évaluation paresseuse :

```
# not (1 = 2) || (1 / 0 = 1) ;;
- : bool = true
# not (1 = 2) && (1 / 0 = 1) ;;
Uncaught exception: Division_by_zero
```

Les booléens

Le type *bool* comporte deux valeurs : **true** et **false**, et les opérateurs logiques : **not**, **&&**, **||**.
Les deux derniers opérateurs fonctionnent suivant le principe de l'évaluation paresseuse :

```
# not (1 = 2) || (1 / 0 = 1) ;;
- : bool = true
# not (1 = 2) && (1 / 0 = 1) ;;
Uncaught exception: Division_by_zero
```

Attention, la préséance du typage reste effective :

```
# not (1 = 2) || (1 / 0 = 1.) ;;
Toplevel input:
>  not (1 = 2) && (1 / 0 = 1.) ;;
>                              ^^
This expression has type float, but is used with type int.
```

Les paires

Ou plus généralement les *n*-uples, sont séparés par une virgule : **x,y,z** est de type *type_de_x * type_de_y * type_de_z*

```
# (1, 2, 3) ;;                (* ici les parenthèses sont superflues *)
- : int * int * int = 1, 2, 3
# (1, 2), 3 ;;
- : (int * int) * int = (1, 2), 3
# 1, (2, 3) ;;
- : int * (int * int) = 1, (2, 3)
```

Les parenthèses servent à hierarchiser les liaisons.

Les paires

Ou plus généralement les *n*-uples, sont séparés par une virgule : **x,y,z** est de type *type_de_x * type_de_y * type_de_z*

```
# (1, 2, 3) ;;                (* ici les parenthèses sont superflues *)
- : int * int * int = 1, 2, 3
# (1, 2), 3 ;;
- : (int * int) * int = (1, 2), 3
# 1, (2, 3) ;;
- : int * (int * int) = 1, (2, 3)
```

Les parenthèses servent à hierarchiser les liaisons.

```
# fst (1, 2), 3 ;;
- : int * int = 1, 3
# fst 1, (2, 3) ;;
Toplevel input:
>fst 1, (2, 3) ;;
>    ^
This expression has type int, but is used with type 'a * 'b.
```

Il est préférable d'écrire **(x, y, z)** plutôt que **x, y, z**.

Polymorphisme

Une fonction polymorphe s'applique indifféremment à tous les types ; *'a 'b 'c* ... désignent alors des types quelconques.

```
# let fst (x, y) = x ;;
fst : 'a * 'b -> 'a = <fun>
# let snd (x, y) = y ;;
snd : 'a * 'b -> 'b = <fun>
```

Polymorphisme

Une fonction polymorphe s'applique indifféremment à tous les types ; *'a 'b 'c* ... désignent alors des types quelconques.

```
# let fst (x, y) = x ;;
fst : 'a * 'b -> 'a = <fun>
# let snd (x, y) = y ;;
snd : 'a * 'b -> 'b = <fun>
```

Utilisées dans un certain contexte, ces fonctions peuvent perdre leur caractère polymorphe :

```
# let f x y = 1 + fst (x, y) ;;
f : int -> 'a -> int = <fun>
```

Polymorphisme

Une fonction polymorphe s'applique indifféremment à tous les types ; *'a 'b 'c* ... désignent alors des types quelconques.

```
# let fst (x, y) = x ;;
fst : 'a * 'b -> 'a = <fun>
# let snd (x, y) = y ;;
snd : 'a * 'b -> 'b = <fun>
```

Utilisées dans un certain contexte, ces fonctions peuvent perdre leur caractère polymorphe :

```
# let f x y = 1 + fst (x, y) ;;
f : int -> 'a -> int = <fun>
```

Les opérations de comparaison =, <>, <, >, <=, >= permettent la comparaison (entre autre) des éléments de type *int*, *float*, *char*, *string*.

```
# 1.0 > 2.0 ;;
- : bool = false
# 'a' < 'b' ;;
- : bool = true
```

Types construits

Types sommes

Les types sommes modélisent des données comportant des alternatives.

```
# type reel_etendu = Reel of float | Plus_infini | Moins_infini ;;
Type reel_etendu defined.
# Plus_infini ;;
- : reel_etendu = Plus_infini
# Reel 3.14 ;;
- : reel_etendu = Reel 3.14
# let etendu_of_float x = Reel x ;;
etendu_of_float : float -> reel_etendu = <fun>
# etendu_of_float 3.14 ;;
- : reel_etendu = Reel 3.14
```

Types construits

Types sommes

Les types sommes modélisent des données comportant des alternatives.

```
# type reel_etendu = Reel of float | Plus_infini | Moins_infini ;;
Type reel_etendu defined.
# Plus_infini ;;
- : reel_etendu = Plus_infini
# Reel 3.14 ;;
- : reel_etendu = Reel 3.14
# let etendu_of_float x = Reel x ;;
etendu_of_float : float -> reel_etendu = <fun>
# etendu_of_float 3.14 ;;
- : reel_etendu = Reel 3.14
```

Les fonctions définies sur un type somme agissent en général par filtrage :

```
# let oppose = function
    | Moins_infini -> Plus_infini
    | Reel x       -> Reel (-.x)
    | Plus_infini  -> Moins_infini ;;
oppose : reel_etendu -> reel_etendu = <fun>
```

Types construits

Types produits

Les types produits (ou enregistrement) modélisent des données définies par plusieurs caractéristiques.

```
# type complexe = {Re : float ; Im : float} ;;
Type Complexe defined.
# let i = {Re = 0. ; Im = 1.} ;;
i : complexe = {Re=0.0; Im=1.0}
# let complexe_of_float x = {Re = x ; Im = 0.} ;;
complexe_of_float : float -> complexe = <fun>
```

Types construits

Types produits

Les types produits (ou enregistrement) modélisent des données définies par plusieurs caractéristiques.

```
# type complexe = {Re : float ; Im : float} ;;
Type Complexe defined.
# let i = {Re = 0. ; Im = 1.} ;;
i : complexe = {Re=0.0; Im=1.0}
# let complexe_of_float x = {Re = x ; Im = 0.} ;;
complexe_of_float : float -> complexe = <fun>
```

Pour accéder à l'une des caractéristiques d'un objet de type produit, il suffit de faire suivre le nom d'un point et du nom de la caractéristique.

```
# let mult x y =
    let a = x.Re and b = x.Im and c = y.Re and d = y.Im in
      {Re = a*.c-.b*.d ; Im = a*.d+.b*.c} ;;
mult : complexe -> complexe -> complexe = <fun>
# mult i i ;;
- : complexe = {Re=-1.0; Im=0.0}
```

Types construits

La définition d'un type peut être récursive :

```
# type couleur = Cyan | Magenta | Jaune
                 | Melange of couleur * couleur ;;
Type couleur defined.
# let Rouge = Melange (Magenta, Jaune) ;;
Rouge : couleur = Melange (Magenta, Jaune)
# let Orange = Melange (Rouge, Jaune) ;;
Orange : couleur = Melange (Melange (Magenta, Jaune), Jaune)
```

Types construits

La définition d'un type peut être récursive :

```
# type couleur = Cyan | Magenta | Jaune
                 | Melange of couleur * couleur ;;
Type couleur defined.
# let Rouge = Melange (Magenta, Jaune) ;;
Rouge : couleur = Melange (Magenta, Jaune)
# let Orange = Melange (Rouge, Jaune) ;;
Orange : couleur = Melange (Melange (Magenta, Jaune), Jaune)
```

Pour calculer la composante CMJ d'une couleur :

```
# let rec cmj = function
    | Cyan                    -> (1., 0., 0.)
    | Magenta                 -> (0., 1., 0.)
    | Jaune                   -> (0., 0., 1.)
    | Melange (coul1, coul2) -> let (a, b, c) = cmj coul1
                                  and (d, e, f) = cmj coul2 in
                                ((a+.d)/.2., (b+.e)/.2., (c+.f)/.2.) ;;
cmj : couleur -> float * float * float = <fun>
# cmj Orange ;;
- : float * float * float = 0.0, 0.25, 0.75
```

Filtrage par motif (pattern matching)

La syntaxe d'un filtrage par motif est la suivante :

```
match expr0 with
  | motif1 -> expr1
  | motif2 -> expr2
  | ..............
  | motifn -> exprn
```

L'expression **expr0** est évaluée, et sa valeur comparée au premier motif :

- si cette valeur répond aux contraintes du motif, c'est **expr1** qui sera évaluée en réponse au filtrage ;
- sinon, sa valeur est comparée au second motif, et ainsi de suite.

Si aucun motif n'est reconnu, une exception *Match_failure* est levée.

Filtrage par motif (pattern matching)

La syntaxe d'un filtrage par motif est la suivante :

```
match expr0 with
  | motif1 -> expr1
  | motif2 -> expr2
  | ..............
  | motifn -> exprn
```

L'expression **expr0** est évaluée, et sa valeur comparée au premier motif :

- si cette valeur répond aux contraintes du motif, c'est **expr1** qui sera évaluée en réponse au filtrage ;
- sinon, sa valeur est comparée au second motif, et ainsi de suite.

Si aucun motif n'est reconnu, une exception *Match_failure* est levée.

```
# let sinc = function
    | 0. -> 1.
    | x  -> sin (x) /. x ;;
sinc : float -> float = <fun>
```

Un nom (ici **x**) s'accorde avec n'importe quelle valeur.

Filtrage par motif (pattern matching)

La syntaxe d'un filtrage par motif est la suivante :

```
match expr0 with
  | motif1 -> expr1
  | motif2 -> expr2
  | ..............
  | motifn -> exprn
```

L'expression **expr0** est évaluée, et sa valeur comparée au premier motif :

- si cette valeur répond aux contraintes du motif, c'est **expr1** qui sera évaluée en réponse au filtrage ;
- sinon, sa valeur est comparée au second motif, et ainsi de suite.

Si aucun motif n'est reconnu, une exception *Match_failure* est levée.

```
# let sinc x = match x with
    | 0. -> 1.
    | _  -> sin (x) /. x ;;
sinc : float -> float = <fun>
```

Le caractère _ s'accorde avec n'importe quelle valeur sans la nommer.

Filtrage par motif (pattern matching)

La syntaxe d'un filtrage par motif est la suivante :

```
match expr0 with
  | motif1 -> expr1
  | motif2 -> expr2
  | ..............
  | motifn -> exprn
```

L'ordre dans lequel on essaye de faire correspondre un motif et une valeur a de l'importance :

```
# let sinc = function
    | x -> sin(x) /. x
    | 0. -> 1. ;;
Toplevel input:
>   | 0. -> 1. ;;
>     ^^
Warning: this matching case is unused.
sinc : float -> float = <fun>

# sinc 0. ;;
- : float = nan.0
```

Motif gardé

Ne pas confondre concordance avec un motif et égalité de valeur.

On peut utiliser une constante dans un motif :

```
# let est_nul = function
    | 0 -> true
    | _ -> false ;;
est_nul : int -> bool = <fun>
```

Motif gardé

Ne pas confondre concordance avec un motif et égalité de valeur.

On peut utiliser une constante dans un motif :

```
# let est_nul = function
    | 0 -> true
    | _ -> false ;;
est_nul : int -> bool = <fun>
```

On ne peut utiliser une valeur calculée dans un motif :

```
# let egal x = function
    | x -> true
    | _ -> false ;;
Toplevel input:
>   | _ -> false ;;
>     ^
Warning: this matching case is unused.
egal : 'a -> 'b -> bool = <fun>
```

Motif gardé

Ne pas confondre concordance avec un motif et égalité de valeur.

On peut utiliser une constante dans un motif :

```
# let est_nul = function
    | 0 -> true
    | _ -> false ;;
est_nul : int -> bool = <fun>
```

On ne peut utiliser une valeur calculée dans un motif :

```
# let egal x = function
    | x -> true
    | _ -> false ;;
Toplevel input:
>   | _ -> false ;;
>     ^
Warning: this matching case is unused.
egal : 'a -> 'b -> bool = <fun>

# egal 1 "artichaut" ;;
- : bool = true
```

Motif gardé

Ne pas confondre concordance avec un motif et égalité de valeur.

On peut utiliser une constante dans un motif :

```
# let est_nul = function
    | 0 -> true
    | _ -> false ;;
est_nul : int -> bool = <fun>
```

On utilise un motif gardé pour lier la reconnaissance d'un motif à une condition :

```
# let egal x = function
    | y when x = y -> true
    | _            -> false ;;
egal : 'a -> 'a -> bool = <fun>
```

Filtrage non exhaustif

L'interprète de commande repère un filtrage incomplet.

```
# let addition = fun
    | (Reel x) (Reel y)       -> Reel (x +. y)
    | Moins_infini (Reel x)   -> Moins_infini
    | Plus_infini (Reel x)    -> Plus_infini
    | (Reel x) Moins_infini   -> Moins_infini
    | (Reel x) Plus_infini    -> Plus_infini ;;
Warning: this matching is not exhaustive.
addition : reel_etendu -> reel_etendu -> reel_etendu = <fun>

# addition Plus_infini Plus_infini ;;
 Exception non rattrapée: Match_failure ("", 294, 534)
```

Filtrage non exhaustif

L'interprète de commande repère un filtrage incomplet.

```
# let addition = fun
    | (Reel x) (Reel y)       -> Reel (x +. y)
    | Moins_infini (Reel x)   -> Moins_infini
    | Plus_infini (Reel x)    -> Plus_infini
    | (Reel x) Moins_infini   -> Moins_infini
    | (Reel x) Plus_infini    -> Plus_infini
    | a a                     -> a
    | _ _                     -> failwith "opération non définie" ;;
Toplevel input:
>     | a a                      -> a
>           ^
The variable a is bound several times in this pattern.
```

Un nom ne peut apparaître plusieurs fois dans un même motif.

Filtrage non exhaustif

L'interprète de commande repère un filtrage incomplet.

```
# let addition = fun
    | (Reel x) (Reel y)       -> Reel (x +. y)
    | Moins_infini (Reel x)   -> Moins_infini
    | Plus_infini (Reel x)    -> Plus_infini
    | (Reel x) Moins_infini   -> Moins_infini
    | (Reel x) Plus_infini    -> Plus_infini
    | a b when a = b          -> a
    | _ _                     -> failwith "opération non définie" ;;
addition : reel_etendu -> reel_etendu -> reel_etendu = <fun>
```

L'utilisation d'un motif gardé permet de contourner le problème.

Filtrage non exhaustif

L'interprète de commande repère un filtrage incomplet.

Notons cependant que la vérification de l'exhaustivité d'un filtrage suppose que l'expression conditionnelle attachée à un motif gardé est fausse :

```
# let f = function
    | x when x = 0  -> 1
    | x when x <> 0 -> 0 ;;
Toplevel input:
Warning: this matching is not exhaustive.
f : int -> int = <fun>
```

Filtrage non exhaustif

L'interprète de commande repère un filtrage incomplet.

L'usage recommande de terminer par un motif générique non gardé :

```
# let f = function
    | x when x >= 0 -> x + 1
    | x             -> x - 1 ;;
f : int -> int = <fun>
```

Récursivité

Le mot clé **rec** indique la définition d'un objet récursif, c'est-à-dire un objet dont le nom intervient dans sa définition.

Récursivité

Le mot clé **rec** indique la définition d'un objet récursif, c'est-à-dire un objet dont le nom intervient dans sa définition.

Deux exemples de fonctions récursives :

```
# let rec fact = function
    | 0 -> 1
    | n -> n * fact (n-1) ;;
fact : int -> int = <fun>
```

```
# let rec maccarthy = function
    | n when n > 100 -> n - 10
    | n              -> maccarthy (maccarthy (n + 11)) ;;
maccarthy : int -> int = <fun>
```

Nous reviendrons plus tard sur le support théorique de la récursivité en informatique.

Récursivité

Le mot clé **rec** indique la définition d'un objet récursif, c'est-à-dire un objet dont le nom intervient dans sa définition.

Quand deux fonctions interviennent chacune dans la définition de l'autre, on parle de récursivité croisée :

```
# let rec est_pair = function
    | 0 -> true
    | n -> est_impair (n-1)
  and est_impair = function
    | 0 -> false
    | n -> est_pair (n-1) ;;
est_pair : int -> bool = <fun>
est_impair : int -> bool = <fun>
```

Attention, les deux fonctions doivent impérativement être définies dans une même phrase.