Introduction au langage OCaml

IAP1

Catherine Dubois

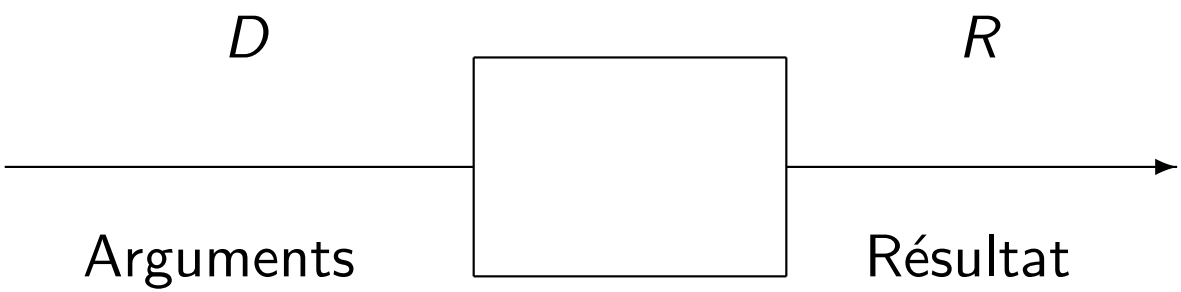
OCaml : un langage de la famille ML

• **un langage fonctionnel**

concepts essentiels : la définition de fonction et l'application de fonctions.

Comme LISP, SCHEME, etc.

notion de fonction : proche de celle des maths,



Le résultat ne dépend que des arguments : transparence référentielle.
Remarque : ce n'est plus vrai dès que l'on introduit des affectations.

• **proche des mathématiques : notation et esprit**
même notation qu'en maths pour désigner une fonction : une expression + des variables (comme paramètres ou arguments)

$N \longrightarrow N$
$n \longmapsto 2n + 3$

En OCaml `function x -> 2*x + 3`

En C, nom obligatoire + `return`

```
int F(int x)
 return (2 * x + 3);
```

Notions de domaine et co-domaine : *type*
La fonction précédente définie de N vers N est de type $int \longrightarrow int$.

• **Langage typé**
Comme la plupart des langages fonctionnels (Lisp, Scheme ne le sont pas)

- **Ordre supérieur**

les fonctions ont le même statut que les entiers, les booléens, les listes ...

$$\begin{array}{rl} \circ : & (A \Rightarrow B) \times (C \Rightarrow A) \rightarrow (C \Rightarrow B) \\ & (f, g) \mapsto (x \mapsto f(g\ x)) \end{array}$$

- **Aspects impératifs, objets**

On les oublie (pour l'instant) !

• **langage interactif** (boucle interactive, top level)

programme = suite de phrases terminées par **;;**

Une phrase est :

- une expression
- ou une définition (globale) **let idf = expression ;;**

Pour chaque phrase :
compilation (analyse lexico-syntaxique, vérification des types, génération de code)
+ édition de liens
+ exécution immédiate
+ affichage de la réponse de OCaml :
- : type = valeur ou *val idf : type = valeur*

```
% ocaml
 Objective Caml version 3.04

#1+2;;                    expression
 - : int = 3
l'expression que vous venez de taper est de type int
et sa valeur est 3
#"Hello" ^ "world";;
 - : string = "Helloworld"
#let x = 2+3;;    définition globale - Posons x = 2 + 3
val x : int = 5
 désormais l'idf x est connu, de type int,
 et lié à la valeur 5
#x+x;;
- : int = 10
#let f  x =  2*x + 3;;
val f : int -> int = <fun>
f  fonction définie des entiers vers les entiers
```

```
# f 4;;
- : int = 11
 application
#let g x = f ( f x);;
val g : int -> int = <fun>
définition d'une autre fonction
```

session = suite d'étapes *questions* / *réponses*
se termine par `ctrl D` ou `#quit;;`

```
##quit;;
```

(Attention, ici on tape le #)

Mais on peut aussi produire un exécutable en compilant un programme source

$\Rightarrow$ exécuter la commande `ocamlc -o monexe monprog.ml` puis lancer le programme `monexe` (voir en TP dans quelques séances)

Expressions, valeurs, types

- Une expression a une syntaxe bien définie (grammaire)

- Evaluer une expression : calculer sa valeur

Les règles qui décrivent comment évaluer une expression forment la sémantique du langage

- On classifie les valeurs du langage suivant leur type

- Seules les expressions syntaxiquement correctes et bien typées sont évaluables

(analogie avec les maths : $f(x)$ n'a de sens que si $x \in D_f$)

`2 + true` est syntaxiquement correct mais mal typé

- Toute expression a un type (c'est le type de sa valeur)

Un type donne une information sur la valeur d'une expression, sur la valeur que pourra prendre un argument de fonction (plus ou moins précis selon les formalismes , exemples : `int`, `int list`, *natural*, *pair*)

$\Rightarrow$ tout langage typé possède :

- un **langage de types** (qui décrit l'ensemble des types autorisés dans les programmes)
On donne des règles de formation : par exemple si t est un type alors t *list* est un type possible.
Un langage incorpore des types de base et des types composées.

- des **règles de typage** (qui précisent quelles sont les expressions bien typées)
Exemples de règle : si e_1 est une expression (bien typée) de type *int*, si e_2 est une expression de type *int* alors $e_1 + e_2$ est une expression de type *int* *true* est de type *bool*.

- un outil qui vérifie les types (ici inférence de types) **typeur** incorporé en général au compilateur

Types de base

- `int` : type des nombres entiers (compris entre -2^{30} et $2^{30}-1$)

Constantes ... -2, -1, 0, 1, 2 ...

Opérateurs utilisables avec le type `int` : +, -, *, /, mod

```
#2 + 3 * 4;;
- : int = 14
```

- `float` : type des nombres flottants

Constantes 1.0, 3.4, 5.1e10, 13.4e-3

Opérateurs avec le type `float` : +., -., *., /., sqrt, exp, asin, ...

```
#13.4e-3 ;;
- : float = 0.0134
#sqrt(56.3);;
- : float = 7.50333259292
```

- `bool` : type des booléens

Constantes `true`, `false` (il n'y en a pas d'autres)
valeurs de vérité, encore appelées *valeurs booléennes*

`true` est la valeur de toute proposition énonçant un fait vrai
par exemple *l'entier 2 est inférieur à 3*.

La valeur `false` est celle des énoncés faux,
par exemple de *2 est plus grand que 3*.

Opérateurs principaux : `&&` (la conjonction, *et* logique), `||` (la disjonction, *ou* logique), `not` (la négation, *non* logique).

Egalement des opérateurs de comparaison : $<$, $>$, $=$ s'appliquent sur des entiers, des flottants (et bien d'autres choses) et retournent des booléens

```
#2<3;;
- : bool = true
#2.5>3.6;;
- : bool = false
# 1 > 5.5;;
      ----
This expression has type float but is here used with type
#not false && false ;;
- : bool = false
#not (false && false);;
- : bool = true
#true || false;;
- : bool = true
# 8638 mod 7 =0;;
- : bool = true
```

- `char` : type des caractères

```
# 'a';;                          # ''';;
- : char = 'a'                   - : char = '''
# '\097';;                       # '\039';;
- : char = 'a'                   - : char = '''

# int_of_char 'a';;                (* int_of_char : char -> int*)
- : int = 97
# char_of_int (int_of_char 'a');; (* char_of_int : int -> c
- : char = 'a'
```

- `string` : type des chaînes de caractères

Constantes "ceci est une chaîne de caractères"

Opérateurs utilisables avec le type `string` : `^`, `String.length` + nombreuses fonctions définies dans le module String.

```
#"ceci est une chaîne de caractères";;
- : string = "ceci est une chaîne de caracteres"
#String.length ("ceci est une" ^ " chaîne");;
- : int = 19
#String.uppercase "ceci est une" ;;
- : string = "CECI EST UNE"
```

Définir = donner un nom à une valeur

Définition globale :

```
#let x = 1 ;;
 x : int = 1
#let y = true;;
y : bool = true
```

→ Syntaxe : `let idf = exp;;`

→ Typage : le type de `idf` est celui de `exp`

→ Sémantique : OCaml calcule la valeur *v* de `exp` et l'associe à `idf`

Définition locale :

```
#let b = true in b && (not b) ;;
 - : bool = false
#not b;;
Unbound value b
#(let x = 4 + 1 in x*x) + 10;;                    - : int = 35
#let a = 3;;                                      val a : int = 3
#let a = 10 in a+1;;                              - : int = 11
#a;;                                              - : int = 3
```

→ Syntaxe : `let idf = exp1 in exp2` **C'est une expression**

→ Typage : le type de `idf` est celui de `exp1`, cette info est valide uniquement dans `exp2`
Type de `let idf = exp1 in exp2` = type de `exp2`

→ Sémantique : OCaml calcule la valeur $v1$ de `exp1` et l'associe à `idf` le temps de calculer la valeur de `exp2`
Valeur de `let idf = exp1 in exp2` = valeur de `exp2`

Expression conditionnelle

→ Syntaxe : `if c then e1 else e2`

→ Typage :

- `c` expression de type `bool`
- `e1` et `e` expressions de même type `t`
- Le type de `if c then e1 else e2` est `t`

→ Sémantique

- Evaluer `c`, soit *v* sa valeur.
- Si *v*=`true` alors valeur de `if c then e1 else e2` = valeur de `e1`
- Si *v*=`false` alors valeur de `if c then e1 else e2` = valeur de `e2`

```
#let r = if 1<2 then 1.34 else "bonjour" ;;
This expression has type string but is here used with type

#("il est passé") ^ (if 1<2 then " par ici" else " par là")
 : string = "il est passé par ici"

#("il est passé") ^ (if 1>2 then " par ici" else " par là")
 : string = "il est passé par là"
```

Priorité dans les expressions

opérateur infixe : placé entre les sous-expressions qui le concernent.

Pour lever l'ambiguïté ⇒ introduire des parenthèses

Exemple : comment lire `2 + 3 * 4` ?
`(2 + 3) * 4` ? ou `2 + (3 * 4)` ?

En pratique, notion de priorité entre les opérateurs

*op*1 *est plus prioritaire que op*2 *s'il se sert avant op*2 *en ce qui concerne le choix de ses arguments*

Traditionnellement, − unaire (l'opposé) plus prioritaire que `*`
plus prioritaire que + et − binaire
`not` a une priorité plus élevée que `&&` et `||`

priorité étendue aux autres constructions du langage :
if . . . *then* . . . *else* . . . a une priorité très basse : `if true then 1 else 2 + 3` interprétée comme `if true then 1 else (2 + 3)`

On utilisera les parenthèses pour contrecarrer les priorités.

Fonctions

- **Définition de fonction anonyme**

```
# function a -> a - 2;;
- : int -> int = <fun>
```

→ Syntaxe `function idf -> expression`
→ Type d'une fonction : `t1 -> t2` avec `t1` type du paramètre et `t2` type du résultat
→ Sémantique : un morceau de code

Comment OCaml trouve-t-il le type d'une fonction ? Réponse : en analysant le texte de la fonction ... A SUIVRE

- **Nommer une fonction**

→ Syntaxe : deux formes

```
let f idf = expr ;;
```

ou

```
let f = function idf -> expr ;;
```

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

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

#let aire_cercle =
  let pi = 3.14 in function r -> pi *. r *. r;;
val  aire_cercle : float -> float = <fun>

#let triple y = let car x = x * x in (car y)*y;;
val  triple : int -> int = <fun>

#let f x = let g y = x + y in (g 2) + x;;
val  f : int -> int = <fun>
```

- **Fonctions à plusieurs arguments :**

```
# function (lo,la) -> lo*la;;
- : int*int -> int = <fun>
#let aire_rect (lo,la) = lo*la;;
val  aire_rect : int*int -> int = <fun>
```

Un n-uplet d'arguments au lieu d'un argument unique

→ Syntaxe `function (idf1, idf2, ... , idfn) -> expression`
→ Type : `t1 * t2 ...* tn -> t`
→ Sémantique : un morceau de code

Pour définir une fonction nommée :
→ Syntaxe : deux formes
`let f (idf1 , ... , idfn) = expr ;;`
ou
`let f = function (idf1, ..., idfn) -> expr ;;`

Toute fonction définie globalement a une interface

Elle sera mise en commentaires **avant** le texte de la fonction

pour la fonction `carré` :

```
(* Interface carré
type : int -> int
arguments : x le nombre à mettre au carré
précondition : aucune
postcondition : résultat = x*x
*)
```

Appliquer une fonction :

```
#carré(25);;
- : int = 625
#carré 25;;
- : int = 625
#carré true;;                                  erreur de type
This expression has type bool but is here used with type in
#carré (3+1);;
- : int = 16
#carré 3 + 4;;
- : int = 13
#aire_rect (3,4);;
- : int = 12
```

→ typage : si `f` est une fonction de type `t1 -> t2`, alors `f e` a le type `t2` si `e` a le type `t1`

→ Sémantique : Appel par valeur : on calcule d'abord la valeur des arguments puis on utilise le corps de la fonction

carré $(3 + 1) \Rightarrow$ carré$(4) \Rightarrow$ 4*4 $\Rightarrow$ 16

- **Portée statique (ou lexicale):**

```
#let x = "ici";;
val x : string = "ici"
#let f s = s  ^  x ;;
val  f : string -> string = <fun>
#f "Que vois-je? " ;;
 - : string = "Que vois-je?  ici"
#let x = "la bas";;
val x : string = "la bas"
#f "Que vois-je ?" ;;
 - : string = "Que vois-je ? ici"
```

$\longrightarrow$ pour les liaisons des identificateurs non arguments (libres), se reporter au moment de la définition de la fonction

• Fonctions récursives

fonction utilisée dans sa définition :

```
let rec f x = ..... (f ...) ...
```

```
#let rec fact n =
   if n = 0 then 1 else n * fact (n - 1) ;;
 val fact : int -> int = <fun>
#fact 10 ;;
 - : int = 3628800
# fact (-5);;
Stack overflow during evaluation (looping recursion?).
#let somme n =
   if n = 0 then 0 else n + somme (n - 1) ;;
Unbound value somme
#let rec somme n =
   if n = 0 then 0 else n + somme (n - 1) ;;
val somme : int -> int = <fun>
# somme 14;;
- : int = 105
```

Memento des fonctions récursives :

- Ne pas oublier le mot-clef `rec`

```
let fact_sans_rec n =
    if n = 0 then 1 else n * fact_sans_rec (n-1);;
Unbound value fact_sans_rec
```

- un ou plusieurs cas de base

- les cas généraux tendent vers les cas de base

On y reviendra ....

- **Restreindre le domaine de définition d'une fonction**

```
# let h x = if x = 1. then failwith "h : indefini"
            else 1. /. (x -. 1.);;
val h : float -> float = <fun>
#h 2.;;
- : float = 1.
#h 1.;;
Exception: Failure "h : indefini".
#let predecesseur y =
  if y <= 0
  then failwith "predecesseur : parametre non naturel"
  else y - 1;;
val predecesseur : int -> int = <fun>
#predecesseur (-5);;
Exception: Failure "predecesseur : parametre non naturel".
#aire_cercle (h 1.);;
Exception: Failure "h : indefini".
```

→ Syntaxe de l'expression d'échec : `failwith e`

→ Typage : `e` est une expression de type `string` et `failwith e` a n'importe quel type

→ Sémantique : **une expression qui échoue n'a pas de valeur**

Interface d'une fonction qui peut échouer

```
(*
Interface
h :  float -> float
arguments x
précondition : x différent de 1
postcondition : calcule la valeur 1/(x-1)
raises : Failure "h : indefini" (si x = 1)
*)
let h x = if x = 1.
          then failwith "h : indefini"
          else 1. /. (x -. 1.)
```

Types composés

- **Types list**

`t list` : type des listes finies d'éléments de type `t`

Exemples : `int list, string list, (int list) list` ...

Constantes : [v1 ; v2 ; ... vn], [] (liste vide)

Opérateurs utilisables sur les listes : `::`, `@`

```
#[1;2;3];;
 - : int list = [1; 2; 3]
#[1;true];;                          erreur de type
#1::2::[];;
 - : int list = [1; 2]               autre syntaxe pour 1::2::[]
#[ [true]; []; [true;false] ] ;;
- : bool list list = [ [true]; []; [true;false] ]
# [1;2] @ [3];;
- : int list = [1; 2; 3]
```

→ **Syntaxe** $e :: r$
→ **Typage** si e est de type t et si r est de type t *list* alors $e :: r$ est de type t *list*
→ **Sémantique** si la valeur de e est v et si la valeur de r est la liste $[v_1; v_2 \ldots v_n]$ alors la valeur de $e :: r$ est $[v; v_1; v_2 \ldots v_n]$

→ **Syntaxe** $l_1 @ l_2$
→ **Typage** si l_1 et l_2 sont de type t *list* alors $l_1 @ l_2$ est de type t *list*
→ **Sémantique** si l_1 a pour valeur $[v_1; v_2 \ldots v_n]$ et l_2 a pour valeur $[w_1; w_2 \ldots w_k]$ alors $l_1 @ l_2$ a la valeur $[v_1; v_2 \ldots v_n; w_1; w_2 \ldots w_k]$.

Les listes peuvent être paramètres ou résultats de fonction

- Fonctions qui calculent une liste : RAS

```
# let mettre_en_liste x = [x];;
val mettre_en_liste : 'a -> 'a list = <fun>
# mettre_en_liste 1;;
- : int list = [1]
```

- Fonctions qui prennent une liste en argument

On s'appuiera sur la remarque suivante :

Une liste est soit vide, soit de la forme element :: liste

fonction `tête` : qui retourne le premier élément d'une liste

Son interface :

```
(*
Interface
tête :  'a list -> 'a
arguments : l
précondition : l non vide
postcondition : retourne le premier élément de l
raises : Failure "tête : liste vide "
*)
```

Ecriture de sa définition :
2 cas :
- liste vide : ça n'a pas de sens
- liste non vide : elle est donc de la forme x::r, le résultat est x

⇒ construction de filtrage : `match ...with`

```
#let tête l = match l with
   []    -> failwith "tête : liste vide "
| e::r -> e;;      clause de filtrage : filtre -> expre
val tête : 'a list -> 'a = <fun>
ou
#let tête l = match l with
   []    -> failwith "tête : liste vide "
| e::_ -> e;;

#tête [];;

#tête [1; 2; 3];;

#tête ["Bonjour"; "Merci"; "Au revoir"];;

#tête [ [1; 2]; [3] ];;
```

```
#let reste l = match l with
| [] -> failwith "reste: liste vide"
| _::l' -> l';;
val 'a list -> 'a list

#reste [];;                          #reste [1; 2; 3];;

#let zéro_en_tête l = match l with
| x::_ -> x=0
| _ -> false;;
val zéro_en_tête int list -> bool = <fun>

#let deuxième l = match l with
| x1::x2::l' -> x2
| _ -> failwith "deuxième: liste de moins de deux éléments"
val 'a list -> 'a
```

Fonctions récursives manipulant des listes

```
#let rec zero_est_dans l =  match l with
   [] -> false
 | e::_ -> e=0 || zero_est_dans r;;
val zero_est_dans : int list -> bool = <fun>

#let rec lg l = match l with
                  [] -> 0
                | _::r -> 1 + (lg r);;
lg : 'a list -> int = <fun>
```

lg s'applique à n'importe quel type de liste.
Son résultat est toujours entier.

```
#[];;
- : 'a list = []          la liste vide est polymorphe
```

`lg` existe : c'est la fonction `length` du module `String`.

- **Expression de filtrage**

→ Syntaxe :

```
match e with
| f1 -> e1
| f2 -> e2
| ...
| fn -> en
```

`fi filtre (motif, pattern)`
`e, ei expressions`

Un filtre est soit la liste vide [], soit un identificateur, soit _, soit filtre::filtre (pour l'instant)

filtre = schéma de valeur, patron, forme de valeur

→ Typage :

- `e, f1, f2 ... fn` expressions de même type
- `e1, e2, .. en` expressions de même type `t`
- le type de `match ... with` est `t`

→ Sémantique :

- On calcule la valeur de `e`, soit *v* cette valeur
- Si `f1` *filtre v*, alors valeur de `match ... with` = valeur de `e1`
- Si `f1` *ne filtre pas v* mais `f2` *filtre v*, alors
  valeur de `match ... with` = valeur de `e2` ... etc

filtrer =
tester si la valeur est de la bonne forme
+
nommer certaines parties de la valeur
(les liaisons sont utilisables derrière ->)

```
#let g l = match l with
  [] -> "cas1"
  cas de la liste vide
| x::_::r -> "cas2"
  cas des listes non vides de lg >= 2
| _::_ -> "cas3"
  cas des listes singleton ;;
```

Filtrage complet (tous les cas possibles pour la valeur filtrée)

Si ce n'est pas le cas : Ocaml le dit (warning, ce n'est pas une erreur)

```
let non_exhaustif l = match l with
  [] -> "cas1"
| _::_::_ -> "cas2";;

let non_exhaustif l = match l with
  [] -> "cas1"
| _::_::_ -> "cas2" ;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1::[]
val non_exhaustif : 'a list -> string = <fun>
```

Ocaml peut aussi vous signaler un cas redondant :

```
let pas_correcte l = match l with
  [] -> false
| x::r -> true
| _::_::r -> true
Warning: this match case is unused.
```

Remarque : les identificateurs qui apparaissent dans les filtres sont des noms *nouveaux*

```
(* interface appartient
type : 'a * 'a list -> bool
arg e, l
pre aucune
post retourne true si e est élément de l, false sinon
*)
let rec appartient (e,l) = match l with
  [] -> false
| e::r -> true
| x::r -> appartient (e,r);;
```

Remarque : les identificateurs qui apparaissent dans les filtres sont des noms *nouveaux*

```
(* interface appartient
type : 'a * 'a list -> bool
arg e, l
pre aucune
post retourne true si e est élément de l, false sinon
*)
let rec appartient (e,l) = match l with
   [] -> false
| e::r -> true
| x::r -> appartient (e,r);;
Warning: this match case is unused.
val appartient : 'a * 'b list -> bool = <fun>
```

En fait cette fonction teste si ..... une liste est vide !!!!

Inadéquation entre implémentation et postcondition !

La solution :

```
(* interface appartient
type : 'a * 'a list -> bool
arg e, l
pre aucune
post retourne true si e est élément de l, false sinon
*)
let rec appartient (e,l) = match l with
  [] -> false
| x::r -> e=x || appartient (e,r);;
val appartient : 'a * 'a list -> bool = <fun>
```

• Types Produits Cartésiens

Règle de formation : si t_1, t_2, ... t_n ($n \geq 2$) sont des types alors
$t_1 * t_2 * ...t_n$ est un nouveau type (type produit)
= type des n-uplets $(v_1, v_2, ..., v_n)$ où v_i est une valeur de type t_i.

```
#(3,true);;
- : int * bool = 3, true
#(5, ("toto", true), false);;
- : int * (string * bool) * bool = 5, ("toto", true), false
#fst (1,2);;                    uniquement sur des couples
 - : int = 1
# fst (1,2,3);;
This expression has type int * int * int but
is here used with type 'a * 'b
#snd (1,2);;                    uniquement sur des couples
 - : int = 2
#(2+4, true && false);;
- : int * bool = 6, false
```

→ **Syntaxe** :
Une expression n-uplet est de la forme :

$$(expr_1, expr_2, \ldots expr_n)$$

où $n > 1$ et $expr_1$, $expr_2$, $\ldots expr_n$ sont des expressions quelconques

→ **Typage** :
Le type de l'expression $(expr_1, expr_2, \ldots expr_n)$ est $t_1 * t_2 * \ldots * t_n$ si t_i est le type de $expr_i$, pour tout i

→ **Sémantique** :
La valeur de $(expr_1, expr_2, \ldots expr_n)$ est $(v_1, v_2, \ldots, v_n)$ si v_i est la valeur de $expr_i$, pour tout i

```
# let pred_succ x = (x-1,x+1);;
val pred_succ : int -> int * int = <fun>
# pred_succ 7;;
- : int * int = (6, 8)
```

Travaillons maintenant avec les vecteurs de $\mathbb{R}^2$. On peut représenter un tel vecteur par ses coordonnées dans une base fixée, soit une paire de nombres flottants (*float* $*$ *float*)

```
# let axe_des_x n = (n, 0.0);;
val axe_des_x : 'a -> 'a * float = <fun>
# axe_des_x 1.5;;
- : float * float = (1.5, 0.)
# let carré_norme v = match v with
  | (x,y) -> x *. x +. y *. y;;
  val carré_norme : float * float -> float = <fun>
# carré_norme (2.0, 3.0);;
- : float = 13.
```

Autre écriture : abbréviation du `match...with`

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

# let prod_scal ((x,y),(z,t)) = x *. z +. y *. t;;
val prod_scal : (float * float) * (float * float) -> float
```

Le paramètre de la fonction n'est pas nommé.

Première projection :

```
# let proj1 (x,y) = x;;
val proj1 : 'a * 'b -> 'a = <fun>

# let u = (1.0, 2.5) in proj1 u;;
- : float = 1
```

C'est une fonction *polymorphe* i.e. qui peut s'appliquer avec des valeurs de différents types : présence de *variables de type* dans le type, notées en Ocaml par `'a 'b 'c` ...

```
# proj1 (2,3);;
- : int = 2

# proj1 ((3,4), 5);;
- : int * int = (3, 4)

# proj1 ("un texte", true);;
- : string = "un texte"

# proj1 (proj1 ((3,4), 5));;
- : int = 3
```

La deuxième projection : de la même façon :

```
# let proj2 (x, y) = y;;
val proj2 : 'a * 'b -> 'b = <fun>
```

Construction de filtrage étendue pour *filtrer* des couples :

nouvelle sorte de filtre : (*filtre*, *filtre*)

Tous ces filtres se mélangent :

```
(x,y)::[]        (y,_)::_        ([],x::r)
```

```
#let f a = match a with
| ([], _) -> "cas 1"
| (_, []) -> "cas 2"
| (x,y) ->  "cas 3" ;;
val f : 'a list * 'a list -> string = <fun>

# f ([4;5],[5]);;

# f ([1],[]);;

# f ([],[]);;
```

```
#let d a = match a with
| (x, y)::_ -> x=y
| _ -> failwith "diag_tete : indéfini";;
val d : ('a * 'a) list -> bool = <fun>


# d [(true, 5)];;

# d [(4,3);(4,4)];;

# d [(1,1)];;

# d [(5,5]; (5, 6));;
```

Fonction récursive retournant une paire :

`div` calcule simultanément le quotient et le reste dans une division entière

⇒ pour illustrer comment manipuler le résultat d'un appel récursif lorsque celui-ci est une paire

```
(* Interface div :
type int*int -> int*int
arguments a : nombre entier à diviser
          b : diviseur
precondition b/=0
postcondition div(a,b)=(q,r)
              tels que a = b*q + r et 0<=r<b
raises : Failure "diviseur nul" si b=0
*)
```

```
let rec div (a,b)=
  if b=0 then failwith "diviseur nul" else
  if a<b then (0,a) else
  match div ((a-b),b) with
  | (q,r) -> (q+1,r);;     filtrage avec des couples
```

ou avec `let` *destructurant*:

```
let rec div (a,b) =
  if b=0 then failwith "diviseur nul" else
  if a<b then (0,a) else
  let (q,r) = div ((a-b), b) in (q+1,r);;
```

Pour suivre les calculs récursifs :

```
# #trace div;;
div is now traced.
# div (15,4);;
div <-- (15, 4)
div <-- (11, 4)
div <-- (7, 4)
div <-- (3, 4)
div --> (0, 3)
div --> (1, 3)
div --> (2, 3)
div --> (3, 3)
- : int * int = (3, 3)
```

Fonctions polymorphes

```
#let identite = function x -> x ;;
identite : 'a -> 'a = <fun>
 identite a le type t -> t pour tout type t
 elle accepte n'importe quel argument,
 le résultat a le même type que l'argument
#identite 3;;
- : int = 3
#identite [1;3];;
- : int list = [1; 3]
#(identite true, identite "toto");;
- : bool * string = true, "toto"
#let f_constante = function x -> 1 ;;
 f_constante : 'a -> int = <fun>
#f_constante (f_constante true);;
- : int = 1
```