Introduction au langage OCaml

Option Informatique

Anthony Lick
Lycée Janson de Sailly

Ocaml : un langage fonctionnel

Introduction au langage OCaml

OCaml

Le langage que l'on va utilisé en Option Informatique est **OCaml** (qui vient de "Objective Caml", car s'est une extension du langage **Caml** permettant de faire de la **programmation orientée objet**).

Il a été créé en 1996 par l'équipe de **Xavier Leroy**, chercheur INRIA à l'université de Paris-Diderot.

Introduction au langage OCaml

OCaml vs Python

On va commencer par voir les bases de la programmation en **OCaml**, en le comparant à **Python** (utilisé en Informatique Commune).

OCaml : un langage fonctionnel

Styles de programmation

On distingue deux principaux styles de programmation :

- Les **langages impératifs**.
  $\hookrightarrow$ ex : C, C++, Java, **Python**, ...
- Les les **langages fonctionnels**.
  $\hookrightarrow$ ex : Haskell, Scala, **OCaml**, ...

Programmation impérative vs Programmation fonctionnelle

Programmation impérative

La **programmation impérative** est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.

Un **état** représente l'ensemble des variables d'un programme.

L'**exécution** d'un programme consiste, à partir d'un état initial, à exécuter une séquence finie de commandes d'affectation modifiant l'état courant.

Les boucles (`for` et `while`) sont à disposition pour permettre la répétition d'instructions et les structures conditionnelles (`if`, `then`, `else`) pour permettre l'exécution conditionnelle d'instructions.

Programmation impérative vs Programmation fonctionnelle

Programmation impérative

La quasi-totalité des processeurs qui équipent les ordinateurs sont de nature **impérative** : ils sont faits pour exécuter du code écrit sous forme d'**opcodes** (pour operation codes), qui sont des instructions élémentaires exécutables par le processeur.

L'ensemble des opcodes disponibles forme le **langage machine** spécifique au processeur et à son architecture.

Programmation impérative vs Programmation fonctionnelle

Programmation impérative

L'état du programme à un instant donné est défini par le contenu de la mémoire centrale à cet instant, et le programme lui-même est écrit en style impératif en langage machine, ou le plus souvent dans une traduction lisible par les humains du langage machine, dénommée **assembleur**.

Les **langages impératifs** suivent cette nature, tout en permettant des opérations plus complexes (il n'y a pas de boucles en assembleur) : c'est pour cela qu'ils sont les plus répandus.

Programmation impérative vs Programmation fonctionnelle

Programmation fonctionnelle

La **programmation fonctionnelle** est un paradigme de programmation qui considère le calcul comme l'**évaluation** de **fonctions mathématiques**.

Ainsi, un programme est une fonction au sens mathématique, l'exécution d'un programme étant l'évaluation d'une fonction.

Le programmeur écrivant un programme dans le paradigme fonctionnel va écrire des fonctions, simples à la base, puis de plus en plus complexes en les composant : une fonction déjà écrite sert de "**boite noire**" dans une autre.

Programmation impérative vs Programmation fonctionnelle

Programmation fonctionnelle

Dans un langage fonctionnel, il n'y a pas de différence de nature entre une **fonction** et un **objet simple** (un entier ou un flottant, par exemple). Une fonction est une **expression** comme une autre et à ce titre pourra elle-même être l'argument ou le résultat d'une autre fonction.

En programmation fonctionnelle, on ne dispose pas d'instructions permettant de modifier l'état du programme, il n'y a donc pas de **boucles** (qui changeraient l'état) : celles-ci sont remplacées par l'usage de la **récursivité** (capacité d'une fonction à s'appeler elle même).

Programmation impérative vs Programmation fonctionnelle

Programmation fonctionnelle

En pratique, les **langages fonctionnels** sont écrits dans un langage de plus **bas niveau** (plus proche du langage machine), et masquent à l'utilisateur la nature impérative du processeur qui va exécuter le programme.

OCaml est un langage fonctionnel, écrit en **C** (comme **Python** d'ailleurs) qui est un langage impératif de bas niveau.

Exemple

Exemple

Il y a deux manières de calculer la factorielle : avec un **produit** ou par **récurrence**.

$$n! = \prod_{i=1}^{n} i \qquad n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Impératif vs Fonctionnel

- La première formule correspond à une vision **impérative** : on peut calculer ce produit avec une **boucle for**.
- La deuxième correspond à une vision **fonctionnelle** : on peut définir une fonction qui va s'**appeler elle-même**, et calculer son résultat **récursivement**.

Exemple

Exemple

En **Python**, il sera plus naturel d'implémenter la factorielle de manière **impérative**, mais on peut aussi définir des fonctions **récursives**.

$$n! = \prod_{i=1}^{n} i \qquad n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

version impérative

```
1  def fact(n):
2      f = 1
3      for i in range(1,n+1):
4          f = f*i
5      return f
```

version récursive

```
1  def fact_rec(n):
2      if n==0:
3          return 1
4      else:
5          return n*fact_rec(n-1)
```

Python vs OCaml

Python vs OCaml

- **Python** est un langage **impératif** permettant l'usage de la **récursivité**.
- **OCaml** est un langage **fonctionnel** permettant l'usage de la programmation **impérative**.

$\hookrightarrow$ Le style naturel pour implémenter la factorielle est plutôt le premier pour **Python**, et le deuxième pour **OCaml**, mais il est possible de faire l'inverse.

Un IDE pour OCaml : OCaml-Top

Programmer avec OCaml

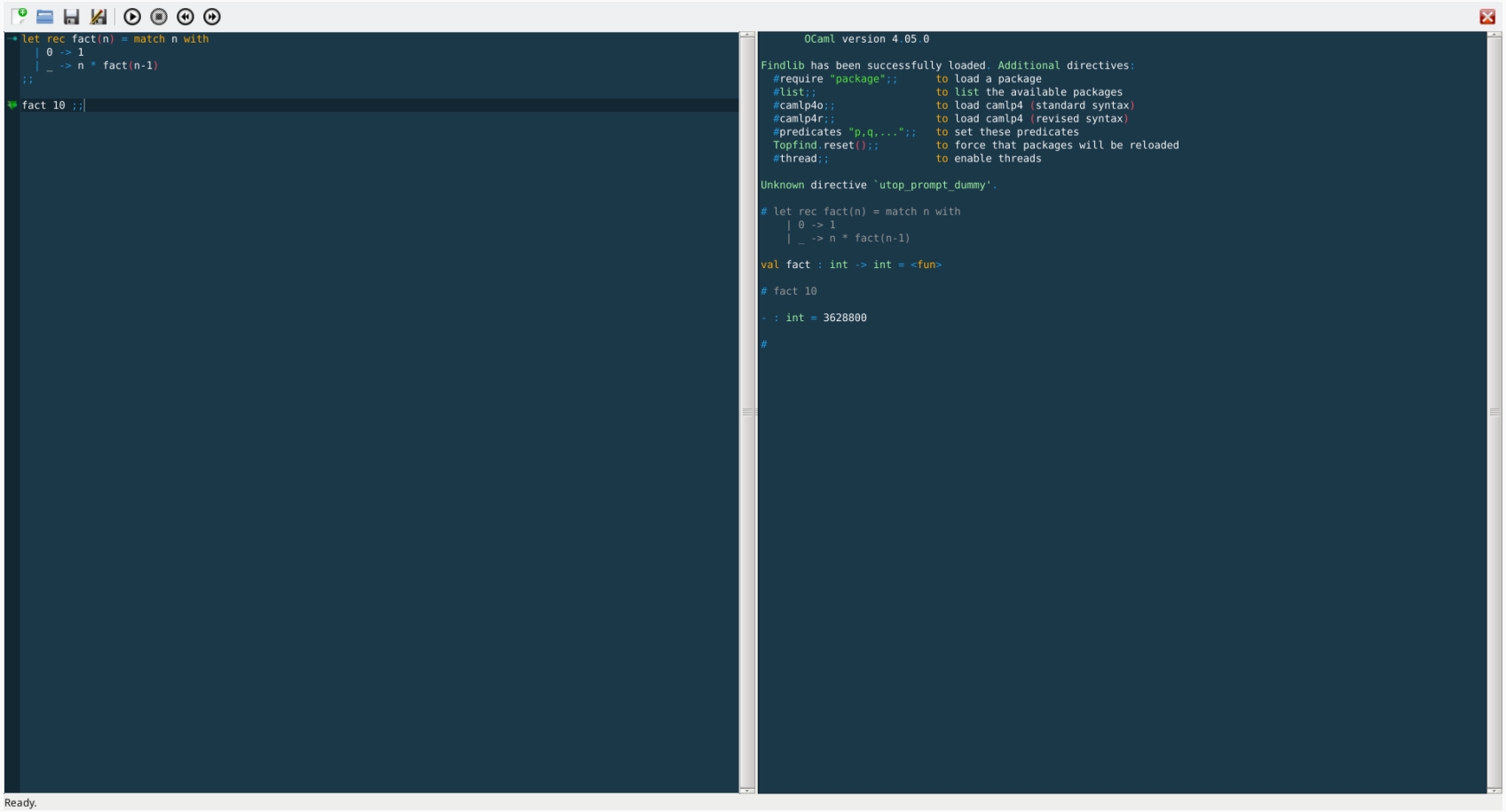
Comme avec **Python**, pour programmer en **OCaml** il faut :

- avoir **OCaml** installé sur sa machine ;
- utiliser un **éditeur** pour écrire nos programmes.

Sur les ordinateurs de Janson, on travaillera avec `ocaml-top`, qui à l'avantage d'être disponible sous **Windows**, **Linux**, et **OSX**.

Vous pouvez bien sûr utiliser un autre éditeur si vous préférez.

Un IDE pour OCaml : OCaml-Top



Un langage fortement typé

Un langage fortement typé

```
1  # 4+1 ;;
2  - : int = 5
```

Exemple

Commençons par le calcul simple ci-dessus.

- Le **prompt** (#) au début de la première ligne est l'**invite** de l'**interpréteur**.
- Le reste de cette ligne a été écrit au clavier.
- La deuxième ligne est le résultat du calcul.

Un langage fortement typé

```
1  # 4+1 ;;
2  - : int = 5
```

Premières remarques

Comme on le voit sur cet exemple :

- un calcul en **Caml** termine par deux points-virgules `;;`
- **Caml** procède (avant même l'évaluation) par une **analyse de types** : il sait avant même de le calculer que le résultat de l'opération est un entier (`int`).

Un langage fortement typé

Langage fortement typé

Caml est un langage **fortement typé** : toute valeur possède un type, les opérateurs et fonctions prennent en paramètres et renvoient en sortie des données d'un certain type.

Le mélange des genres est interdit : on ne peut pas, comme en **Python**, définir une fonction qui renvoie dans certains cas un entier et dans d'autres un booléen, ou considérer des listes d'éléments inhomogènes.

Caml permet cependant la généricité (**polymorphisme**) : par exemple, la fonction `max` et l'opérateur <= peuvent comparer deux entiers ou deux flottants, et une fonction de tri permettra de trier une liste d'entiers comme une liste de flottants.

Types simples

Le type “rien”

unit

Le type `unit` en **Caml** correspond au type `None` en **Python**.

Il n'y a qu'une constante égale à ce type-là, dont l'utilité apparaîtra plus tard.

On peut la construire à l'aide de deux parenthèses.

```
1  # () ;;
2  - : unit = ()
```

Les entiers

int

Le type `int` correspond aux entiers.

Sur une implémentation 64 bits, ceux-ci sont restreints à la plage de valeurs $[\![-2^{62}, 2^{63} - 1]\!]$.

Les opérateurs sur les entiers sont `+`, `*`, `-`, `/` (division entière), et `mod` (modulo).

```
# 2*16 ;;
- : int = 32
# 58 mod 14 ;;
- : int = 2
```

Les flottants

float

Le type `float` correspond aux flottants.

Ce sont essentiellement les mêmes qu'en **Python**.

Les opérations sont les mêmes que pour les entiers (sauf `mod` qui n'a pas d'équivalent), mais suivis d'un point qui les différencie de leurs équivalents sur les entiers.

Les fonctions usuelles (**`cos`**, **`sin`**, **`exp`**, **`atan`**, ...) sont définies sur les flottants, ainsi que l'opérateur d'exponentiation (**`**`**).

```
1  # 2.0 *. 8.9 ;;
2  - : float = 17.8
3  # 2.0 ** 8.9 ;;
4  - : float = 477.712891666845508
5  # cos 5. ;;
6  - : float = 0.283662185463226246
```

Les flottants

```
# 2.0 * 3.0 ;;
Characters 0-3:
  2.0 * 3.0 ;;
  ^^^
Error: This expression has type float but an expression was expected of type int
# 2 *. 3.0 ;;
Characters 0-1:
  2 *. 3.0 ;;
  ^
Error: This expression has type int but an expression was expected of type float
# 2 ** 5 ;;
Characters 0-1:
  2 ** 5 ;;
  ^
Error: This expression has type int but an expression was expected of type float
```

Attention à respecter les types

Dans le premier cas ci-dessus, * est défini sur les entiers, mais `2.0` est un flottant.

Dans les deux autres cas c'est l'inverse : `*.` et `**` sont définis sur les flottants.

Les booléens

bool

Semblables aux booléens de **Python**, le type `bool` de **Caml** n'a que deux constantes : `true` et `false` (sans majuscules).

Les opérateurs logiques sont `&&` (**et** logique), `||` (**ou** logique), et `not` (**non** logique).

En terme de priorité, `not` est prioritaire sur `&&` qui l'est sur `||`.

```
1  # true && false ;;
2  - : bool = false
3  # false || true ;;
4  - : bool = true
5  # not true ;;
6  - : bool = false
```

Les booléens

Inégalités

Pour tester des inégalités, la syntaxe est la même qu'en **Python** : <, >, <=, et >=.

Attention, en **Caml** il faut que les deux valeurs que l'on compare soit du **même type**.

```
# 1 < 2 ;;
- : bool = true
# 5.0 >= 12.0 ;;
- : bool = false
# 2.5 < 1 ;;
Characters 6-7:
  2.5 < 1 ;;
        ^
Error: This expression has type int but an expression was expected of type float
```

Les booléens

Égalités

- Tester la différence se fait avec le symbole <>.
- Tester l'égalité se fait avec un seul symbole =.

```
# 1 = 1 ;;
- : bool = true
# 1 <> 1 ;;
- : bool = false
```

Attention

En **Caml**, les opérateurs **==** et **!=** existent, mais testent l'égalité **physique** (c'est-à-dire si les deux valeurs sont stockées au même endroit dans la mémoire).

Attention à ne pas confondre avec la syntaxe Python !

```
# 2.0 == 2.0 ;;
- : bool = false
```

Les booléens

```
# false && 1/0>0 ;;
- : bool = false
# true || 1/0>0 ;;
- : bool = true
```

Évaluation paresseuse

Comme en Python, les opérateurs `&&` et `||` sont paresseux : si la partie gauche suffit à déterminer le résultat de l'opération, la partie droite n'est pas évaluée.

Notez que ceci n'empêche pas Caml d'exiger naturellement la cohérence du type. `false && 1` n'a pas de sens et provoquera une erreur.

Déclarations de variables et références

Déclaration globale

Déclaration

Pour déclarer une variable, on utilise `let`.

La déclaration permet de déclarer une variable **globale**, qui ne change pas à moins qu'un autre déclaration globale ne l'écrase.

```
# let x = 44 ;;
val x : int = 44
```

Constantes

Parler de variable ici est abusif : on déclare plutôt une constante.

En effet, on effectue simplement une liaison entre un nom (celui de la variable) et une valeur, mais on ne peut pas la modifier.

Déclaration locale

Déclaration locale

Le même mot clé `let` combiné avec `in` sert à effectuer une déclaration locale.

Si la variable utilisée était déjà associée à une valeur, celle-ci est temporairement oubliée, mais retrouvée à la fin de l'exécution.

Exemple

Voici un moyen de calculer $\sqrt[4]{5} + \sqrt[4]{5}^2 + \sqrt[4]{5}^3$.

```
# let x = 5.**0.25 in x +. x *. x +. x ** 3. ;;
- : float = 7.07511828360311945
# x ;;
- : int = 44
```

Déclaration locale

Remarque

Si le nom de la variable n'était pas lié à une valeur avant la déclaration locale, il ne l'est toujours pas après.

```
# let y = exp(1.0) in (y +. 1.0 /. y) /. 2.0 ;;
- : float = 1.54308063481524371
# y ;;
Characters 0-1:
  y ;;
  ^
Error: Unbound value y
```

Déclaration simultanée

Déclaration simultanée

On peut déclarer simultanément deux variables avec le mot clé **and**.

Cette déclaration peut bien sûr être locale.

```
1  # let x = 0 and y = 1 ;;
2  val x : int = 0
3  val y : int = 1
4
5  # let x=0 and y=1 in x+y ;;
6  - : int = 1
```

Déclaration simultanée

```
# let z=0 and y=z+2 in y+z ;;
Characters 14-15:
  let z=0 and y=z+2 in y+z ;;
                ^
Error: Unbound value z
```

Remarque

Il ne faut pas confondre déclaration locale et simultanée.

Si la variable z n'est pas définie avant l'exécution du code ci-dessus, on obtient une erreur.

Références

Variables

Les variables de **Caml** ne sont donc pas des variables au sens traditionnel des langages de programmation, puisqu'il est impossible de modifier leur valeur.

Il est pourtant souvent nécessaire d'utiliser dans les programmes des variables modifiables comme en **Python**.

Références

En **Caml**, on utilise pour cela une **référence** modifiable vers une valeur, c'est-à-dire une case mémoire dans laquelle on peut lire et écrire le contenu.

Pour créer une référence, on applique le constructeur **`ref`** au contenu initial de la case mémoire.

```
1  # let x = ref 0 ;;
2  val x : int ref = contents = 0
```

Références

Références

La variable `x` est liée à une valeur (de type `int ref`), qui est une référence pointant vers 0 à la création.

Pour lire le contenu d'une référence, on utilise l'opérateur de **déférencement** `!`, qui signifie "contenu de".

```
# !x ;;
- : int = 0
```

Références

Adresse mémoire

De même qu'avec une déclaration "globale" avec `let`, la liaison entre la variable et la case mémoire pointée est définitive jusqu'à ce qu'une nouvelle déclaration écrase cette liaison.

Plus précisément, la valeur de `x` est ici l'**adresse** de la case mémoire (modifiable) qui contient la valeur, et cette adresse est une constante.

Pour modifier le contenu d'une référence on utilise l'opérateur d'affectation `:=`.

Références

Exemple

Par exemple, `x := !x + 1` incrémente le contenu de la case mémoire pointée par `x`, de manière similaire à `x+=1` ou `x=x+1` en **Python**.

```
# x := !x + 1 ;;
- : unit = ()
# !x ;;
- : int = 1
```

Références

Programmation impérative

Les références sont liées à la possibilité de faire de la programmation **impérative** en **Caml** : on les utilisera donc souvent dans des boucles.

Le type de la valeur pointée par une référence est fixé à la création : on a vu précédemment que `x` était de type `int ref`.

On peut de même créer des `bool ref`, `float ref`, ... et des références vers des types plus complexes.

Références

Remarque

Les opérations `x := !x + 1` et `x:= !x - 1` étant d'usage très courant, elles ont un raccourci.

```
# !x ;;
- : int = 1
# incr x ;
- : unit = ()
# !x
- : int = 2
# decr x ; !x ;;
- : int = 1
```

Quelques types plus complexes

Les tuples

Tuples

Les **tuples** (n-uplets en français) sont similaires aux tuples de **Python** : ils permettent de construire des séquences de valeurs de type possiblement différents.

Le **type** d'un tuple $(t_1, t_2, \ldots, t_n)$ est le **produit cartésien** des types de ses éléments.

```
# (true, 0, atan 1.0) ;;
- : bool * int * float = (true, 0, 0.785398163397448279)
```

Remarque

Les parenthèses sont facultatives.

Les tuples

En pratique

En pratique, les tuples seront souvent utilisés comme valeur de retour de fonctions.

On peut récupérer dans des variables les composantes d'un tuple.

```
1  # let a,b = (0, 1.0) ;;
2  val a : int = 0
3  val b : float = 1.
```

Les tuples

Couples

Les fonctions **`fst`** et **`snd`** permettent de récupérer la première et la deuxième composante d'un couple.

Attention, ça ne marche qu'avec des couples (tuples de taille 2).

```
# fst (1, 5.0) ;;
- : int = 1
# snd (0, true) ;;
- : bool = true
```

Les tableaux

array

En **Caml**, les **tableaux** (**array** en anglais) sont très similaires aux listes de **Python**, à deux restrictions près :

- La **taille** du tableau est **fixée** à la création, et ne peut pas changer.
  Il n'y a donc pas d'équivalent aux méthodes `pop` et `append` pour les tableaux **Caml**.
- le type des éléments d'un tableau doit être le même pour tous les éléments.

On aura donc par exemple des `int array`, des `bool array`, des `(int * bool) array`, ...

Les tableaux

Syntaxe

Pour déclarer un tableau, les éléments sont séparés par des points-virgules, et placés entre `[|  |]`.

```
# [|5; 0; 7|] ;;
- : int array = [|5; 0; 7|]
```

Les tableaux

Indices

De même qu'en **Python**, si n est la taille du tableau, les éléments sont indexés de 0 à $n-1$.

Pour l'accès et la modification des éléments, si `t` est un tableau de taille n et i un indice (entre 0 et $n-1$), on utilise :

- `t.(i)` pour récupérer la valeur stockée à l'indice i, et
- `t.(i) <- x` pour la changer en `x`.

```
# let t = [|0; 5; 7|] ;;
val t : int array = [|0; 5; 7|]
# t.(0) <- 2 ;;
- : unit = ()
# t ;;
- : int array = [|2; 5; 7|]
```

Les tableaux

Remarque

L'expression `t.(0) <- 2` a pour type `unit`, mais elle a un **effet de bord** : la modification de la première case du tableau.

Pour parcourir des tableaux, on utilise fréquemment des références et des boucles : on fait donc de la **programmation impérative**.

Fonctions utiles sur les tableaux

Commande	Description
`[|0; 1; 7; 8; 9|]`	construit un tableau par donnée explicite des éléments.
`t.(i)`	renvoie le i-ème élément de `t` $(0 \leq i < n)$, avec n la taille de la liste.
`t.(i) <- x`	remplace le i-ème élément de `t` par `x`.
`Array.length t`	renvoie la longueur de `t`.
`Array.make n x`	construit un tableau de longueur n contenant des `x`.

À retenir

Les commandes ci-dessus sont à retenir **absolument**.

Fonctions utiles sur les tableaux

Commande	Description
`Array.init n f`	construit un tableau contenant les $f(i)$ pour $0 \leq i < n$.
`Array.copy t`	renvoie une copie de `t`.
`Array.sub t i k`	renvoie un tableau de longueur k, égal à la portion de `t` qui démarre à l'indice i.
`Array.append t1 t2`	concatène deux tableaux (ne pas confondre avec Python).
`Array.concat l`	concatène une liste de tableaux.
`Array.make_matrix n m x`	construit une matrice de taille (n, m) contenant des `x`. Les éléments sont accessibles par `t.(i).(j)`.

Fonctions utiles sur les tableaux

Commande	Description
`Array.map f t`	crée un tableaux dont les éléments sont les $f(x)$ pour x dans `t`.
`Array.iter f t`	applique f sur chaque élément de `t` (f est de type `'a -> unit`).
`Array.sort f t`	trie le tableau `t` **en place**, avec la fonction de comparaison `f`. `f x y` renvoie un entier, nul si les éléments sont égaux, > 0 si $x > y$, et < 0 sinon.

Les chaînes de caractères

char et str

Contrairement à **Python**, **Caml** fait la distinction entre un **caractère** (type `char`) et une **chaîne de caractère** (type `string`).

Syntaxe

- Les **caractères** sont encadrés par des **apostrophes**.
- Les **chaînes** sont encadrées par des **guillemets**.

Les chaînes de caractères

Accès à une lettre

L'accès à l'élément d'indice i d'une chaîne `s` se fait avec `s.[i]`.

```
# let s="abc" ;;
val s : string = "abc"
# s.[0] ;;
- : char = 'a'
```

Sémantique

Les chaînes sont très semblables à des `char array`, à ceci près qu'elles sont **immuables**, comme en **Python**.

↪ On ne peut pas modifier la valeur d'un `s.[i]`.

Les chaînes de caractères

Fonctions usuelles

Bien que très semblables à des `char array`, les chaînes de caractères ont leur propre syntaxe, et leurs propres fonctions associées. En voici quelques unes.

Commande	Description
`'a'`	le caractère a.
`"abc"`	la chaîne de caractère abc.
`s.[i]`	le i-ème caractère de `s`.
`String.length s`	longueur de la chaîne `s`.
`String.make n c`	crée la chaîne $ccc\cdots c$ de longueur n.
`String.sub s i k`	extrait la sous-chaîne de `s` de taille k commençant à l'indice i.
`s1^s2`	renvoie la concaténation de `s1` et `s2`.

Les chaînes de caractères

Remarque

La longueur d'une chaîne est directement liée au nombre d'octets utilisés pour représenter un caractère.

Attention aux caractères accentués, qui ne sont pas ASCII. Ce sont des chaînes de taille 2, et pas des caractères.

```
# String.length "é" ;;
- : int = 2
# let c = "é" ;;
val c : string = "\195\169"
# let c = 'é' ;;
Characters 9-10:
  let c = 'é' ;;
          ^
Warning 3: deprecated: ISO-Latin1 characters in identifiers
Error: Syntax error
```

Fonctions

Les fonctions

Fonctions

En **Caml**, une fonction est une valeur, qui a donc un type.

Le type d'une fonction sera de la forme :

type du paramètre $\rightarrow$ type du résultat

```
# int_of_float ;;
- : float -> int = <fun>
# int_of_float (3.5) ;;
- : int = 3
```

Exemple

La fonction `int_of_float` convertie un entier en flottant.

Les fonctions

```
# fst ;;
- : 'a * 'b -> 'a = <fun>
# fst (true, 5.4) ;;
- : bool = true
```

Polymorphisme

La fonction `fst` renvoie le premier élément d'un couple.

Comme les types des composantes de ce couple peuvent être quelconque, **Caml** utilise des types **génériques** (on dit aussi **polymorphes**) pour donner le type de la fonction.

Ici, un couple générique est de type `'a * 'b`, et le résultat de `fst` est du type `'a` car nécessairement du type de la première composante.

Fonctions à un seul argument

Syntaxe

Pour créer une fonction a un seul argument, on utilise `function`.

```
1  # function x -> x+1 ;;
2  - : int -> int = <fun>
```

Exemple

On crée ici la fonction qui à un entier x associe $x + 1$.

Caml détecte tout seul que le type est `int -> int` car l'opérateur **+** n'est valable que sur les entiers.

Fonctions à un seul argument

Variables

Puisqu'une fonction est une valeur, on peut l'affecter à une variable.

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

Fonctions à un seul argument

Appel

Pour effectuer un **appel de fonction**, il y a deux syntaxes possibles :

- `fonction(valeur)` (comme en maths).
- `fonction valeur` (sans parenthèses).

```
# f(5) ;;
- : int = 6
# f 5 ;;
- : int = 6
# (function x -> x +. 1.0) 2.5 ;;
- : float = 3.5
```

Fonctions à un seul argument

Raccourcis syntaxique

Puisque la syntaxe précédente est un peu longue, il existe un raccourcis pour déclarer une fonction a un seul argument.

En mathématiques, on écrit souvent "soit $f(x) = x + 1$". En **Caml**, c'est pareil (mais les parenthèses sont facultatives).

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

Syntaxe

On déclare une fonction à un seul argument avec la syntaxe

`let f x = expression ;;`

Curryfication des fonctions

Mathématiques

En mathématiques, lorsqu'on considère une fonction ayant deux arguments, on considère une application de la forme :

$$\begin{aligned} f : \ E \times F &\to G \\ (x, y) &\mapsto f(x, y) \end{aligned}$$

En **Caml**, il est possible de définir des fonctions prenant un tuple en argument. C'est par exemple le cas pour la fonction `fst` vue précédemment.

Informatique

En informatique (surtout en programmation fonctionnelle), il est plus pratique de procéder autrement.

Curryfication des fonctions

Applications partielles

Notons $\mathcal{F}(E, F)$ l'ensemble des applications de E dans F.

Il existe une bijection naturelle entre les ensembles $\mathcal{F}(E \times F, G)$ et $\mathcal{F}(E, \mathcal{F}(F, G))$:

$$\begin{aligned} \varphi : \ \mathcal{F}(E \times F, G) &\to \mathcal{F}(E, \mathcal{F}(F, G)) \\ f &\mapsto x \mapsto (y \mapsto f(x, y)) \end{aligned}$$

Grâce à cette vision des choses, on peut considérer des **applications partielles**.

Curryfication des fonctions

Exemple

Si une fonction `max` était écrite avec cette vision des choses, ou pourrait facilement construire la fonction $y \mapsto \max(20, y)$ qui donne le maximum entre y et 20.

Curryfication

La plupart des fonctions **Caml** sont données sous cette forme, appelée forme **curryfiée**.

Curryfication des fonctions

```
# max ;;
- : 'a -> 'a -> 'a = <fun>
# max 20 ;;
- : int -> int = <fun>
# max 20 58 ;;
- : int = 58
# Array.make ;; (* création d'un tableau de taille n initialisé avec l'élément x *)
- : int -> 'a -> 'a array = <fun>
# Array.make 5 0 ;;
- : int array = [|0; 0; 0; 0; 0|]
# Array.append ;; (* concaténation de tableaux *)
- : 'a array -> 'a array -> 'a array = <fun>
# Array.append [|2; 0|] ;;
- : int array -> int array = <fun>
```

Curryfication

Voici quelques exemples de fonctions **Caml**.

Toutes les fonctions ci-dessus sont **curryfiées**.

Curryfication des fonctions

```
# max ;;
- : 'a -> 'a -> 'a = <fun>
# max 20 ;;
- : int -> int = <fun>
# max 20 58 ;;
- : int = 58
# Array.make ;; (* création d'un tableau de taille n initialisé avec l'élément x *)
- : int -> 'a -> 'a array = <fun>
# Array.make 5 0 ;;
- : int array = [|0; 0; 0; 0; 0|]
# Array.append ;; (* concaténation de tableaux *)
- : 'a array -> 'a array -> 'a array = <fun>
# Array.append [|2; 0|] ;;
- : int array -> int array = <fun>
```

Curryfication

Pour une fonction curryfiée à deux arguments, `f a b` est équivalent à `(f a) b` : **la fonction a priorité dans l'évaluation**.

Curryfication des fonctions

```
# max ;;
- : 'a -> 'a -> 'a = <fun>
# max 20 ;;
- : int -> int = <fun>
# max 20 58 ;;
- : int = 58
# Array.make ;; (* création d'un tableau de taille n initialisé avec l'élément x *)
- : int -> 'a -> 'a array = <fun>
# Array.make 5 0 ;;
- : int array = [|0; 0; 0; 0; 0|]
# Array.append ;; (* concaténation de tableaux *)
- : 'a array -> 'a array -> 'a array = <fun>
# Array.append [|2; 0|] ;;
- : int array -> int array = <fun>
```

Curryfication

Inversement, une fonction de type `'a -> 'b -> 'c` est en fait une fonction curryfiée de type `'a -> ('b -> 'c)`.

Curryfication des fonctions

```
# max ;;
- : 'a -> 'a -> 'a = <fun>
# max 20 ;;
- : int -> int = <fun>
# max 20 58 ;;
- : int = 58
# Array.make ;; (* création d'un tableau de taille n initialisé avec l'élément x *)
- : int -> 'a -> 'a array = <fun>
# Array.make 5 0 ;;
- : int array = [|0; 0; 0; 0; 0|]
# Array.append ;; (* concaténation de tableaux *)
- : 'a array -> 'a array -> 'a array = <fun>
# Array.append [|2; 0|] ;;
- : int array -> int array = <fun>
```

Curryfication

Tout ceci se généralise naturellement à des fonctions à plus de deux arguments.

Création de fonctions curryfiées

Exemple

L'opérateur + de **Caml** est un opérateur **infixe** (c'est-à-dire qu'il s'utilise sous la forme $a \ \mathrm{op} \ b$), mais on peut le transformer en opérateur **préfixe** (c'est à dire qui s'utilise comme une fonction curryfiée, sous la forme $\mathrm{op} \ a \ b$) en l'encadrant de parenthèses.

```
1  # (+) ;;
2  - : int -> int -> int = <fun>
```

Création de fonctions curryfiées

Exemple

Un moyen d'obtenir une fonction équivalente est d'utiliser plusieurs fois le mot-clé `function`.

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

Syntaxe

Le mot clé `fun` permet de construire directement des fonctions curryfiées à plusieurs arguments.

```
# fun x y -> x+y ;;
- : int -> int -> int = <fun>
# (fun x y -> x+y) 1 6 ;;
- : int = 7
```

Création de fonctions curryfiées

Syntaxe

En général, on préfère donner un nom à la fonction. Pour cela, on peut utiliser `let` et se passer de `fun`, comme pour les fonctions avec un seul argument.

La syntaxe générale pour déclarer une fonction curryfiée à n arguments est : `let f x1 ... xn = expression ;;`

```
# let somme x y = x + y ;;
val somme : int -> int -> int = <fun>
# somme 1 6 ;;
- : int = 7
```

Création de fonctions non curryfiées

Fonctions non curryfiées

Il sera toutefois utile de temps en temps de créer des fonctions non curryfiées.

Par exemple, si on veut travailler avec les points du plan, on travaillera explicitement avec des couples de type `int * int` ou `float * float`.

On peut également déclarer une fonction non curryfiée avec `let`.

Création de fonctions non curryfiées

```
# let f (x,y) = x**2. +. y**2. <= 1. ;;
val f : float * float -> bool = <fun>
```

Exemple

La fonction ci-dessus teste si le couple de flottants passé en paramètre est dans le disque unité fermé.

Une définition équivalente (ci-dessous) déconstruit le couple à l'intérieur du corps de la fonction.
(l'usage de **`fst`** et **`snd`** était également possible)

```
# let f z = let x,y=z in x**2. +. y**2. <= 1. ;;
val f : float * float -> bool = <fun>
```

Conditions et boucles

Expressions conditionnelles

Syntaxe

En **Caml**, la syntaxe d'une **expression conditionnelle** est de la forme `if cond then a else b`, où `cond` est une **expression booléenne**, et `a` et `b` sont des expressions de **même type**.

Le type de l'expression conditionnelle est ce type commun.

Le résultat de l'expression conditionnelle peut être utilisé dans d'autres expressions.

```
1  # if true then 1 else 0 ;;
2  - : int = 1
3  # 1+(if true then 1 else 0) ;;
4  - : int = 2
```

Expressions conditionnelles

Typage

Si `a` et `b` sont deux expressions de types différents, l'interpréteur avertit immédiatement d'une erreur.

```
# if true then 1.0 else 2 ;;
Characters 22-23:
  if true then 1.0 else 2 ;;
                        ^
Error: This expression has type int but an expression was expected of type float
```

Exemple

Ici, l'interpréteur a détecté que la première expression (`1.0`) est de type `float`, la deuxième doit donc également avoir ce type.

Expressions conditionnelles

Remarque

L'absence de `else` est compris comme `else ()`, où `()` est de type `unit`.

L'expression `a` doit alors être de type `unit`.

```
# if true then print_string "c'est vrai !" ;;
c'est vrai !- : unit = ()
```

Expressions conditionnelles

Programmation impérative

Cette possibilité sera par exemple utilisée lorsqu'on programmera de manière **impérative** : une expression modifiant la valeur pointée par une **référence** a pour type `unit`.

```
1  # let x = ref 0 ;;
2  val x : int ref = {contents = 0}
3  # if true then x := !x + 1 ;;
4  - : unit = ()
```

Séquence d'instructions

Séquence d'instructions

En **Caml**, deux expressions successives sont séparées par un **point-virgule**.

Dans une suite d'expressions successives, **elles doivent toutes avoir le type `unit`, sauf peut-être la dernière**.

Le type de l'expression totale est celui de la dernière expression.

```
1  # let x = ref 0 in x := !x + 1 ; print_int 8 ; print_string " encore une expression " ; !x ;;
2  8 encore une expression - : int = 1
```

Séquence d'instructions

Séquence dans une expression conditionnelle

Lorsqu'on utilise la construction **if**, **then**, **else**, il ne doit y avoir qu'une seule expression dans le **then** et le **else**.

On peut délimiter une séquence d'instructions par **begin** et **end**.

```
1  # let x = ref 0 in if !x > 0 then x:= !x + 1 ; print_int !x ;;
2  0- : unit = ()
3  # let x = ref 0 in if !x > 0 then begin x:= !x + 1 ; print_int !x end ;;
4  - : unit = ()
```

Exemple

Dans le premier cas, **print_int** est exécuté car en dehors du **if** ... **then**.

Dans le deuxième cas, on a encadré avec **begin** et **end**.

Séquence d'instructions

Remarque

On peut éviter l'usage de **begin** et **end** en utilisant des parenthèses.

C'est plus lisible lorsqu'on écrit tout sur une ligne.

On préférera l'usage de **begin** et **end** lorsqu'on écrira des programmes sur plusieurs lignes (avec de l'indentation).

```
1  # let x = ref 0 in if !x > 0 then (x := !x + 1 ; print_int !x) ;;
2  - : unit = ()
```

Boucle `for`

for

La syntaxe d'une **boucle** `for` est la suivante :
`for i = i1 to i2 do instructions done`.

Le compteur de boucle (`i` ici) prend toutes les valeurs entre `i1` et `i2`, par pas de 1. Si `i2` $<$ `i1`, aucune instruction n'est exécutée.

Les instructions, séparées par des **points-virgules**, ont toutes type `unit`, qui est aussi le type de la boucle totale.

Il n'est pas possible de modifier le compteur de boucle.

Boucle `for`

Exemple

Voici une fonction, écrite dans un style impératif, qui calcule la factorielle d'un entier.

```
let fact n =
  let y = ref 1 in
  for i=1 to n do
    y:= !y * i
  done ;
  !y
;;
```

```
# fact 5 ;;
- : int = 120
```

Boucle `for`

Pas négatif

La syntaxe `for i = i1 downto i2 do instructions done` permet d'avoir un pas de -1.

Il faut donc que `i1` $\geq$ `i2` pour qu'au moins une instruction soit exécutée.

Boucle `while`

while

La syntaxe d'une **boucle** `while` est très similaire :

`while condition do instructions done`, où **`condition`** est une **expression booléenne**.

Boucle `while`

Exemple

L'**algorithme d'Euclide** suivant, de type `int -> int -> int`, est écrit avec une **boucle** `while`.

Notez l'utilisation d'une variable `r` locale au corps de la boucle, tandis que les références sont locales à la fonction.

```
1  let pgcd a b =
2    let x = ref a and y = ref b in
3    while !y > 0 do
4      let r = !x mod !y in
5      x := !y ;
6      y := r
7    done ;
8    !x
9  ;;
```

```
1  # pgcd 451 123 ;;
2  - : int = 41
```

Filtrages

Filtrage

Filtrage

Le filtrage peut être vu comme une alternative aux `if`, `else`, qui peuvent s'enchaîner de manière disgracieuse, mais est en réalité beaucoup plus que ça.

Filtrage

```
1  let sgn x = if x=0 then 0 else abs(x)/x ;;
```

Exemple

D'une part, il peut être utilisé pour construire une fonction "par morceaux". Par exemple, la fonction suivante peut être implémentée comme ci-dessus.

$$\begin{aligned} \mathrm{sgn} : \ & \mathbb{Z} \longrightarrow \mathbb{Z} \\ & x \longmapsto \begin{cases} 0 & \text{si } x = 0 \\ 1 & \text{si } x > 0 \\ -1 & \text{si } x < 0 \end{cases} \end{aligned}$$

Filtrage

Exemple

Une autre manière est d'utiliser un **filtrage par motif** (**pattern matching** en anglais), avec le mot-clé `function`.

```
1  let sgn = function
2    | 0 -> 0
3    | x -> abs(x)/x
4  ;;
```

Filtrage

Pattern matching

Ceci s'étend naturellement à des fonctions de plusieurs arguments, avec l'utilisation de `fun`.

Néanmoins, l'utilisation de `fun` et `function` étant assez trompeuse, on préfère filtrer avec un `match ... with`.

```
1  let sgn x = match x with
2    | 0 -> 0
3    | y -> abs(y)/y
4  ;;
```

Filtrage

Filtrage

Plus généralement, un **filtrage** permet de gérer différents motifs possibles d'une expression.

D'autre part, il permet d'accéder aux composantes d'un type construit (en particulier récursif), ce qui sera très utile lorsqu'on aura vu les listes et les arbres par exemple.

Règles du filtrage par motifs

Cas successifs

Lors d'un filtrage, les cas successifs sont examinés un par un, et le premier qui "colle" à l'expression examinée est réalisé, pas les autres.

Exemple

Voici une réécriture du "ou exclusif" en Caml, sur les booléens.

```
1  let xor a b = match (a,b) with
2    | true, true -> false
3    | false, true -> true
4    | true, false -> true
5    | false, false -> false
6  ;;
```

Règles du filtrage par motifs

Typage

Dans une instruction de la forme `if... else...`, les expressions dans chaque bloc `if` et `else` doivent avoir le même type.

Il en va de même des expressions qui sont résultats d'un filtrage.

```
# match true with | true -> 0 | false -> "erreur" ;;
Characters 39-47:
  match true with | true -> 0 | false -> "erreur" ;;
                                         ^^^^^^^^
Error: This expression has type string but an expression was expected of type int
```

Règles du filtrage par motifs

Filtrage exhaustif

Un filtrage se doit d'être **exhaustif**, c'est-à-dire qu'il doit pouvoir filtrer toutes les valeurs possibles de l'expression en fonction de son type.

Règles du filtrage par motifs

Exemple

Dans la fonction suivante, qui simule le lancé d'un dé à 6 faces et indique une action, le filtrage n'est pas exhaustif (du point de vue de l'**interpréteur Caml**, qui ne voit que les types).

En effet, la fonction est acceptée, mais le filtrage non exhaustif est signalé.

```
let action() = match 1+Random.int 6 with
  | 1 -> "un pas à droite"
  | 2 -> "un pas à gauche"
  | 3 -> "demi tour"
  | 4 -> "grand écart"
  | 5 -> "saut perilleux"
  | 6 -> "vrille"
;;
```

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: 0
val action : unit -> string = <fun>
```

Règles du filtrage par motifs

Motif joker

Avoir des filtrages non exhaustifs est disgracieux et doit être évité. Le motif "**joker**" _ (**underscore**) permet de filtrer tous les motifs possibles (ou une partie d'un motif).

Exemple

La dernière ligne du filtrage précédent peut avantageusement être remplacée par un **joker**.

```
1  let action() = match 1+Random.int 6 with
2    | 1 -> "un pas à droite"
3    | 2 -> "un pas à gauche"
4    | 3 -> "demi tour"
5    | 4 -> "grand écart"
6    | 5 -> "saut perilleux"
7    | _ -> "vrille"
8  ;;
```

Règles du filtrage par motifs

Cas inutile

De même que l'interpréteur indique si un filtrage est non exhaustif, il indique aussi si un cas de filtrage est **inutile**.

```
# match true with | _ -> 0 | false -> 1 ;;
Toplevel input:
> match true with | _ -> 0 | false -> 1 ;;
>                            ^^^^^
Warning: this matching case is unused.
- : int = 0
```

Règles du filtrage par motifs

Filtrage par motifs

Le filtrage d'une expression s'effectue par **motifs** et non par valeurs : la "**forme**" de l'expression située à gauche qui est comparée à la valeur filtrée.

Tout cela sera plus clair lorsqu'on aura vu les types construits et les listes, mais un motif est essentiellement :

- une constante (`true`, 0, `(0, "a")`, etc...) ;
- un identificateur ;
- le joker `_` ;
- une construction de motifs à l'aide de motifs plus simples.

Règles du filtrage par motifs

Exemple

On peut déjà faire un exemple avec les couples.

Le filtrage suivant réalise des actions différentes suivant que le couple filtré possède une composante nulle ou non.

```
1  match c with
2    | (0,_) -> ...
3    | (_,0) -> ...
4    | _ ->
```

Règles du filtrage par motifs

Liaison locale

Lorsqu'un ou plusieurs identificateurs se trouvent dans le motif et que le filtrage réussit, une **liaison locale** est effectuée entre les identificateurs et les valeurs filtrées.

Exemple

```
1  match c with
2    | (0,y) -> y
3    | (x,0) -> x
4    | (x,y) -> -x-y
```

Remarque

Notons que ce filtrage est bien **exhaustif** : `(x,y)` filtre tous les couples possibles.

Règles du filtrage par motifs

Remarque

Dans un motif, ne peuvent figurer que des identificateurs **distincts** : par exemple filtrer `(x,x)` n'a aucun sens.

Le mot clé `when` permet de relâcher un peu la rigidité du filtrage sur motif : une fois la liaison effectuée, on peut réaliser une comparaison de valeurs.

Exemple

Voici une réécriture de la fonction `xor`.

```
1  let xor a b = match (a,b) with
2    | (x,y) when x=y -> false
3    | _ -> true
4  ;;
```

Règles du filtrage par motifs

Exemple

Voici une écriture de la fonction **`arg`** donnant un argument d'un nombre complexe identifié à un couple de flottants, avec un filtrage (on rappelle que $\pi = 4\arctan(1)$).

```
1  let arg (x,y) = match (x,y) with
2    | (0., 0.) -> failwith "zero"
3    | (0., y) when y>0. -> 2. *. atan 1.
4    | (0., y) -> -. 2. *. atan 1.
5    | (x,_) when x>0. -> atan (y/. x)
6    | _ -> atan (y/. x) +. 4. *. atan 1.
7  ;;
```

Types enregistrement et types somme

Types enregistrements et types sommes

Nouveaux types

En **Caml**, tout objet doit avoir un type.

On va maintenant voir comment construire de nouveaux types à partir de types existants ou non.

Types enregistrement

Les **types enregistrement** (ou **types produit**) sont similaires à des n-uplets, donc à un produit cartésien de types, à quelques différences près :

- les composantes ont des **noms** (**étiquettes** ou **champs**) ;
- on peut déclarer une ou plusieurs composantes comme **modifiable** (ou **mutable**) ;
- il n'y a pas d'ordre dans les champs d'un enregistrement.

L'utilisation d'enregistrements, en particulier modifiables, se fait beaucoup en programmation **impérative**.

Types enregistrement

```
# type personne = {nom: string ; prenom: string ; age: int} ;;
type personne = { nom : string; prenom : string; age : int; }
# let a = {nom="Dupond" ; age=42 ; prenom="Jean"} ;;
val a : personne = {nom = "Dupond"; prenom = "Jean"; age = 42}
# a.nom ;;
- : string = "Dupond"
```

Exemple: syntaxe et utilisation

On a déclaré ci-dessus un type **`personne`**, dont les champs sont **`nom`**, **`prenom`** et **`age`**, associés à un type particulier.

Attention, le nom d'un type et les noms des champs doivent être en **minuscules**.

Pour créer un élément de type personne, il suffit de fixer les valeurs des champs.

On accède au champ **`c`** d'un élément **`a`** d'un type produit avec **`a.c`**.

Types enregistrement et filtrages

Exemple: filtrage

Les motifs de filtrage peuvent être des types produits.

Par exemple pour tester si une personne est majeure.

```
1  let est_majeur p = match p with
2    | {prenom = _ ; nom = _ ; age = x} when x>=18 -> true
3    | _ -> false
4  ;;
```

```
let est_majeur p = match p with
  | {age = x} when x>=18 -> true
  | _ -> false
;;
```

Remarque

En fait, il n'est pas nécessaire de préciser tous les enregistrements dans un filtrage.

La fonction de droite est équivalente.

Types enregistrement

Champ modifiable

A priori, l'âge d'une personne peut changer.

On aurait pu rendre le champ **age mutable**.

La modification d'un champ mutable `c` de l'élément `a` en la valeur `x` se fait avec `a.c <- x`.

```
# type personne = {nom: string ; prenom: string ; mutable age: int} ;;
type personne = {nom: string ; prenom: string ; mutable age: int;} ;;
# let a = {nom="Dupond" ; age=42 ; prenom="Jean"} ;;
val a : personne = {nom = "Dupond"; prenom = "Jean"; age = 42}
# a.age <- 43 ;;
- : unit = ()
# a;;
- : personne = {nom = "Dupond"; prenom = "Jean"; age = 43}
```

Types enregistrement

Types paramétrés

On peut faire usage du **polymorphisme** dans les types produits.

Exemple

Définissons nous même un type similaire aux couples, mais en imposant des éléments homogènes.

```
# type 'a couple = {f: 'a ; s: 'a} ;;
type 'a couple = { f : 'a; s : 'a; }
# {f=5; s=2} ;;
- : int couple = {f = 5; s = 2}
# {f=true; s=false} ;;
- : bool couple = {f = true; s = false}
```

Types enregistrement

Remarque

Bien sûr, on peut utiliser autant de types polymorphes que nécessaire.

```
# type ('a, 'b, 'c) truc = {un: 'a ; deux: 'b ; trois: ('a * 'c) array ; quatre: bool} ;;
type ('a, 'b, 'c) truc = {un : 'a; deux : 'b; trois : ('a * 'c) array; quatre : bool;}
```

Types enregistrement

```
1  type 'a reference_perso = {mutable contenu: 'a} ;;
2  let creer_ref x = {contenu = x} ;;
3  let acceder_ref r = r.contenu ;;
4  let modifier_ref r x = r.contenu <- x ;;
```

Exemple: recréer manuellement les références

Il est intéressant de voir que l'on peut recréer "manuellement" un type semblable au type **`ref`** de **Caml** à l'aide d'un enregistrement, contenant un seul champ (naturellement mutable).

Pour pouvoir créer des références vers des types quelconques, on fait naturellement usage de polymorphisme en créant un type paramétré.

Types enregistrement

```
type 'a reference_perso = {mutable contenu: 'a} ;;
let creer_ref x = {contenu = x} ;;
let acceder_ref r = r.contenu ;;
let modifier_ref r x = r.contenu <- x ;;
```

```
type 'a reference_perso = { mutable contenu : 'a; }
val creer_ref : 'a -> 'a reference_perso = <fun>
val acceder_ref : 'a reference_perso -> 'a = <fun>
val modifier_ref : 'a reference_perso -> 'a -> unit = <fun>
```

```
# ref ;;
- : 'a -> 'a ref = <fun>
# ( ! ) ;;
- : 'a ref -> 'a = <fun>
# ( := ) ;;
- : 'a ref -> 'a -> unit = <fun>
```

Exemple

À la compilation, voici les types obtenus (à gauche).

On remarque que ceux-ci sont très similaires aux opérations semblables avec les références de **Caml** (à droite).

Parenthéser un opérateur **infixe** permet de le transformer en opérateur **préfixe**, i.e. en une fonction.

Types somme

Types somme

Les types produit correspondait à des produits cartésiens, les **types somme** correspondent à des **unions disjointes**.

De même que l'on utilise des champs pour désigner les composantes d'un type produit, on utilise des **constructeurs** pour indiquer dans quelle partie de l'union disjointe on se situe.

Les constructeurs peuvent être constants, ou d'un certain type.

Types somme

Exemple

Voici d'abord un type constitué de constructeurs constants, qui redéfinissent les booléens.

```
# type booleen = Vrai | Faux ;;
type booleen = Vrai | Faux
# Vrai ;;
- : booleen = Vrai
```

Définition

Un tel type constitué uniquement de constructeurs constants est dit **énuméré**.

Types somme

Exemple

Voici un type mélangeant entiers et flottants.

On utilise deux constructeurs aux noms explicites.

```
# type nombre = Ent of int | Flo of float ;;
type nombre = Ent of int | Flo of float
# Flo 4.5 ;;
- : nombre = Flo 4.5
```

Types somme

```
# type carte_tarot = Excuse | Roi | Dame | Cavalier | Valet
                   | Atout of int | Petite_carte of int ;;
type carte_tarot = Excuse | Roi | Dame | Cavalier | Valet | Atout of int | Petite_carte of int
# Atout 21 ;;
- : carte_tarot = Atout 21
```

Exemple

On peut bien sûr mélanger constructeurs constants ou non.

(Oui, il manque la couleur. On pourrait facilement l'intégrer.)

Types somme et filtrage

Fonctions sur les types somme

On fonctionne énormément par filtrage dans l'utilisation de types somme.

```
1  let negation b = match b with
2    | Vrai -> Faux
3    | Faux -> Vrai
4  ;;
```

```
1  let valeur_absolue x = match x with
2    | Ent a -> Ent (abs a)
3    | Flo a -> Flo (abs_float a)
4  ;;
```

Exemple

Voici une fonction de négation sur nos booléens fraichement redéfinis, et une autre sur les nombres.

Types somme et filtrage

```
1  let point_tarot c = match c with
2    | Roi | Excuse -> 4.5
3    | Dame -> 3.5
4    | Cavalier -> 2.5
5    | Valet -> 1.5
6    | Atout x when x=1 || x=21 -> 4.5
7    | _ -> 0.5
8  ;;
```

Exemple

Voici une fonction qui donne la valeur d'une carte de tarot.

On remarque que Roi et Excuse ont été filtrés ensemble.

Il est possible de procéder ainsi lorsque dans les motifs de filtrage les identificateurs sont les mêmes (et ont les mêmes types).

Types somme

Paramétrage

Comme pour les types produit, on peut paramétrer en faisant usage de **polymorphisme**.

```
1  type ('a,'b) union = A of 'a | B of 'b ;;
```

Exemple

Voici comment faire quelque chose qui ressemble à $A \cup B$ avec A et B disjoints.

On peut bien sûr l'utiliser pour manipuler deux copies d'un même ensemble : quelque chose comme $\mathbb{Z} \cup \mathbb{Z}'$, où $\mathbb{Z}'$ est une copie de $\mathbb{Z}$, serait représenté par un **`(int * int) union`**.

Exceptions

Exceptions

Détection de cas particuliers

Lorsqu'on écrit un programme, il arrive que certaines valeurs soient interdites ou requièrent un traitement particulier.

En programmation impérative, on pourrait gérer ce genre de situations avec des variables booléennes.

Exceptions

Un moyen (pas explicitement au programme) pour programmer efficacement en évitant le lourd usage d'une référence vers un booléen est d'utiliser des **exceptions**.

Exceptions

```
# 1/0 ;;
Exception: Division_by_zero.
# [|5; 2|].(2) ;;
Exception: Invalid_argument "index out of bounds".
# failwith "echec" ;;
Exception: Failure "echec".
```

Exemple

Il y a 3 exceptions différentes dans cet exemple : `Division_by_zero`, `Invalid_argument` et `Failure`.

Les deux dernières prennent en paramètre une chaîne de caractères.

Exceptions

exn

Les **exceptions** en **Caml** forment un type à part entière, qui est le type **`exn`** (abbréviation de exception, sans surprise).

```
1  # Division_by_zero ;;
2  - : exn = Division_by_zero
3  # Failure "truc" ;;
4  - : exn = Failure "truc"
```

Exceptions

Utilisation

À priori, une **exception** dans un programme **Caml** est **levée** lorsqu'on tente de faire une **opération interdite** : diviser par zéro, accéder à un indice d'un tableau qui n'est pas défini (comme ci-dessus), ...

La levée d'une exception **interrompt** le déroulement du programme, et l'exception **remonte** de fonctions appelées en fonctions appelantes jusqu'au programme principal, sauf si elle est **rattrapée**.

Pour **rattraper** une exception, il suffit d'encadrer un code pouvant produire une exception par `try ... with`, et de rattraper l'exception dans le `with`.

Exceptions

```
1  let sgn x =
2    try
3      x/abs(x)
4    with Division_by_zero -> 0
5  ;;
```

Exemple

La fonction ci-dessus est une implémentation **Caml** de :

$$\begin{aligned} \text{sgn} : \ & \mathbb{Z} \longrightarrow \mathbb{Z} \\ & x \longmapsto \begin{cases} 0 & \text{si } x = 0 \\ 1 & \text{si } x > 0 \\ -1 & \text{si } x < 0 \end{cases} \end{aligned}$$

Rattrapage d'une exception

Le mécanisme de **rattrapage** des exceptions à l'intérieur d'un `with` est celui d'un **filtrage**.

On a vu que certaines exceptions pouvaient prendre en paramètre une valeur d'un certain type, c'est le cas de `Failure` (exception produite par `failwith`), qui est un constructeur à un argument (de type `string`).

Exceptions

```
1  let sgn x =
2    try
3      if x=0 then failwith "division par zero !" ;
4      x/abs(x)
5    with Failure s -> 0 (* filtrage: s est liée localement à la valeur associée à Failure *)
6  ;;
```

Exemple

Voici la même fonction que précédemment, avec **`failwith`**.

Exceptions

raise

Il est possible de lever nous même des exceptions (autres que `Failure` avec `failwith`) via la fonction `raise`.

```
# raise ;;
- : exn -> 'a = <fun>
```

Exceptions

Exemple

Dans le code précédent, on peut donc remplacer l'instruction `failwith ...` par `raise (Failure ...)` : c'est strictement équivalent.

Refaisons le avec `Division_by_zero` qu'on lève "manuellement".

```
1  let sgn x =
2    try
3      if x=0 then raise Division_by_zero ;
4      x/abs(x)
5    with Division_by_zero -> 0
6  ;;
```

Création d'exceptions

Enfin, il est possible de créer nous même des exceptions, simplement avec :

```
exception nom_nouvelle_exception ;;
```

On peut également créer une exception prenant un paramètre d'un certain type, avec :

```
exception nom_nouvelle_exception of type ;;
```

Exceptions

```
1  exception Trouve ;;
2
3  let appartient t x =
4    try
5      for i=0 to Array.length t - 1 do
6        if t.(i) = x then raise Trouve
7      done ;
8      false
9    with Trouve -> true
10 ;;
```

Exemple

Voici un code d'une fonction permettant de chercher si un élément se trouve dans un tableau, sans référence.

On crée une exception `Trouve` qu'on pourra lever pour indiquer qu'on a trouvé l'élément.

Bien sûr, on aurait pu lever n'importe quelle exception (comme `Division_by_zero`), mais c'est plus clair ainsi.

Exceptions

```
1  exception Indice of int ;;
2
3  let indice t x =
4    try
5      for i=0 to Array.length t - 1 do
6      if t.(i) = x then raise (Indice i)
7      done ;
8    -1
9    with Indice a -> a
10 ;;
```

Exemple

La fonction ci-dessus renvoie l'indice d'un élément dans un tableau, et -1 si l'élément n'y est pas.

Elle utilise une exception "de type entier".