Fonctions

1 Définition d'une fonction

1.1 Syntaxe et typage d'une fonction

1.1.a. Définition d'une fonction anonyme

La syntaxe pour définir une fonction utilise le mot clé « `function` » et le symbole « `->` » par lequel on définit, de la façon suivante, le nom d'une variable locale (**paramètre** de la fonction) ainsi que l'expression à calculer à partir de la valeur que prendra cette variable au moment de l'appel :

```
# function x -> <expression à évaluer>
```

`Caml` affecte à une fonction ainsi définie, un type dénotant le type de la valeur en entrée puis, après un symbole « `->` » le type de la valeur en sortie :

```
- : type_entree -> type_sortie = <fun>
```

Plus précisément on dit que le couple (type_entree, type_sortie) donne la **signature de la fonction**.

1.1.b. Typage d'une fonction et polymorphisme

Comme pour les autres valeurs, Caml infère le type de la fonction à partir des contraintes posées par la définition de la fonction :

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

Si aucune contrainte ne pèse sur le type de la valeur en entrée, Caml affecte un type générique à la valeur entrée et à la valeur en sortie :

```
# function x -> x;;
- : 'a -> 'a = <fun>
```

Les types génériques sont notés avec une apostrophe (« *quote* » en anglais) et une lettre (`a`, `b`, …, `z`, `a1`, *etc.*)

On dit dans ces cas que la fonction est **polymorphe**.

1.1.c. Contraintes explicites sur la signature d'une fonction

Il est possible de spécifier la signature d'une fonction de façon explicite en imposant le type de la valeur entrée ou en sortie, ou les deux, et d'aller ainsi à l'encontre d'un typage de nature polymorphe :

```
# function (x : int) -> 2.0;;
- : int -> float = <fun>
# function x -> (x : int);;
- : int -> int = <fun>
```

Dans tous les cas, le compilateur de Caml analyse la cohérence des types

```
# function (x : int) -> (x : float);;
Error: This expression has type int but an expression was expected of type
         float
   1: function (x : int) -> (x : float);;
```

1.2 Définition pratique d'une fonction

Pour attacher un nom à une fonction, on procède comme pour attacher un nom à une valeur :

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

On notera la façon équivalente de définir la même fonction `f`, sans utiliser le mot clé `function` :

```
# let f x = x * x ;;
```

Lorsqu'un langage introduit ainsi des raccourcis syntaxiques, on parle de « sucre syntaxique ».

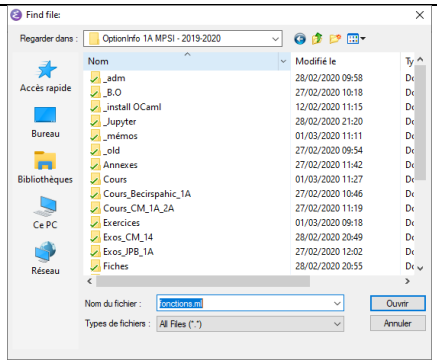
Pour définir une fonction dans un fichier :

- Ouvrir Emacs avec le raccourci « Démarrer »
- Choisir dans le menu d'emacs :

File > Visit New File...

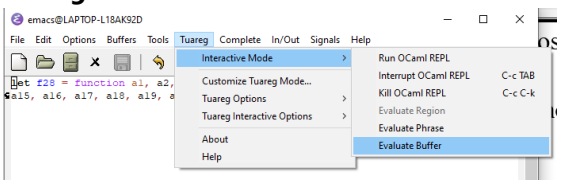
- Choisir l'emplacement dans l'arborescence des répertoires
- Donner un nom au fichier
- Ajouter l'extension .ml
- Cliquer sur « Ouvrir »

Si on utilise le raccourci Emacs_Tuareg ouvrant le fichier init.ml, choisir dans le menu d'emacs **File > Save As...**



Pour exécuter le fichier avec sortie dans la console interactive :

Tuareg > Interactive Mode > Evaluate Buffer



ou bien le raccourci **Ctrl+c** suivi de **Ctrl+b**.

Fonctions.ml	**Résultat d'exécution**

`let f = function x -> x + 1 ;;`	`OCaml version 4.02.1+ocp1`
	
	`# let f = function x -> x + 1;;`
	`val f : int -> int = <fun>`

1.3 Appel à une fonction

L'appel à une **fonction anonyme** (à laquelle il n'a pas été attaché de nom) peut se faire « à la volée » :

```
# (function x -> 2 * x) 3;;
- : int = 6
```

Pour attacher un nom, `f`, à une fonction,

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

on peut utiliser la syntaxe usuelle utilisant des parenthèses

```
# f(4);;
- : int = 5
```

mais **on préférera en Caml la forme suivante, sans parenthèses**

```
# f 4;;
- : int = 5
```

On notera que l'appel à une fonction avec une entrée dont le type n'est pas conforme au type de la fonction déclenche une erreur, ainsi pour la fonction définie ci-dessus :

```
# f 3.;;
Characters 2-4:
  f 3.;;
    ^^
Error: This expression has type float but an expression was expected of
type
         Int
```

1.4 Fonction agissant par effet de bord

Il arrive qu'il n'y ait aucune valeur d'intérêt à renvoyer, car le rôle de la fonction est d'interagir avec l'environnement extérieur (envoyer une commande à un périphérique, écrire une valeur en mémoire). C'est le cas par exemple pour les fonctions d'affichage.

On parle dans ce cas de **fonction agissant par effet de bord** (on oppose aussi parfois ce genre de fonctions aux fonctions destinées à renvoyer une valeur en les nommant « procédures » plutôt de « fonctions »)
Cependant en **Caml**, toute fonction prend un paramètre et renvoie une valeur, aussi dans ces cas de figure, il est convenu que la fonction renverra la valeur `()` (unique valeur du type `unit`) :

```
# print_int;;
- : int -> unit = <fun>
```

Dans ce qui nous occupera, les fonctions agissant par effet de bord seront principalement les fonctions agissant sur les valeurs en mémoire, par modification de références ou d'objets de type mutable, comme les tableaux (programmation de style impératif).

1.5 Fonctions « sans paramètre »

Il arrive aussi qu'il n'y ait aucune valeur pertinente à transmettre en entrée d'une fonction, dans ce cas, pour ne pas contrevenir au fait que toute fonction doit avoir un paramètre (recevoir une valeur en entrée lors de l'appel) on convient d'imposer le type `unit` à l'entrée.
C'est le cas de la fonction `print_newline` qui permet de le passage à la ligne dans les affichages

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

- : unit = ()
```

Ce sera le cas aussi pour une fonction constante, que l'on pourra définir ainsi :

```
# let f = function () -> 2.0;;
val f : unit -> float = <fun>
# f ();;
- : float = 2.
```

Note : pour définir une fonction constante, pour des raisons de cohérence (pour l'addition ou la composition avec une autre fonction par exemple) préférer une syntaxe (par exemple) : `let f = function (x : float) -> 2.0 ;;`.
Remarque : la définition précédente d'une fonction constante égale à deux procède en fait par filtrage (voir suite de ce cours), une définition conforme à ce qui précède serait

```
# let f = function (x : unit) -> 2.0;;
val f : unit -> float = <fun>
```

ou

```
# let f = function (_ : unit) -> 2.0 (* utilisant une variable anonyme, _, en entrée*);;
val f : unit -> float = <fun>
```

2 Fonctions à plusieurs paramètres

Notons préliminairement que, fondamentalement, une fonction, en **Caml**, ne prend, toujours, qu'**un unique argument et ne renvoie qu'une unique valeur**.

2.1 Présentation des types produits

2.1.a. Types pour les produits cartésiens

Sur les types de base présentés au chapitre précédent, il est possible en Caml de construire d'autres types sur lesquels nous reviendrons ultérieurement, mais dont les premiers sont les **types-produit**.
Les types-produits sont les types des n-uplets de valeurs et sont dénotés de la façon suivante :

$$\mathbf{type_1 * type_2 * \dots * type_n}\ ,$$

tandis que les n-uplets de valeurs s'écrivent la forme parenthésée suivante :

$$\mathbf{(valeur_1, valeur_2, \dots, valeur_n)}\ ,$$

ou, si cela ne cause pas d'ambiguïté, sous la forme $\mathbf{valeur_1, valeur_2, \dots, valeur_n}$.

- Exemples :

```
# 0, 5., true;;
- : int * float * bool = (0, 5., true)
```

```
# let u = 1, (2, 3);;
val u : int * (int * int) = (1, (2, 3))
```

2.1.b. Récupération des composantes d'un n-uplet

Pour les paires (2-uplets), et uniquement pour les paires, on dispose de deux fonctions, `fst` et `snd`, pour récupérer les première et deuxième composantes d'une paire :

```
# let t = ('a', "artifice");;
val t : char * string = ('a', "artifice")
# fst t;;
- : char = 'a'
# snd t;;
- : string = "artifice"
```

Pour les n-uplets avec $n > 2$, il convient si on le souhaite, de définir des fonctions de projection, f_i ($1 \leq i \leq n$), afin de réaliser les opérations $f_i(x_1, x_2, \dots, x_n) = x_i$, ou bien, on peut utiliser le mécanisme d'affectation multiple déjà vu :

```
# let a, b, c = (1, 2, 3);;
val a : int = 1
val b : int = 2
val c : int = 3
```

```
# let (x, y) = (0., 1.);;
val x : float = 0.
val y : float = 1.
```

2.2 Fonctions à plusieurs paramètres ou plusieurs valeurs de retour

Les types-produit permettent de définir des fonctions renvoyant plusieurs valeurs, sous la forme d'un n-uplet de valeurs :

```
# let f x = (2 * x, x * x);;
val f : int -> int * int = <fun>
```

```
# f 3;;
- : int * int = (6, 9)
```

mais aussi de définir des fonctions admettant plusieurs paramètres en entrée, passés sous la forme d'un n-uplet :

```
# let f = function (x, y) -> (x + y, x * y);;
val f : int * int -> int * int = <fun>
# f (2, 3);;
- : int * int = (5, 6)
```

2.3 Fonctions à plusieurs paramètres curryfiées

2.3.a. Principe de la curryfication d'une fonction

Il est possible de définir une fonction à plusieurs paramètres sous une autre forme dite curryfiée (du nom du logicien et mathématicien Haskell Brooks Curry).

Cette façon de définir une fonction à plusieurs paramètres revient en fait à définir plusieurs fonctions à un paramètre ayant pour valeur de retour une fonction.

Par exemple, la définition de la fonction suivante, déclinée ci-dessous sous ses deux formes équivalentes (*noter que les parenthèses sont nécessaires*) :

```
# let f = function (x, y) -> 10 * x + y;;
val s : int * int -> int = <fun>
```

```
# let f (x, y) = 10 * x + y;;
val s : int * int -> int = <fun>
```

sera remplacée par la définition suivante :

```
# let f_curry = function x -> (function y -> 10 * x + y);;
val f_curry : int -> int -> int = <fun>
```

Avec la première définition, la fonction `f` prend en paramètre un couple de valeurs, et on peut, par exemple, procéder à l'appel suivant :

```
# f (1, 2);;
- : int = 12
```

Avec la seconde définition (fonction curryfiée), le même résultat est obtenu par l'évaluation de l'expression suivante :

```
# (f_curry 1) 2;; (* parenthèses non nécessaires *)
- : int = 12
```

qui est à décomposer en l'appel à la fonction `f_curry`, pour la valeur 1 de son paramètre x,

```
# f_curry 1;;
- : int -> int = <fun>
```

appel qui renvoie la fonction « `function y -> 10 * 1 + y` », ce qui est mis en évidence ci-dessous :

```
# let f1 = f_curry 1;;
val f1 : int -> int = <fun>
# f1 0;;
- : int = 10
# f1 3;;
- : int = 13
```

2.3.b. Définition de fonctions curryfiées

La définition de fonctions curryfiées se fait de la façon explicite suivante à l'aide du mot-clé `fun` :

`fun x1 x2 … xn -> <expression faisant intervenir (ou pas 😉) les paramètres>,`

où les n paramètres ne doivent pas être séparés par des virgules mais par des espaces.

- Exemples :

```
# let f0 = fun x y z -> x + y + z;;
val f0 : int -> int -> int -> int = <fun>
```

où le type de la fonction `f0` se comprend de façon explicite de la façon suivante :

`int -> (int -> (int -> int))`,

traduisant que `f0` est une fonction qui à une valeur entière `x`, associe une fonction qui, à une valeur entière `y`, associe une fonction qui, à une valeur entière `z` associe la somme `x + y + z`.

```
# let f1 = fun x y z -> (x, y, z);;
val f1 : 'a -> 'b -> 'c -> 'a * 'b * 'c = <fun>
```

où l'on observe que les types des valeurs entrée et du triplet en sortie sont polymorphes.

Il est à noter la **façon abrégée de définir une fonction curryfiée**, dès lors que l'on lui associe un nom :

`let f x1 x2 … xn = <expression>`

L'appel à une fonction curryfiée ne nécessite aucun parenthésage.

```
# let f0 x y z = x + y + z;;
# f0 1 2 3;;
- int : 6
```

L'expression `f0 1 2 3` est comprise comme `((f0 1) 2) 3`, ce que l'on retiendra en retenant que

les expressions (juxtaposées) sont <u>associées à gauche</u>.

On a vu ci-avant, qu'à l'inverse, pour une fonction, qu'un type `int -> int -> int -> int` est à interpréter comme étant le type , ce que l'on retiendra en retenant que

le typage des fonctions est associatif à droite.

2.3.c. Fonctions prédéfinies (« *built-in functions* ») curryfiées

Citons là les fonctions polymorphes `min` et `max` sont des fonctions curryfiées :

```
# min, max;;
- : ('a -> 'a -> 'a) * ('b -> 'b -> 'b) = (<fun>, <fun>)
```

auxquelles on fait appel de la façon suivante (les appels ci-dessous illustrent le polymorphisme) :

```
# max 2 3;;
- : int = 3
```

```
# min 'a' '\t';;
- : char = '\t'
```

```
# max "50" "100";;
- : string = "50"
```

et dont le caractère curryfié permet, par exemple de définir une fonction définie dans le domaine des flottants renvoyant x si $x \geq 0$ et 0 sinon :

```
# let f = max 0.;;
val f : float -> float = <fun>
# f 1.2;;
- : float = 1.2
```

```
# f (- 0.7);;
- : float = 0.
```

<u>Note</u> : dans le dernier exemple, les parenthèses sont nécessaires (pourquoi ? 😉)

Citons encore la version préfixe des opérateurs usuels :

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

```
# (+) 1 2 ;;
- : int = 3
```

permettant de définir par exemple une fonction linéaire :

```
#   let fois_2 = ( *.) 2. (*noter l'espace est nécessaire après "(" *);;
val fois_2 : float -> float = <fun>
#   fois_2 3.;;
- : float = 6.
```

2.3.d. Exemple d'utilisation de fonctions curryfiées

À titre d'exemples, on peut proposer d'utiliser les fonctions curryfiées pour définir facilement une suite de fonctions ou une famille de fonctions.

- Définition d'une suite de fonctions

Considérons la suite de fonctions $(f_n)_{n\in\mathbb{N}}$ de $\left(\mathbb{R}^{\mathbb{R}}\right)^{\mathbb{N}}$ définie par $f_n : x \mapsto \frac{1}{1+x^{2n}}$. Une implémentation possible de cette suite de fonctions est la suivante :

```
# let f n x = 1. /. (1. +. x ** (2. * float_of_int n));;
val f : int -> float -> float = <fun>
```

Cette définition étant posée, une implémentation de la fonction f_2, par exemple, serait `f2` définie par :

```
# let f1 = f 1;;
val f1 : float -> float = <fun>
```

```
# f1 3.;;
- : float = 0.1
```

où `f1` implémente la fonction $f_1 : x \mapsto \frac{1}{1+x^2}$, qui vérifie $f_1(3) = \frac{1}{1+9} = 0{,}1$.

- Définition d'une famille de fonctions

Considérons la famille des fonctions exponentielles de base a $(\exp_a)_{a\in\mathbb{R}}$ de $\left(\mathbb{R}^{\mathbb{R}}\right)^{\mathbb{R}}$ définie par

$$\exp_a : x \mapsto e^{ax}.$$

Une implémentation possible de cette famille de fonctions est la suivante :

```
# let fexp a x = exp (a *. x);;
val fexp : float -> float -> float = <fun>
```

Cette définition étant posée, une implémentation de la fonction $\exp_{1/2}$, par exemple, serait `exp_demi` définie par :

```
# let exp_demi = fexp 0.5;;
val exp_demi : float -> float = <fun>
```

où `exp_demi` implémente la fonction $x \mapsto e^{\frac{1}{2}x}$.

3 Filtrage [première présentation]

Le filtrage est un *trait* puissant du langage **Caml** qui permet d'associer au(x) paramètre(s) d'une fonction des expressions différentes suivant les valeurs prises par les paramètres au moment de l'appel.
Le filtrage fonctionne comme un enchaînement fini (séquence) de branchements conditionnels

« **if … then … else (if … then … else (if … then … else (…)))** »,

où les conditions testées par les clauses *if* portent sur la valeur d'une même expression.
Nous verrons ultérieurement que plus généralement le filtrage porte sur la forme de l'entrée testée : on dira que l'on procède à un **filtrage de motifs** (*pattern-matching*).

L'utilisation naturelle du filtrage consiste à l'appliquer sur la valeur prise par les paramètres d'une fonction (ou par une expression dépendant des paramètres d'une fonction), mais on peut aussi l'utiliser pour appliquer un traitement différent selon la valeur prise par une expression, en dehors de la définition d'une fonction.

3.1 Mise en œuvre d'un filtrage sur les valeurs pour les types de base

3.1.a. Filtrage implicite sur la valeur de l'entrée d'une fonction

La forme élémentaire de définition d'une fonction (à un paramètre) par filtrage est la suivante :

```
function | <motif 1>  -> <expression 1>
         | <motif 2>  -> <expression 2>
         …
         | <motif n>  -> <expression n>
```

On parle alors de **filtrage implicite** (l'entrée sur laquelle porte le filtrage n'est pas nommée).

Dans le cas où le type de l'entrée est un type de base (`unit`, `int`, `float`, `bool`, `char`, `string`) les motifs ont la forme d'une valeur (décrite par un *littéral*) ou d'un nom :

- **si le motif est une valeur**, l'expression associée est utilisée (évaluée) si l'entrée prend cette valeur ;
- **si le motif est un nom**, l'expression associée est utilisée quelle que soit la valeur de l'entrée testée. (on dit que le motif « s'accorde » avec n'importe quelle valeur).
  Ce nom est un nom <u>local</u> dont la portée est limitée à l'évaluation de l'expression associée.

Il découle du ***typage fort*** appliqué par **Caml**, que **tous les motifs sont de même type** (le type de l'entrée de la fonction) et **toutes les expressions associées doivent être de même type** (le type de la sortie de la fonction).

On peut ainsi définir comme suit la fonction caractéristique d'un singleton sur le domaine des entiers, ici la fonction caractéristique, `f0`, du singleton $\{0\}$ ou une fonction, `f1`, qui associe à un entier 1 si cet entier est nul et son opposé sinon :

```
# let f0 = function
               | 0 -> 1
               | x -> 0;;
```

```
# let f1 = function
               | 0 -> 1
               | x -> - x;;
```

Si le motif est un nom, et que ce nom n'est pas utilisé, on peut le remplacer par le caractère joker (*wildcard*) « _ » (on dit encore que « _ » est une **variable anonyme**).
Ainsi, on peut aussi définir `f0` comme suit :

```
# let f0 = function
              0 -> 1 (* note : la barre | pour le premier cas est facultative *)
            | _ -> 0;;
```

On pourra noter qu'il existe un ***abrégé*** permettant d'associer une même expression à plusieurs motifs, ainsi pour définir la fonction caractéristique de la partie $\{0, 1\}$ sur le domaine des entiers, on pourra écrire indifféremment :

```
# let f12 = function
                | 0 -> 1
                | 1 -> 1
                | x -> 0;;
```

ou

```
# let f12 = function
                | 0 | 1 -> 1
                | x -> 0;;
```

Il est essentiel de ne pas oublier que dans un filtrage, les motifs sont testés dans l'ordre où ils sont écrits et donc, **l'expression évaluée est celle qui correspond, dans la séquence des motifs, au <u>premier</u> motif qui s'accorde avec la valeur de l'entrée**.
Ainsi si l'on change l'ordre des motifs dans la définition de `f0` ci-dessus, le motif `x` s'accordant avec toutes les valeurs de l'entrée, le motif `0` ne sera jamais sélectionné, ce que l'interpréteur signale :

```
# let f0 = function
               | x -> 0
               | 0 -> 1;;
val f0 : int -> int = <fun>
File "[7]", line 3, characters 15-16:
Warning 11: this match case is unused.
```

et pour toute valeur entière en entrée, la fonction `f0` renverra `0` :

```
# f0 0;;
- : int = 0
```

3.1.b. Filtrage explicite

Le **filtrage explicite** utilise la syntaxe `match <expression> with` où la valeur de l'expression `<expression>` (souvent juste un nom d'expression ou de valeur) est comparée aux motifs de filtrage. Avec cette syntaxe, on peut effectuer un filtrage sur la valeur d'une variable ou d'une expression, en dehors d'un contexte de définition de fonction :

```
# match a with
      | 0 -> 1
      | x -> 0;;
```

```
# match a mod 3 with
      | 0 -> 1
      | x -> 0;;
```

La construction
`match <expression> with | motif₁ -> expression₁ | motif₂ -> expression₂ … | motifₙ -> expressionₙ`
définit une expression, qui, dans le premier exemple, prend la valeur 1 si `a` est l'entier nul et zéro sinon, et, dans le second exemple, prend la valeur 1 si `a` est un multiple de 3 et zéro sinon.

On peut utiliser cette construction de la même façon que n'importe quelle autre expression, en particulier dans la définition d'une variable ou d'une fonction :

```
# let b = match a with
              | 0 -> 1
              | x -> 0;;
```

```
# let f x = match x with
                | 0 -> 1
                | x -> 0;;
```

Note : pour la définition de `b`, il faudra bien sûr que la variable `a` ait précédemment été définie.

3.2 Motifs gardés

Il est possible d'adjoindre au motif une condition sous la forme d'une expression booléenne, la syntaxe est d'un cas de filtrage est alors

`| <motif> when <condition> -> <expression>`

Le résultat du filtrage sera alors l'évaluation de `<expression>` si l'entrée correspond au motif et si la condition `<condition>` est vérifiée (évaluée à `true`). On doit garder à l'esprit que **l'adéquation de l'entrée du filtrage avec la séquence des motifs de filtrage, est testée dans l'ordre dans lequel les motifs (gardés ou non) sont énumérés**. Il est donc essentiel de se poser la question de l'ordre dans lequel sont énumérés les motifs lors de la mise en place d'un filtrage.

Voici comment on pourra implémenter une fonction affine par morceaux sur le domaine des flottants :

$$f : \mathbb{R} \to \mathbb{R}, \quad x \mapsto \begin{cases} x+1 & \text{si } x < -1 \\ 0 & \text{si } x \in [-1\,;1] \\ x-1 & \text{sinon} \end{cases}$$

```
# let f x = match x with
      | a when a <= -1. -> a +. 1.
      | a when a <= 1.  -> 0.
      | a               -> a -. 1.;;
```

On a utilisé ici le nom local a qui constitue le motif (et s'accorde à toutes les entrées).
On aurait pu aussi utiliser pour nom local le nom `x` (ce nom identique à celui du paramètre en entrée de la fonction) aurait pris l'ascendant, localement sur le nom `x` de l'entrée) (fonction `fv1`), ou bien utiliser le nom de l'entrée dans la condition et l'expression à renvoyer (fonction `fv2`) :

```
# let fv1 x = match x with
      | x when x <= -1. -> x +. 1.
      | x when x <=  1. -> 0.
      | x               -> x -. 1.;;
```

```
# let fv2 x = match x with
      | _ when x <= -1. -> x +. 1.
      | _ when x <=  1. -> 0.
      | _               -> x -. 1.;;
```

3.3 Filtrage sur les n-uplets

Le filtrage sur les valeurs de l'entrée d'une fonction ou sur la valeur prise par une expression, vu jusque là pour des valeurs de l'un des types de base, **se généralise à tous les types d'objets**.
Il importera d'ailleurs, pour chaque nouveau type d'objet que nous rencontrerons, de bien cerner tous les filtrages pertinents qu'il est possible de mettre en place.

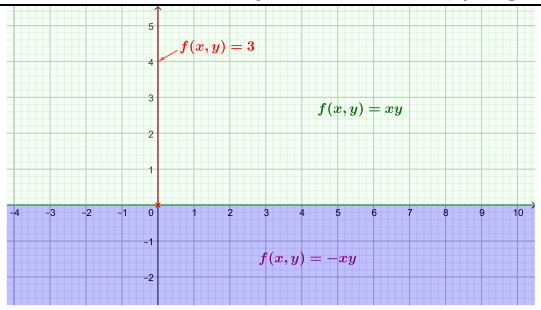
Ici, il s'agit d'examiner comment il est possible de mettre en place des filtrages sur un n-uplet, *i.e.* sur un objet dont le type est un type produit.

On notera préliminairement que la mise en place est identique sur un n-uplet ou pour une fonction curryfiée de n variables, et que toutes les remarques précédentes (ordre dans l'examen des motifs, portée des noms, cas inutilisés) restent valables.
L'élément nouveau est la possibilité de **nommer chaque composante du n-uplet**, par ailleurs :

- les composantes du n-uplet peuvent indifféremment être nommées au niveau du motif du filtrage ou en amont ;
- le filtrage se met en place de la même façon pour une fonction de plusieurs variables curryfiée ou non (dans le cas où le n-uplet) est nommé[(*)].

Ainsi pour cette fonction définie sur les couples d'entiers à l'aide d'une disjonction de cas (*cf.* figure) :

$$f : \begin{array}{ccc} \mathbb{Z}^2 & \to & \mathbb{R} \\ (x,y) & \mapsto & \begin{cases} 3 & \text{si} \quad x = 0 \text{ et } y \geq 0 \\ -y & \text{si} \quad y < 0 \\ xy & \text{sinon} \end{cases} \end{array}$$



les trois implémentations suivantes (par exemple) sont licites.
La première correspondant à une transcription littérale de la définition mathématique :

```
let f = function | x, y -> 3 when x = 0 && y >= 0 | _, y -> - y when y < 0 | x, y -> x*y;;
```

Les deux suivantes utilisant l'ordre d'examen des motifs, dans une version curryfiée et l'autre non :

```
let f x y = match x, y with
            | _  when y < 0 -> - x * y
            | 0, _          -> 3
            | _             -> x * y;;
```

```
let f v = match v with
            | x, y  when y < 0 -> - x * y
            | 0, _             -> 3
            | x, y             -> x * y;;
```

(*) : l'implémentation est la même dans la deuxième proposition si l'on remplace `let f x y` (curryfiée) par `let f (x, y)` (non curryfiée).

3.4 Exhaustivité du filtrage et « bonnes pratiques »

On a vu que **Caml** a la capacité de détecter si un filtrage comporte un cas inutilisé (**3.1.a.**)
Caml a aussi la capacité de détecter si un filtrage est **exhaustif**, à savoir si pour toute valeur possible de l'entrée, une sortie est définie - en vérité, il serait plus précis d'écrire « si toutes les formes possibles de l'entrée » sont couvertes par les motifs de filtrage.

3.4.a. Filtrage exhaustif sur les types de base

L'analyse menée par l'interpréteur ou le compilateur, détecte si toutes les valeurs possibles pour le type de valeurs en entrée sont couvertes.
Ainsi, pour un type pour lequel il n'existe que deux valeurs, il est détecté si les deux valeurs sont traitées.
Sur l'exemple-jouet suivant :

```
# let f a = match a with
              | true -> false;;
File "[13]", line 1, characters 10-52:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
false
```

Aucune expression n'est associée à une entrée a qui vaudrait `true`, ce que signale l'interpréteur.
On notera que cependant, aucune exception n'est levée, et que l'exécution du script ou la compilation du programme ne seront pas interrompues ; ce n'est qu'un avertissement qui est émis.

Remarque : si le type en entrée ne prend qu'une valeur, le filtrage ne peut qu'être exhaustif, ce qui est le cas pour le type `unit`, ainsi un filtrage sur le type `unit` ne peut qu'être exhaustif.
On peut donc indifféremment définir une fonction constante sur le type `unit` des deux façons suivantes (avec et sans filtrage, respectivement) :

```
# let f = function (x : unit) -> 0;;
```

```
# let f = function | () -> 0;;
```

Dans tous les cas, et en particulier si le type de l'entrée comporte un grand nombre de valeurs possibles, il est possible de couvrir toutes les valeurs possibles, et donc d'**assurer l'exhaustivité du filtrage**, **en utilisant le caractère joker « _ »** ou un nom (les deux s'accordent avec toutes les valeurs) comme dernier motif de filtrage.
Voici donc trois exemples de filtrage exhaustif :

```
let f x =
    match x with
        | true  -> 1
        | false -> 0;;
```

```
let f x =
    match x with
        | true -> 1
        | _    -> 0;;
```

```
let f x =
    match x with
      | 0 | 1-> 1
      | x    -> x + 1;;
```

3.4.b. Filtrage exhaustif avec des motifs gardés

La portée de l'analyse menée par l'interpréteur ou le compilateur, se limite à la forme de l'entrée, aussi, dans le cas de motifs gardés, il est nécessaire, pour que le filtrage considéré comme exhaustif, d'imposer un dernier cas (utilisant un nom ou un « _ ») prenant en charge « tous les cas » restant possiblement « à traiter ».
Ainsi, dans les deux exemples suivants, le filtrage est détecté comme non exhaustif, alors que pourtant ils le sont :

```
# let f x = match x with
              | 0 -> 0
              | _ when true -> 1;;
File "[25]", line 1, characters 10-78:
Warning 8: this pattern-matching is not
exhaustive.
Here is an example of a case that is not
matched:
1
(However, some guarded clause may match this
value.)
```

```
let f x = match x with
              | x when x > 0. -> 1.
              | x when x = 0. -> 0.
              | x when x < 0. -> -1.;;

File "[27]", line 1, characters 10-131:
Warning 8: this pattern-matching is not
exhaustive.
All clauses in this pattern-matching
are guarded.
```

3.4.c. Filtrage exhaustif sur des *n*-uplets

Dans le cas où l'entrée a la forme d'un n-uplet ou dans le cas d'un filtrage explicite sur les entrées d'une fonction curryfiée, l'analyse du filtrage est menée selon les mêmes modalités, mais en prenant en compte toutes des combinaisons possibles de valeurs des n entrées, qui doivent toutes être couvertes.

```
let f (x, y) z = match x, y, z with
    | 0, _, _ when y < 0 -> 0
    | _, 0, _            -> 1
    | _, _, _ when z = 0 -> 3
    | _, _, _            -> 4;;
```

```
let f = function
    | true,  true  -> true
    | true,  false -> true
    | false, true  -> false
    | false, true  -> true;;
```

3.4.d. <u>Bonne pratique</u> : toujours assurer l'exhaustivité du filtrage

Si on met en place un filtrage non exhaustif, cela affectera l'exécution d'un programme et pourra causer des erreurs lors de son utilisation : il y aura possiblement des entrées licites sur lesquelles le programme « plantera ». Il faut donc s'assurer de l'exhaustivité de chaque filtrage.
Cependant, même si les entrées ne sont pas destinées à prendre les valeurs non traitées dans le filtrage, c'est une **bonne pratique,** de ***programmation défensive***, de **ne mettre en place <u>que</u> des filtrages exhaustifs**.
Lorsque les cas non traités du filtrage sont correspondent à des valeurs impropres pour l'entrée, ce sera une bonne pratique de **lever une exception** si l'entrée prend l'une de ces valeurs interdites.
On peut lever (*raise*) une exception prédéfinie (avec le mot-clé `raise` suivi du nom de l'exception) ou une exception que l'on aura précédemment définie, ou spécifiquement l'exception prédéfinie `Failure` (avec le mot-clé `failwith` suivi d'une chaîne de caractères explicitant le problème rencontré). Cette dernière façon conviendra à nos besoins d'assurer des filtrages exhaustifs.
Par exemple, si on souhaite définir une fonction destinée à ne prendre en entrée que des entiers positifs :

```
# exception ValueError;;
# let f x = match x with
      | 0 -> raise ValueError
      | _ -> x + 1 ;;
```

```
# let f x = match x with
      | 0 -> failwith "l'entrée est nulle
      | x -> x + 1 ;;
```