ManO (MU4IN511) : COURS 1

Mise à Niveau et Ouverture (STL)

Emmanuel Chailloux

Noyau fonctionnel

Programmes en Objective Caml (ou F#)

Seuls les programmes **correctement typés** peuvent être exécutés ! ! !

un programme est :

- le calcul d'une expression
- c'est-à-dire l'application d'une fonction à ses arguments
- avec évaluation immédiate des arguments
- et rupture d'évaluation si calcul impossible

Types de base, valeurs et fonctions

- nombres
  - *int* ($[-2^{30}, 2^{30}-1]$ sur 32 bits ou $[-2^{62}, 2^{62}-1$ sur 64 bits) voir `min_int` et `max_int`
  - *float* (IEEE 754) mantisse 53bits, exposant $\in [-1022, 1023]$
- caractères : *char*
- chaînes de caractères : *string*
- booléens : *bool*

Opérations sur les nombres

entiers		flottants	
`+`	addition	`+.`	addition
`-`	soustraction	`-.`	soustraction
`*`	multiplication	`*.`	multiplication
`/`	division entière	`/.`	division
`mod`	modulo	`**`	exponentiation

opérateurs infixes ! ! !

```
1 # 1 + 3;;
2 - : int = 4
3 # 1.8 *. 9.1;;
4 - : float = 16.38
```

Fonctions sur les nombres

`ceil`		`cos`	cosinus
`floor`		`sin`	sinus
`sqrt`	racine carrée	`tan`	tangente
`exp`	exponentielle	`acos`	arccosinus
`log`	log népérien	`asin`	arcsinus
`log10`	log en base 10	`atan`	arctangente

angles en radiant ! ! !

Calculs sur les nombres (1)

```
1  # 1 + 2;;
2  - : int = 3
3
4  # 9/2;;
5  - : int = 4
6
7  # max_int + 1;;
8  - : int = -4611686018427387904
9
10 # 9.1 /. 2.2;;
11 - : float = 4.13636363636
12
13 # 1. /. 0.;;
14 - : float =          Inf
```

Calculs sur les nombres (2)

```
# 2 + 2.1;;
This expression has type float but
is here used with type int
```

Un *int* ne peut pas remplacer un *float* ! ! !

```
# sin;;
- : float -> float = <fun>

# asin 1.;;
- : float = 1.57079632679
```

Calculs sur les chaînes

```
# 'B';;
- : char = 'B'

# int_of_char 'B';;
- : int = 66

# "est une chaine";;
- : string = "est une chaine"

# (string_of_int 1987)^" est l'annee de CAML";;
- : string = "1987 est l'annee de CAML"
```

Opérations sur les booléns

Type des constantes et opérateurs:

```
true  : bool
false : bool
not   : bool -> bool
&&    : bool -> bool -> bool
||    : bool -> bool -> bool
```

si *e*1 et *e*2 sont des expressions booléennes alors `true`, `false`, `not e1`), `(e1 || e2)`, `(e1 && e2)` sont des expressions booléennes et réciproquement.

```
1 # true && false ;;
2 - : bool = false
3 # not (true && false) ;;
4 - : bool = true
```

Relations d'égalité et d'inégalité

=	égalité structurelle	<	inférieur
==	égalité physique	>	supérieur
<>	négation de =	<=	inférieur ou égal
!=	négation de ==	>=	supérieur ou égal

égalités et inégalités sont génériques (polymorphes)
elles s'appliquent à des arguments de n'importe quel type (les deux arguments doivent être du même type) ! ! !

```
= : 'a -> 'a -> bool
== : 'a -> 'a -> bool
```

```
1  # 3 == (1 + 2);;
2    - : bool = true
3  # not(true) = false ;;
4  - : bool = true
```

Produits cartésiens, n-uplets

- constructeur de valeurs : `,`
- constructeur de types : `*`
- accesseurs (couples) : `fst` et `snd`

```
1  # (28, "janvier");;
2  - : int * string
3  # (20, "janvier", 2020);;
4  - : int * string * int
5  # fst (20, "janvier");;
6  - : int = 20
7  # snd (20, "janvier", 2020);;
```

erreur de typage sur la dernière ligne : la fonction `snd` est appliquée à un triplet et non pas à un couple.

Listes homogènes

- liste vide : `[]`
- constructeur `::`
- type *list*
- accesseurs `List.hd` et `List.tl`

```
# [] ;;
- : 'a list
# 1::2::3::[] ;;
- : int list
# [1; 2; 3;] ;;
- : int list
# [1; "hello"; 3] ;;
erreur de typage
# List.hd [1.1; 1.2; 1.3] ;;
- : float = 1.1
# List.hd [] ;;
Exception: Failure "hd".
```

exception non récupérée

Expression conditionnelle

Syntaxe:

if expr1 **then** expr2 **else** expr3

- *expr1* de type *bool*
- *expr2* et *expr3* de même type

```
# if 3 = 4 then 0 else 4;;
- : int = 4
# if 5 = 6 then 1 else "Non";;
- : erreur de typage
# (if 3 = 5 then 8 else 10) + 5;;
- : int = 15
# 5 = 6 || 7 = 9;;
- : bool = false
```

Déclarations de valeurs (1)

Déclarations globales:

Syntaxe:

let p = e

```
1  # let pi = 3.14159;;
2  val pi : float = 3.14159
3
4  # sin (pi /. 2.0);;
5  - : float = 0.999999999999
```

Déclarations de valeurs (2)

Déclatations locales:

Syntaxe:

let p = e1 **in** e2

```
# let x = 3 in
    let b = x < 10 in
      if b then 0 else 10;;
- : int = 0

# b;;
Unbound value b
```

Déclarations combinées

Syntaxe:

let $p_1 = e_1$ **and** $p_2 = e_2$ ... **and** $p_n = e_n$; ;

```
# let x = 1 and y = 2;;
val x : int = 1
val y : int = 2
# x + y;;
- : int = 3
```

Syntaxe:

let $p_1 = e_1$ **and** $p_2 = e_2$ ... **and** $p_n = e_n$ **in** expr ; ;

```
# let a = 3.0 and b = 4.0 in sqrt(a*.a+.b*.b);;
- : float = 5
```

Valeurs fonctionnelles, fonctions

Syntaxe:

function p -> e

- fonction anonyme : **function** p -> e
- de type : typede(p) -> typede(e)

```
# function x -> x + 1;;
 - : int -> int
# (function x -> x + 1) 2019 ;;
- : int = 2020
# function x -> if x < 0 then -x else x;;
- : int -> int
# function x -> (function y -> 2*x + 3*y);;
- : int -> int -> int
```

Syntaxe:

fun p1 ... pn -> e ≡

function p1 -> ... -> **function** pn -> e

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

Déclaration de valeurs fonctionnelles

```
# let succ = function x -> x + 1;;
succ : int -> int
# succ 420;;
- : int = 421
```

Syntaxe:

let f = **function** p1 -> ... -> **function** pn -> e *ou*

let f p1 ... pn = e *ou* **let** f = **fun** p1 ... pn = e

```
# let pred(x) = x -1;;
pred : int -> int = <fun>
# let f c = 2*(fst c) + 3*(snd c);;
f : int * int -> int = <fun>
# f(1,2);;
- : int = 8
# let g = function x -> (function y -> 2*x + 3*y);;
val g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
# let h = g 1 ;;
val h : int -> int = <fun>
# h 2 ;;
- : int = 8
```

Explicitation des types des paramètres des fonctions

Coercicion de types:

Syntaxe:

(e : t)

let p : t = e

```
# let u = [];;
val u : 'a list = []
# let v = (u : int list);;
val v : int list = []
```

C'est principalement utilisé pour les fonctions, on peut écrire

```
let xor (x:bool) (y:bool) : bool = (x || y) && (not (x && y))
```

indiquant ainsi que le paramètre x est de type `bool`, le paramètre y de type `bool`, et la fonction retourne un booléen.

Portée des déclarations

- portée lexicale (liaison statique)

```
1  # let p = q + 1;;                    (* erreur q inconnue *)
2
3  # let p = p + 1;;                    (* erreur p inconnue *)
4
5  # let p = 10;;                        p : int = 10
6
7  # let addp x = x + p;;                addp : int ->int = <fun>
8
9  # addp 8;;                           - : int = 18
10
11 # let p = 40;;                        p : int = 40
12
13 # addp 8;;                           - : int = 18
14
15 # p;;                                - : int = 40
```

Appels de fonction

```
1  let fois (a, b) = a * b ;;
2  let moins (a, b) = a - b ;;
3  let carre x = fois (x, x) ;;
4  let delta (a, b, c) = moins (carre b, fois (4, fois (a, c))) ;;
```

empilement dans le cadre d'appel des arguments et des adresses de retour pour chaque appel de fonction

```
1  #  #trace moins ;;
2  moins is now traced.
3  #  #trace fois ;;
4  fois is now traced.
5  # #trace carre ;;
6  carre is now traced.
7  #  #trace delta;;
8  delta is now traced.
```

dépilement des arguments empilés au retour de la fonction

```
1  # delta (2,5,2) ;;
2  delta <-- (2, 5, 2)
3  fois <-- (2, 2)
4  fois --> 4
5  fois <-- (4, 4)
6  fois --> 16
7  carre <-- 5
8  fois <-- (5, 5)
9  fois --> 25
10 carre --> 25
11 moins <-- (25, 16)
12 moins --> 9
13 delta --> 9
14 - : int = 9
```

Curryfication

Passage d'une fonction à plusieurs paramètres en une fonction à un paramètre qui retourne une nouvelle fonction sur le reste des paramètres, cela revient à écrire uniquement des fonctions à un paramètre,

exemple en OCaml : passage d'une fonction prenant un couple d'entiers $(x, y) \rightarrow 2x + 3y$ en une fonction prenant un paramètre entier x et retournant une fonction prenant un paramètre y et calculant $2x + 3y$: $x \rightarrow (y \rightarrow 2x + 3y)$

```
# let f (x,y) = 2*x + 3*y ;;
val f : int * int -> int = <fun>
# f (1,0) ;;
- : int = 2
# f (1,1) ;;
- : int = 5
# f (1,2) ;;
- : int = 8
# let g = function x ->
            (function y -> 2*x + 3*y);;
val g : int -> int -> int = <fun>
```

```
# let h = g 1 ;;
val h : int -> int = <fun>
# h 0 ;;
- : int = 2
# h 1 ;;
- : int = 5
# h 2 ;;
- : int = 8
```

Récursivité (1) : définitions

Syntaxe:

let rec p = *expr*

```
# let rec sigma x = if x = 0 then 0
                    else x + sigma(x-1);;
sigma : int -> int = <fun>

# sigma 10;;
 - : int = 55
```

récursion mutuelle:

```
# let rec odd x = if x = 0 then false else even(x-1)
  and even x = if x = 0 then true else odd(x-1);;
val odd : int -> bool = <fun>
val even : int -> bool = <fun>

# odd 27;;
- : bool = true
```

Récursivité (2) : trace et schéma d'évaluation

trace au toplevel

```
1  # #trace sigma;;
2  sigma is now traced.
3  # sigma 4 ;;
4  sigma <-- 4
5  sigma <-- 3
6  sigma <-- 2
7  sigma <-- 1
8  sigma <-- 0
9  sigma --> 0
10 sigma --> 1
11 sigma --> 3
12 sigma --> 6
13 sigma --> 10
14 - : int = 10
15 # #untrace sigma;;
16 sigma is no longer ↩
       traced.
17 # sigma 4;;
18 - : int = 10
```

schéma d'évaluation

$$
\begin{aligned}
(\text{sigma } 4) &= 4 + (\text{sigma } 3) \\
&= 4 + (3 + (\text{sigma } 2)) \\
&= 4 + (3 + (2 + (\text{sigma } 1))) \\
&= 4 + (3 + (2 + (1 + (\text{sigma } 0)))) \\
&= 4 + (3 + (2 + (1 + 0))) \\
&= 4 + (3 + (2 + 1)) \\
&= 4 + (3 + 3) \\
&= 4 + 6 \\
&= 10
\end{aligned}
$$

2 phases :

- la descente récursive : empilement des additions tant qu'il y a des appels à `sigma`
- remontée récursive : effectuer les additions jusqu'au résultat final

Récursivité (3) : terminale

```
# let rec sigma_aux (n,a) = if n = 0 then a else sigma_aux (n-1,a+n);;
val sigma_aux : int * int -> int = <fun>
# let sigma n = sigma_aux (n,0);;
val sigma : int -> int = <fun>
```

```
# #trace sigma_aux;;
sigma_aux is now traced.
# #trace sigma;;
sigma is now traced.
# sigma 4;;
sigma <-- 4
sigma_aux <-- (4, 0)
sigma_aux <-- (3, 4)
sigma_aux <-- (2, 7)
sigma_aux <-- (1, 9)
sigma_aux <-- (0, 10)
sigma_aux --> 10
sigma_aux --> 10
sigma_aux --> 10
sigma_aux --> 10
sigma_aux --> 10
sigma --> 10
- : int = 10
```

un appel récursif dans lequel la fonction n'exécute aucune instruction après l'appel est un appel récursif terminal.

$$\begin{aligned}(\text{sigma } 4) &= \text{sigma\_aux } (4,0)\\ &= \text{sigma\_aux}(3,4)\\ &= \text{sigma\_aux}(2,7)\\ &= \text{sigma\_aux}(1,9)\\ &= \text{sigma\_aux}(0,10)\\ &= 10\end{aligned}$$

dans un appel récursif terminal il est possible de ré-utiliser le cadre d'appel précédent, ce qui permet de ne pas comsommer de mémoire supplémentaire.

Récursivité (4) : définition locale

fonction auxilaire locale curryfiée.

```
# let sigma n =
    let rec aux n a =
      if n = 0 then a
      else aux (n-1) (a+n)
    in
      aux n 0 ;;
      val sigma : int -> int =↩
            <fun>

# #trace sigma;;
      sigma is now traced.

# sigma 4 ;;
sigma <-- 4
sigma --> 10
- : int = 10
```

pas de trace de fonctions locales

schéma d'évaluation de `(sigma 4)` :

$$\begin{aligned}(\text{sigma } 4) &= \text{aux } 4\ 0\\ &= \text{aux } 3\ 4\\ &= \text{aux } 2\ 7\\ &= \text{aux } 1\ 9\\ &= \text{aux } 0\ 10\\ &= 10\end{aligned}$$

le premier cadre d'appel de aux est réutilisé dans les autres appels à aux. Ces appels récursifs sont en taille mémoire constante.

Récursivité (5) : définition locale

Une fonction locale peut utiliser l'ensemble de l'environnement accessible.

```
1  # let pow x n =
2      let rec aux n a =
3        if n = 0 then a
4        else aux (n-1) (x * a)
5      in
6        aux n 1 ;;
7  val pow : int -> int -> int = <fun>
8  # pow 2 5;;
9  - : int = 32
```

ici à la ligne 4 le paramètre x de pow est utilisé dans aux.

C'est un style camélien assez classique.

Polymorphisme paramétrique

appelé « généricité » dans d'autres langages

- même code pour des arguments de types différents
- la fonction n'utilise pas la structure de l'argument

```
# let mp a b = a,b;;
val mp : 'a -> 'b -> 'a * 'b = <fun>

# let x = mp 3 6.8;;
val x : int * float = 3, 6.8

# let y = mp true [1];;
val y : bool * int list = true, [1]

# fst x;;
- : int = 3
```

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# let app x y = x y;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
# app id 1;;
- : int = 1
```

OCaml (et F#) : Résumé des expressions rencontrées

```
expr ::= constante
       | ( expr )
       | ident
       | Mod.ident
       | op expr
       | expr infix-op expr
       | if expr then expr else expr
       | function ident -> expr
       | expr expr
       | expr , expr
       | epxr :: epxr
       | let [rec] ident = expr
           [ and ident = expr ]*          // pas en F#
         in expr
```

OCaml : List.hd et List.tl

F# : List.head et List.tail

Exemple

Composition de fonctions:

- OCaml et Swift

```
# let compose f g x = f (g x);;
val compose :
  ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let add2 x = x + 2 and mult5 x = x * 5;;
val add2 : int -> int = <fun>
val mult5 : int -> int = <fun>

# compose mult5 add2 9;;
- : int = 55
```

- Swift

```
  1> func compose<R,S,T>(f:(S) -> R, g: (T) -> S, x : T) -> R {
          return f(g(x)) }
  2> func add2 (x:Int) -> Int { return x + 2 }
  3> func mult5 (x:Int) -> Int { return 5 * x }
  4> compose(f:mult5, g:add2, x:9)
$R0: Int = 55
```

Filtrage de motif

permet l'accès aux structures de données

- en testant une valeur
- en nommant une partie de la structure

motif:

- assemblage correct (syntaxe et type) d'objets
  - de types de base (*int*, *bool*,...)
  - de paires, listes et constructeurs
  - d'identificateurs
  - du motif « ramasse tout » (_)
- ce n'est pas une expression (il n'a pas de calcul)

Syntaxe du filtage de motif

Syntaxe:

match e **with** p_1 -> e_1 | p_2 -> e_2 ...| p_n -> e_n

- filtrage séquentiel de l'expression e par les différents motifs p_i.
- Si un des motifs correspond à la valeur de e, alors sa branche e_i est évaluée.
- tous les e_i sont du même type, idem pout les p_i et le type de e
- motif linéaire (motif non-linéraire comme (x,x)) interdit)
- détection d'un filtrage non exhaustif
- détection de branches inutiles

Exemple : fonction imply

- Enumération des cas

```
# let imply v =
     match v with
     |    (true,true) -> true
     |   (true,false) -> false
     |   (false,true) -> true
     |  (false,false) -> true;;
val imply : bool * bool -> bool = <fun>
```

- Version plus compacte

```
# let imply v =
     match v with
     |  (true,x)  -> x
     |  (false,_) -> true;;
val imply : bool * bool -> bool = <fun>
```

Warnings

- filtrage non exhaustif :

```
# let f x =
     match x with
     | (true, x) -> true
     | (false, false) -> true;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(false, true)
val f : bool * bool -> bool = <fun>
```

- cas inutile :

```
# let f x =
     match x with
     | (a,b) -> true
     | (true,false) -> false;;
Warning: this match case is unused.
val f : bool * bool -> bool = <fun>
```

Style camélien : définition par cas

en utilisant le filtrage de motifs sur les différents constructeurs d'uns structure de données :

```
let rec long l =
  match l with
  |   [] -> 0
  | _::q -> 1 + long q ;;
val long : 'a list -> int = <fun>
```

```
let rec map f l =
  match l with
  | [] -> []
  | t::q -> (f t) :: map f q ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
let rec mem_assoc x l =
  match l with
  | [] -> false
  | (y,_)::q -> if x = y then true else mem_assoc x q ;;
val mem_assoc : 'a -> ('a * 'b) list -> bool = <fun>
```

Motif dans les déclarations (1)

- **Formes équivalentes:**

  Syntaxe:

  match *e* **with** filtrage $\equiv$ *(***function** filtrage*)* *e*

```
1  #  match (2,1)
2      with  (n,0) -> -1 | (0,n) -> 0 | (n,1) -> n | (n,d) -> n / d ;;
3  - : int = 2
4  # function (n,0) -> -1 | (0,n) -> 0 | (n,1) -> n | (n,d) -> n / d ;;
5  - : int * int -> int = <fun>
6  # (function (n,0) -> -1 | (0,n) -> 0 | (n,1) -> n | (n,d) -> n / d)
7     (2,1) ;;
8  - : int = 2
```

Motif dans les déclarations (2)

- **Déclarations déstructurantes:**

  Syntaxe:

  let $p = e$ p est un motif

```
1  # let (x,y) = (3,true);;
2  val x : int = 3
3  val y : bool = true
```

Déclarations de types en OCaml (1)

- produit cartésien : enregistrement

  Syntaxe:

  type *nom* = enregistrement

- union discriminante : somme avec constructeurs

  Syntaxe:

  type *nom* = union

- abréviation

  Syntaxe:

  type nom = nom2

Déclarations de types en OCaml (2)

- déclarations combinées

  Syntaxe:

  type *nom_1* = ...
  and *nom_2* = ...
  and *nom_n* = ...

- avec paramètres

  Syntaxe:

  type (p1, p2, ..., pn) *nom* = ...

Enregistrements (1)

Syntaxe:

type t =$\{$f1 : t1 ; f2 : t2 ; . . . ; fn : tn$\}$

- constructeur : `{ ... }`
- accesseurs : `.re` et `.im`

```
# type complex = {re:float;im:float} ;;
type complex = { re: float; im: float}

# let c = {re=2.;im=3.};;
val c : complex = {re=2; im=3}

# let add_complex c1 c2 =
   {re=c1.re+.c2.re; im=c1.im+.c2.im};;
val add_complex :
   complex -> complex -> complex = <fun>

# add_complex c c;;
   - : complex = {re=4; im=6}
```

Enregistrements (2)

pouvant aussi définir un motif pour un filtrage :

```
1
2
3  # let mult_complex c1 c2 =
4      match (c1,c2) with
5        ({re=x1;im=y1}, {re=x2;im=y2}) ->
6           {re=x1*.x2-.y1*.y2;im=x1*.y2+.x2*.y1};;
7  val mult_complex :
8     complex -> complex -> complex = <fun>
9
10 # mult_complex c c;;
11 - : complex = {re=-5; im=12}
```

Unions discriminantes

sommes avec constructeurs:

Syntaxe:

type *t* = C1 | *C2* **of** *t1* | ... | *Cm* **of** *t2* / *Cn*

Constructeurs constants:

```
1  # type piece = Pile | Face;;
2  type piece = | Pile | Face
3
4  # Pile;;
5  - : piece = Pile
6
7  # [Pile; Face; Face; Pile];;
8  - : piece list = [Pile; Face; Face; Pile]
```

Constructeurs avec paramètres

```
# type couleur = Pique | Coeur | Carreau | Trefle;;
type couleur = | Pique | Coeur | Carreau | Trefle

# type carte = As of couleur
           | Roi of couleur
           | Dame of couleur
           | Valet of couleur
           | Autre of couleur * int
;;
type carte =...

# let valeur couleur_atout cr = match cr with
| As _ -> 11
| Roi _ -> 4
| Dame _ -> 3
| Valet c -> if c = couleur_atout then 20 else 2
| Autre (_,10) -> 10
| Autre (c,9)  -> if c = couleur_atout then 14 else 0
| _ -> 0
;;
val valeur : couleur -> carte -> int = <fun>
```

Types récursifs

Les déclarations de types sont récursives

```
# type intArbre = IntEmpty
                | IntNode of intArbre * int * intArbre ;;
type intArbre = ...

# let monarbre =
    IntNode ( IntNode ( IntEmpty, 4, IntEmpty ),
              2,
              IntNode ( IntEmpty, 1, IntEmpty ) ) ;;
val monarbre : intArbre = ...

# let rec nb_noeuds a = match a with
  IntEmpty -> 0
| IntNode (fg, _, fd) -> 1 + (nb_noeuds fg) + (nb_noeuds fd);;
val nb_noeuds : intArbre -> int = <fun>

# nb_noeuds monarbre;;
- : int = 3
```

Types paramétrés

Les déclarations de types peuvent être paramétrées

```
# type 'a arbre = Empty
                | Node of 'a arbre * 'a * 'a arbre;;
type 'a arbre = ...

# Empty;;
- : 'a arbre = Empty
# Node (Empty, 1, Empty);;
- : int arbre = Node (Empty, 1, Empty)
# Node (Empty, 1.0, Empty);;
- : float arbre = Node (Empty, 1., Empty)
# Node (Node (Empty, 1, Empty), 4, Empty);;
- : int arbre = Node (Node (Empty, 1, Empty), 4, Empty)

# let rec long_ma a = match a with
  Empty -> 0
| Node(fg,_,fd) -> 1 + long_ma fg + long_ma fd;;
val long_ma : 'a arbre -> int = <fun>

# long_ma (Node (Empty, 2, Node (Empty, 3, Empty)));;
- : int = 2
# long_ma (Node (Empty, 'a', Node (Empty, 'z', Empty)));;
- : int = 2
```

Constructeurs et enregistrements en OCaml

L'argument d'un constructeur d'un type somme peut être défini (et utilisé) à la manière d'un enregistrement.

```
# type 'a arbreB = Vide
  | Feuille of 'a
  | Noeud of {etiq : 'a; fg : 'a arbreB; fd : 'a arbreB} ;;

# let ma = Noeud {
  etiq = 4;
  fg = Noeud {etiq = 2; fg = Feuille 1; fd = Vide};
  fd = Feuille 8} ;;

# let rec nombre_noeud a = match a with
      Vide -> 0
    | Feuille _ -> 1
    | Noeud n ->   1 + (nombre_noeud n.fg) + (nombre_noeud  n.fd) ;;
```

Exemple : type option

```
# type 'a option = None
                 | Some of 'a;;

# let x = Some Pique;;
val x : couleur option = Some Pique

# let y = None;;
val y : 'a option = None

# let create_as oc = match oc with
  None -> As Pique
| Some  coul -> As coul;;
val create_as : couleur option -> carte

# create_as None;;
- : carte = As Pique

# create_as (Some Coeur);;
- : carte = As Coeur
```

Exemples sur les listes (homogènes)

- liste vide : `[]`
- constructeur `::`
- type paramétré *'a list*
- accesseurs `List.hd` et `List.tl`

```
# [] ;;
- : 'a list
# 1::2::3::[] ;;
- : int list
# [1; 2; 3;] ;;
- : int list
# [1; "hello"; 3] ;;
erreur de typage
# List.hd [1.1; 1.2; 1.3] ;;
- : float = 1.1
# List.hd [] ;;
Exception: Failure "hd".
```

Autres fonctions dans le module `List` : `length`, `mem`, `append`, `map`, `nth`.

`length` : comptage des éléments d'une liste (1)

fonctionne pour toutes les listes (`'a list`), 2 cas :

- la liste `[]` ne contient aucun élément ;
- la liste `x::xs` contient un élément de plus que la liste `xs`.

La fonction `length` satisfait donc les deux équations :

$$\begin{cases} (length\ []) & = & 0 \\ (length\ (x :: xs)) & = & 1 + (length\ xs) \end{cases}$$

```
1  # let rec length (l : 'a list) : int↩
       =
2      if l = [] then 0
3      else 1 + length (List.tl l);;
4  val length : 'a list -> int = <fun>
5  # length [2;4;7];;
6  - : int = 3
7  # length [];;
8  - : int = 0
9  # length ['A'; 'G'];;
10 - : int = 2
```

```
1  let rec length (l : 'a list) : int =
2    match l with
3    | [] -> 0
4    | x::xs -> 1 + (length xs) ;;
5  val length : 'a list -> int
6  # length [2;4;7];;
7  - : int = 3
8  # length [];;
9  - : int = 0
10 #  length ['A'; 'G'];;
11 - : int = 2
```

`mem` : appartenance d'un élément à une liste (1)

fonctionne pour toutes les listes (`'a list`)
3 cas pour `mem z l` :

- si la liste `l` est vide, alors `z` n'appartient pas à la liste ;
- si la liste `l` est de la forme `x::xs` alors, il y a deux possibilités :
  - le premier élément de la liste `l` est égal à `z` et donc `z` appartient à `x::xs`
  - `z` appartient à la liste `xs`

la fonction `mem` satisfait donc les trois équations suivantes :

$$\begin{cases} (mem\ z\ []) & = & false & \\ (mem\ z\ (x :: xs)) & = & true & si\ x\ =\ z \\ (mem\ z\ (x :: xs)) & = & (mem\ z\ xs) & sinon \end{cases}$$

mem : appartenance d'un élément à une liste (2)

Implantation: : 2 versions

```
# let rec mem (z : 'a) (xs : 'a list) : bool = match xs with
  | [] -> false
  | x :: xs -> if (x = z) then true else (mem z xs) ;;
  val mem : 'a -> 'a list -> bool = <fun>
```

```
# let rec mem (z : 'a) (xs : 'a list) : bool = match xs with
  | [] -> false
  | x :: xs -> (x = z) ||  (mem z xs) ;;
val mem : 'a -> 'a list -> bool = <fun>
```

```
# mem 3 [1 ; 2 ; 3 ; 4 ];;
- : bool = true
# mem 3 [1 ; 2 ; 4 ] ;;
- : bool = false
# mem true [] ;;
- : bool = false
# mem true [false] ;;
- : bool = false
# mem [true] [[false]; [true; false]];;
- : bool = false
```

append : concaténation de deux listes (1)

fonctionne pour deux listes de même type :
polymorphisme et partage (la seconde liste n'est pas copiée)

La concaténation de deux listes en crée une troisième en suivant les schémas d'équations suivants :

$$\begin{array}{lcl} (append[][y1;...;ym] & = & [y1;..;ym]) \\ (append[x1;...;xn][y1;...;ym] & = & [x1;...;xn;y1;..;ym]) \end{array}$$

On peut noter que :

$$(append(x1 :: [x2;..;xn])[y1;..;ym]) = x1 :: [x2;..;xn;y1;..;ym]$$

d'où la définition de la concaténation apr les deux équations suivantes :

$$\begin{cases} (append\ []\ ys) & = & ys \\ (append\ (x :: zs)\ ys & = & x :: (append\ zs\ ys) \end{cases}$$

append : concaténation de deux listes (2)

Implantation: :

```
#  let rec append (xs : 'a list) (ys : 'a list) : ('a list) = match xs with
 | [] -> ys
 | x :: zs -> x :: (append zs ys)    ;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1 ; 2] [3 ; 4] ;;
- : int list = [1; 2; 3; 4]
```

Exemple (schématique) d'application :

$$
\begin{aligned}
&(append[x1; x2; x3][y1; y2; y3; y4]) \\
&\quad = \quad x1 :: (append[x2; x3][y1; y2; y3; y4]) \\
&\quad = \quad x1 :: x2 :: (append[x3][y1; y2; y3; y4]) \\
&\quad = \quad x1 :: x2 :: x3 :: (append[][y1; y2; y3; y4]) \\
&\quad = \quad x1 :: x2 :: x3 :: [y1; y2; y3; y4]
\end{aligned}
$$

à noter qu'en OCaml la fonction List.append se note aussi par le symbole infixe @

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

map : application d'une fonction sur les éléments d'une liste

$$map\ f\ [x1\ ;\ \ldots\ ;\ xn] \quad = \quad [f\ x1\ ;\ \ldots\ ;\ f\ xn]$$

```
# let rec map f l =
  if l = [] then []
  else
    let t = List.hd l
    and q = List.tl l in
      (f t) :: (map f q) ;;
val map : ('a -> 'b) -> 'a ↩
     list -> 'b list = <↩
     fun>
```

```
let rec map f l =
  match l with
  | [] -> []
  | t::q -> (f t) :: map f q ;;
  val map : ('a -> 'b) -> 'a list -> 'b ↩
       list = <fun>
```

```
# let add2 x = x + 2 ;;
val add2 : int -> int = <fun>
# let l1 = map add2 [7; 2; 9] ;;
val l1 : int list = [9; 4; 11]

# let l2 = map length [['z';'a';'m']; ↩
     ['.';'n';'e';'t']] ;;
val l2 : int list = [3; 4]
```

```
#  map (append [10;  20]) ↩
     [[3;4;5];  [8; 3; 1]];;
- : int list list = [[10; 20; 3; ↩
     4; 5]; [10; 20; 8; 3; 1]]
```

`nth` : recherche du n-ième élément d'une liste (1)

fonction partielle :

l'appel `nth [x0 ; x1 ; ... ; xn] i` retourne la valeur `xi` si $0 \leq i \leq n$.

`nth` doit satisfaire les équations conditioonnelles suivantes :
$(nth(x :: xs)i) = xsii = 0(nth(x :: xs)i) = (nthxs(i-1))sinon$

Attention : si l'indice `i` n'est pas dans l'intervalle $[0, (length\ l) - 1]$.

nth : recherche du n-ième élément d'une liste (2)

Implantation:

```
# let rec nth (xs : 'a list) (i:int) : 'a =
  match xs with
  | [] -> raise (Failure "nth")
  | x::xs -> if (i=0) then x else (nth xs (i-1))      ;;
val nth : 'a list -> int -> 'a = <fun>

# nth ['a' ; 'm' ;  'l' ] 2;;
- : char = 'l'
# nth ['a' ; 'm' ;  'l' ] 3;;
Exception: Failure "nth".
```

Fonctions sur les listes - module List (1)

2 constructeurs (infixes) :

- `[]` : liste vide
- `::` (cons) : liste non vide

```
1  let rec length_aux len l = match l with
2      [] -> len
3    | a::l -> length_aux (len + 1) l
4
5  let length l = length_aux 0 l
6
7  let rec rev_append l1 l2 =
8    match l1 with
9      [] -> l2
10   | a :: l -> rev_append l (a :: l2)
11
12 let rev l = rev_append l []
13
14 (*
15   val length_aux : int -> 'a list -> int = <fun>
16   val length : 'a list -> int = <fun>
17   val rev_append : 'a list -> 'a list -> 'a list = <fun>
18   val rev : 'a list -> 'a list = <fun>
19 *)
```

Fonctions sur les listes - module List (2)

```
1  let rec map f = function
2      [] -> []
3    | a::l -> let r = f a in r :: map f l
4
5  let rev_map f l =
6    let rec rmap_f accu = function
7      | [] -> accu
8      | a::l -> rmap_f (f a :: accu) l
9    in
10   rmap_f [] l
11 ;;
12
13 (*
14   val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
15   val rev_map : ('a -> 'b) -> 'a list -> 'b list = <fun>
16 *)
```

Fonctions sur les listes - module List (3)

```
let rec fold_left f accu l = (* fold_left f r [e1;e2;e3]=f(f(f r e1)e2)e3 *)
  match l with
    [] -> accu
  | a::l -> fold_left f (f accu a) l

let rec fold_right f l accu = (* fold_right f [e1;e2;e3] r=f e1(f e2( fe3 r))↩
      *)
  match l with
    [] -> accu
    | a::l -> f a (fold_right f l accu)

(* val fold_left :   ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
   val fold_right :  ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun> *)
```

```
List.fold_left (+) 0 [8;4;10];;
(* - : int = 22 *)
List.fold_right (+) [8; 4; 10] 0;;
(* - : int = 22  *)

List.fold_left (/) 0 [8;4;10];;
(* - : int = 0 *)
List.fold_right (/) [8;4;10] 0;;
(* Exception: Division_by_zero. *)
```

Exemple : arbres (1)

Représentation des arbres:

```
# type 'a arbre =  Empty
| Node of 'a * 'a arbre list;;
type 'a arbre = ...

# let rec nombre_noeud a = match a with
  Empty -> 0
| Node (_,[]) -> 1
| Node (_,h::t) ->
    1 + (List.fold_left (+) (nombre_noeud h) (List.map nombre_noeud t));;
val nombre_noeud : 'a arbre -> int = <fun>

# let rec hauteur a = match a with
  Empty -> 0
| Node (_,[]) -> 1
| Node (_,h::t) ->
    1 + (List.fold_left (max) (hauteur h) (List.map hauteur t));;
val hauteur : 'a arbre -> int = <fun>
```

Exemple arbres (2)

```
# let rec somme_noeud a = match a with
  Empty -> 0
| Node (e,[]) -> e
| Node (e,h::t) ->
    e + (List.fold_left (+) (somme_noeud h) (List.map somme_noeud t));;
val somme_noeud : int arbre -> int = <fun>

# let rec app_arbre e a =
    let rec app_s_arbre  l = match l with
      [] -> false
    | h::t -> app_arbre e h || app_s_arbre t
    in
      match a with  Empty -> false
        | Node (ne,l) ->
           ne = e  || app_s_arbre l;;
val app_arbre : 'a -> 'a arbre -> bool =
```

Types fonctionnels

```
1  type 'a listf =
2      Val of 'a
3    | Fun of ('a -> 'a) * 'a listf ;;
4  (* type 'a listf = | Val of 'a | Fun of ('a -> 'a) * 'a listf *)
5
6  let huit_div = (/) 8 ;;
7  (* val huit_div : int -> int = <fun> *)
8
9  let gl = Fun (succ, (Fun (huit_div, Val 4))) ;;
10 (* val gl : int listf = Fun (<fun>, Fun (<fun>, Val 4))*)
11
12 let rec compute = function
13     Val v -> v
14   | Fun(f, x) -> f (compute x) ;;
15 (* val compute : 'a listf -> 'a = <fun> *)
16  compute gl;;
17 (* - : int = 3 *)
```

Typage et domaine de définition

type inféré $\neq$ domaine de définition:

- c'est une approximation
- exemple : division entière, tête de liste vide
- provient souvent d'un filtrage non exhaustif

Que faire ?:

- utiliser une valeur spéciale (ou plusieurs)

```
1 # asin 2.;;
2 - : float =          NaN
```

- effectuer une rupture de calcul jusqu'à un récupérateur d'une telle rupture
  $\Rightarrow$ exceptions

Exceptions

Une exception est une rupture de calcul.
utilisée :

- pour éviter les erreurs de calcul
  - division par zéro
  - accès à la référence `null`
  - ouverture d'un fichier inexistant
  - ...
- comme style de programmation
  - sortie de boucles
  - remontée directe d'appels imbriqués
  - ...

En OCaml une exception est une valeur de type exn

Exceptions

Syntaxe:

exception *E1* ; ;

exception *E1* **of** *t1* ; ;

- une exception est une valeur de type *exn*
- le type *exn* est un type somme monomorphe **extensible**

```
1  # exception A_MOI;;
2  exception A_MOI
3
4  # A_MOI;;
5  - : exn = A_MOI
6
7  # exception Depth of int;;
8  exception Depth of int
9
10 # Depth 4;;
11 - : exn = Depth(4)
```

Déclenchement d'une exception

`raise` : *exn* `-> 'a`

- impossible à écrire ⇒ primitive
- l'expression (`raise` E1) n'a pas de contrainte de type car elle ne calcule pas de valeur

---

```
1  # raise A_MOI ;;
2  Exception: A_MOI
3
4  # let x = 18 ;;
5  val x : int = 18
6
7  # if (x = 0) then raise A_MOI else x ;;
8  - : int = 18
9  # if (x = 10) then raise A_MOI else x ;;
10 Exception: A_MOI.
```

Déclarations et déclenchements (1)

```
# exception Echec of string;;
exception Echec of string

# let declenche_echec s = raise (Echec s);;
val declenche_echec : string -> 'a = <fun>

# declenche_echec "argument invalide";;
Exception: Echec "argument invalide".
```

la fonction `failwith` s'écrit :

```
let failwith s = raise (Failure s);;

let invalid_argument s =
   raise (Invalid_argument s);;
```

Déclarations et déclenchements (2)

```
# exception OrthoExn  of int * int * string;;
exception OrthoExn of int * int * string

# raise (OrthoExn (3, 6, "le caml"));;
Exception: OrthoExn (3, 6, "le caml").

# exception FuncTreat of (int -> int);;
exception FuncTreat of (int -> int)

# raise (FuncTreat (fun x -> x + 1));;
Exception: FuncTreat <fun>.
```

Déclarations et déclenchements (3)

Filtrage de motifs incomplet:

```
1  # let tete l = match l with t::q -> t;;
2  Warning: this pattern-matching is not exhaustive.
3  Here is an example of a value that is not matched:
4  []
5  val tete : 'a list -> 'a = <fun>
6
7  # tete [1;2;3];;
8  - : int = 1
9
10 # tete [];;
11 Exception: Match_failure ("", 13, 35).
```

Déclarations et déclenchements (4)

```
# exception Found_zero;;
exception Found_zero

# let rec mult_aux l= match l with
      h::[] -> h
   |  0::t -> raise Found_zero
   |  h::t -> h * mult_aux t ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val mult_aux : int list -> int = <fun>
```

Récupération d'exceptions

Syntaxe:

try expr **with** filtrage

Le type des motifs du *filtrage* doit être *exn*.

```
1  # let mult_list l = match l with
2    [] -> 0
3  | lo ->  try mult_aux lo with
4      Found_zero -> 0;;
5  val mult_list : int list -> int = <fun>
6
7  # mult_list [1;2;3;0;5;6];;
8  - : int = 0
```

Module List (1)

```
1
2  let hd = function
3      [] -> failwith "hd"
4    | a::l -> a
5
6  let tl = function
7      [] -> failwith "tl"
8    | a::l -> l
9
10 let rec nth l n =
11   match l with
12     [] -> failwith "nth"
13   | a::l ->
14       if n = 0 then a else
15       if n > 0 then nth l (n-1) else
16       invalid_arg "List.nth"
```

Module List (2)

```
1
2  #let rec fold_left f accu l =
3    match l with
4      [] -> accu
5    | a::l -> fold_left f (f accu a) l
6  val fold_left :
7    ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
8
9  # let rec fold_right f l accu =
10   match l with
11     [] -> accu
12   | a::l -> f a (fold_right f l accu)
13 val fold_right :
14   ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Exemples fonctionnels

```
# fold_left (/) 1000 [3;5;11];;
- : int = 6

# fold_left (/) 1000 [3;0;11];;
Exception: Division_by_zero.

# let idiv a b = b / a;;
val idiv : int -> int -> int = <fun>

# fold_right idiv [3;5;11]  1000;;
- : int = 6

# fold_right idiv  [3;0;11] 1000;;
Exception: Division_by_zero.
```

Exemple : filtrage d'une liste

- filtrage des éléments d'une liste par un prédicat
- sans recopie inutile

```
# exception Identity;;
exception Identity

# let share f x = try f x with Identity -> x;;
val share : ('a -> 'a) -> 'a -> 'a = <fun>

# let filter f l =
 let rec fil l = match l with
   | [] -> raise Identity
   | h :: t ->
      if f h then h :: fil t else share fil t in
 share fil l;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Utilisation des exceptions

- Gestion de situations exceptionnelles où le calcul ne peut pas se poursuivre $\rightarrow$ rupture du calcul
- style de programmation : exemple précédent (`filter`)

Attention au coût du `try`