

Master Sciences et Technologies
Mention Informatique
Spécialité Sciences et Technologies du Logiciel
Université Paris 6
2006 – 2007

OCAML en quelques transparents ... (noyau fonctionnel)

Mathieu Jaume `Mathieu.Jaume@lip6.fr`

1

Langage OCaml

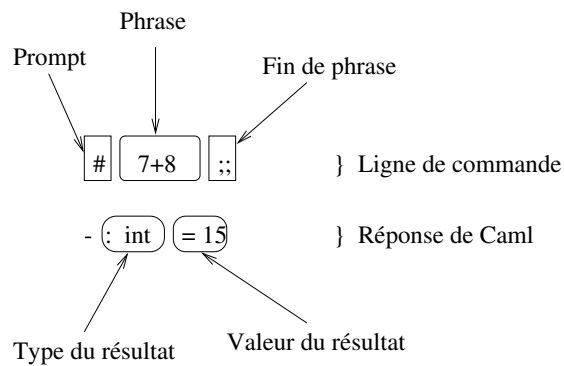
Langage typé fonctionnel (+ traits impératifs et traits objets) développé par l'INRIA.

<http://www.ocaml.org/>

- Programme fonctionnel : suite de définitions d'expressions (fonctions)
- Exécution d'un programme fonctionnel : évaluation d'une expression (application d'une fonction à des arguments)

2

Session interactive



```
# 2+3;;
- : int = 5
# "deux"^^"trois";;
- : string = "deuxtros"
# "deux" + 3;;
This expression has type string but is here used with type int
# x;;
Unbound value x
```

3

Définitions – Liaisons

Conserver le résultat d'un "calcul" : associer un **nom** à une **valeur** dans l'**environnement**

Environnement : liste de couples (**nom**, **valeur**)

```
let ident = expression ;;
```

```
# let s=1+2+3;;
val s : int = 6
# s;;
- : int = 6
# s/2;;
- : int = 3
# let s2=s*(s-1);;
val s2 : int = 30
```

4

Masquage

Définitions \neq Affectations

L'effet d'une définition est de lier une valeur à un nom qui pourra être utilisé comme abréviation à la place de la valeur. Il est impossible de changer la valeur liée à un nom, on peut seulement **masquer** la définition courante par une nouvelle définition.

```
# let z = 0.5;;  
val z : float = 0.5 [(z,0.5)]  
# z ;;  
- : float = 0.5  
# let z = true;;  
val z : bool = true [(z,true);(z,0.5)]  
# z ;;  
- : bool = true
```

5

Définitions locales

Une définition locale sert à lier un nom **temporairement**, c'est-à-dire durant l'évaluation d'une expression.

```
let ident = expression1 in expression2 ;;
```

```
# s;;  
- s : float = 0.5  
# let s=9*11 in s*s;;  
- : int = 9801  
# s;;  
- s : float = 0.5  
# let a = let s=9*11 in s*s;;  
val a : int = 9801
```

6

Types de base

- **int** : entiers relatifs sur lesquels portent les opérateurs $+$, $-$, $*$, $/$, mod ...
- **float** : flottants sur lesquels portent les opérateurs $+$, $-$, $*$, $/$, exp , log , sin ...
- **bool** : booléens **true** et **false** sur lesquels portent les opérateurs **&** (et), **or** (ou), **not** et qui peuvent être retournés par exemple par des opérateurs de comparaison $<$, \leq , $>$, $=$ qui s'appliquent sur des entiers ou des flottants.
- **char** : caractères
- **string** : chaînes de caractères sur lesquelles porte l'opérateur \wedge de concaténation.

7

Fonctions anonymes

```
function paramètre -> expression ;;
```

Exemple. La fonction $x \mapsto x + 1$ s'écrit en OCAML :

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

On peut l'appliquer à un entier :

```
# (function x -> x+1) 4;;  
- : int = 5
```

La variable x est **liée** : elle correspond à l'argument sur lequel sera appliquée la fonction.

Le nom x n'est pas forcément lié à une valeur dans l'environnement.

8

Fonctions : variables libres / variables liées

```
# y;;  
Unbound value y  
# function x -> x - y;;  
Unbound value y
```

x est une variable **liée**

y est une variable **libre** : il est nécessaire que l'environnement puisse associer une valeur à y.

```
# let y=1;;  
val y : int = 1  
# function x -> x - y;;  
- : int -> int = <fun>  
# (function x -> x - y) 5;;  
- : int = 4
```

9

Définition de fonctions

```
let nom_fonction p1 p2 ... pn = exp ;;  
let nom_fonction = function p1 -> function p2 -> ... -> function pn -> exp ;;
```

```
# y;;  
- : int = 1  
# let f=function x -> x - y;;  
val f : int -> int = <fun>  
# let f x = x - y;;  
val f : int -> int = <fun>  
# f 5  
- : int = 4
```

10

Définition de fonctions : portée statique

```
# y;;  
- : int = 1  
# let f=function x -> x - y;;  
val f : int -> int = <fun>  
# f 5  
- : int = 4  
# let y=3;;  
val y : int = 3  
# y;;  
- : int = 3  
# f 5;;  
- : int = 4
```

La valeur d'une fonction (<fun>) est une fermeture qui contient l'environnement courant au moment de la définition de la fonction : c'est cet environnement qui sera utilisé pour évaluer le corps de la fonction au moment de son application.

La liaison de y à la valeur 1 qui existait **au moment de la définition** de f n'a pas été modifiée par la liaison de y à 3 : **portée statique**.

11

Définition locale de fonctions

```
# let carre=function x -> x*.x in sqrt(carre 3. +. carre 4.);;  
- : float = 5
```

Exemple.

```
# let x=5;;  
val x : int = 5  
# let x = x+1  
  in let f = function y -> x + y  
  in let x = 3  
  in f x;;  
- : int = 9
```

12

Expressions conditionnelles

```
if expression_booléenne then expression1 else expression2 ;;
```

expression_booléenne est une expression de type bool – expression1 et expression2 sont deux expressions de même type.

La partie else est obligatoire.

```
# let abs x = if x > 0 then x else 0 - x ;;
val abs : int -> int = <fun>
# let h = function x -> if x=1 then failwith "h indefinie"
                        else 1/(x - 1);;
val h : int -> int = <fun>
# h 5;;
- : int = 0
# h 1;;
Exception: Failure "h indefini".
```

13

Fonctions récursives

```
# let rec fact = function n -> if n=0
                              then 1 else n*fact(n-1);;
val fact : int -> int = <fun>
# fact 6;;
- : int = 720
```

Fonctions mutuellement récursives.

```
# let rec pair =
    function n -> if n = 0 then true
                 else if n=1 then false else impair(n-1)
and   impair = function n -> if n = 0
                              then false else pair(n-1);;
val pair : int -> bool = <fun>
    impair : int -> bool = <fun>
```

14

Application partielle

Exemple. moyenne arithmétique de deux entiers

```
# let moyenne=function x -> function y -> (x+y)/2;;  
val moyenne : int -> int -> int = <fun>  
int -> (int -> int) : fonction qui à un entier associe une fonction de  
type int -> int.
```

```
# moyenne 3 5;;
```

```
- : int = 4
```

```
# moyenne 3;;
```

Application partielle

```
- : int -> int = <fun>
```

```
# let moy1 = moyenne 3;;
```

```
val moy1 : int -> int = <fun>
```

```
# moy1 5;;
```

```
- : int = 4
```

15

Liaison filtrante

Il est possible de lier l'argument d'une fonction à un n -uplet de variables : **liaison filtrante**.

```
# let moyenne=function (x,y) -> (x+y)/2;;
```

```
val moyenne : int * int -> int = <fun>
```

```
# moyenne 3;;
```

```
This expression has type int but is here used with type int*int
```

```
# moyenne (3,5);;
```

```
- : int = 4
```

16

Polymorphisme – Ordre supérieur

- La fonction identité `let f x = x` de type `'a -> 'a` peut s'appliquer à un argument de type quelconque.
- Une fonction d'ordre supérieur est une fonction qui prend en argument une autre fonction. Par exemple la fonction de composition de fonctions (notée o en mathématiques) s'écrit en OCAML de la façon suivante :

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

17

Types produits : n -uplets

Les valeurs peuvent être regroupées en paires, triplets ou plus généralement en n -uplets.

```
# 4,6.7;;  
- : int * float = 4, 6.7  
# exp 1. , 3<4 , "0'"^"caml";;  
- : float * bool * string = 2.71828182846, true, "0'caml"  
# (5,let s="ab" in s^" ^s");;  
- : int * string = (5, "ab ab")
```

Le parenthésage des expressions ”produits” n'est pas neutre :

$$t_1 * t_2 * t_3 \neq (t_1 * t_2) * t_3 \neq t_1 * (t_2 * t_3)$$

18

Accès aux composantes d'un n -uplets

Accès aux composantes d'un couple :

```
fst : 'a*'b -> 'a    snd : 'a*'b -> 'b
```

Fonctions polymorphes : elles agissent sur des paires dont les éléments sont de types quelconques.

```
# fst(true or false, 4-6);;    # fst((2.5,666),true);;
- : bool = true                - : float * int = 2.5, 666
# snd((2.5,666),true);;
- : bool = true
```

Comment accéder à la troisième composante d'un triplet ? Utilisation d'une liaison filtrante.

```
# let third = function (_,_,x) -> x;;
val third : 'a * 'b * 'c -> 'c = <fun>
# third (0.5+.0.5,2,"trois");;
- : string = "trois"
```

19

n -uplets : Exemple

Fonction calculant le quotient et le reste de la division entière de deux entiers.

```
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);;
(* div : int -> int -> int * int *)
```

20

Types produits : Enregistrements

Un *enregistrement* est un produit cartésien dans lequel les différentes composantes (qu'on appelle aussi des *champs*) sont nommées.

Exemple. Type des nombres complexes

```
type complexe = {re:float;im:float};;

# let c1= {re=1.; im=0.};;           val c1 : complexe = {re=1; im=0}
# let c2 = {re=0.; im=1.};;       val c2 : complexe = {re=0; im=1}
# c1.re;;                          - : float = 1
# c1.im;;                          - : float = 0
# let somme_cplx = fun c1 c2 -> {re=c1.re+.c2.re; im=c1.im+.c2.im};;
val somme_cplx : complexe -> complexe -> complexe = <fun>
# somme_cplx c1 c2;;                - : complexe = {re=1; im=1}
# let somme_cplx =
  function {re=r1;im=i1} {re=r2;im=i2}->{re=r1+.r2;im=i1+.i2};;
val somme_cplx : complexe -> complexe -> complexe = <fun>
```

21

Types sommes avec constructeurs constants

Exemple.

```
type booleen = True | False;;
```

Introduit 2 nouvelles constantes qui peuvent être utilisées comme n'importe quelles autres constantes.

True et False sont des **constructeurs** constants (sans arguments) de valeurs de type booleen. Les constructeurs commencent par une majuscule.

Fonction sur une valeur de type somme : **filtrage**.

```
let negation b = match b with
  True -> False
  | False -> True;;
val negation : booleen -> booleen = <fun>

# negation True;;
- : booleen = False
```

22

Motifs de filtrage

```
type booleen = True | False;;
```

Motif de filtrage “muet”

```
let negation b = match b with
  True -> False   | _ -> True;;
val negation : booleen -> booleen = <fun>
```

Filtrage d'une paire / Variable dans les motifs.

```
let et b1 b2 = match (b1,b2) with
  (False,_) -> False   | (True,x) -> x;;
val et : booleen -> booleen -> booleen = <fun>
```

Le filtrage est linéaire.

```
let foo b1 b2 = match (b1,b2) with
  (x,x) -> False   | _ -> True;;
This variable is bound several times in this matching
```

23

Ordre des motifs de filtrage

Les motifs de filtrage sont envisagés dans l'ordre où ils apparaissent dans la fonction.

```
let negation b = match b with
  True -> False
  | _ -> True;;
val negation : booleen -> booleen = <fun>
```

```
# negation True;;
- : booleen = False
```

```
let negation b = match b with
  _ -> True
  | True -> False;;
Warning: this match case is unused.
val negation : booleen -> booleen = <fun>
```

```
# negation True;;
- : booleen = True
```

24

Exhaustivité du filtrage

```
let negation b = match b with
  True -> False;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
False
val negation : booleen -> booleen = <fun>

# negation False;;
Exception: Match_failure ("", 5, -31).

let negation b = match b with
  True -> False
  | False -> True
  | _ -> False;;
Warning: this match case is unused.
val negation : booleen -> booleen = <fun>
```

25

Type somme – Union disjointe

Les constructeurs d'un type somme peuvent avoir des arguments.
Permet de réunir dans un même type des valeurs de types différents.

```
# type nombre = Entier of int | Flottant of float;;
```

Union disjointe d'ensembles :

$$E_1 \uplus E_2 = \{g(x) \mid x \in E_1\} \cup \{d(x) \mid x \in E_2\}$$

Injections canoniques :

$$g : E_1 \rightarrow E_1 \uplus E_2 \quad d : E_2 \rightarrow E_1 \uplus E_2$$

$$x \in E_1 \uplus E_2 \Leftrightarrow (\exists y \in E_1 \ x = g(y)) \text{ XOR } (\exists y \in E_2 \ x = d(y))$$

A chaque composante de l'union correspond un constructeur.

Le type `nombre` est l'union disjointe des types `int` et `float` et les constructeurs `Entier` et `Flottant` sont les injections canoniques des types `int` et `float` dans le type `nombre`.

26

Type somme

```
# type nombre = Entier of int | Flottant of float;;

# Entier 7*8;;
- : nombre = Entier 56

# Flottant 5.5;;
- : nombre = Flottant 5.5

# let add_nombre arg = match arg with
  (Entier n1, Entier n2) -> Entier(n1+n2)
| (Entier n1, Flottant n2) -> Flottant(float(n1)+.n2)
| (Flottant n1, Entier n2) -> Flottant(n1+.float(n2))
| (Flottant n1, Flottant n2) -> Flottant(n1+.n2);;

val add_nombre : nombre * nombre -> nombre = <fun>
```

Les arguments d'un constructeur sont de types quelconques et peuvent donc être aussi des fonctions.

```
type fonctions_bool = Fbool of (bool -> bool) ;;
```

27

Type somme récursif : Entiers de Péano

Un entier est soit 0, soit le successeur $S(n)$ d'un entier n .

```
type nat = Zero | S of nat;;           # let deux = (S (S Zero));;
                                       val deux : nat = S (S Zero)
```

Fonction récursive : addition de deux entiers de type nat

$$n_1 + n_2 = \begin{cases} n_2 & \text{si } n_1 = 0 \\ S(p + n_2) & \text{si } n_1 = S(p) \end{cases}$$

```
let rec add n1 n2 = match n1 with
  Zero -> n2
| (S p) -> (S (add p n2));;
val add : nat -> nat -> nat = <fun>
```

```
# add deux deux;;
- : nat = S (S (S (S Zero)))
```

28

Type somme récursif : Listes d'entiers

Une liste est soit la liste vide LV , soit une liste $cons(e, \ell)$ composée d'un premier élément entier e suivie de la liste ℓ des éléments suivants.

```
type liste = LV | CONS of int*liste;;  
let l = CONS(2,CONS(1,(CONS(3,LV))));;
```

Fonction récursive : somme des entiers apparaissant dans une liste

$$somme(\ell) = \begin{cases} 0 & \text{si } \ell = LV \\ e + somme(\ell') & \text{si } \ell = cons(e, \ell') \end{cases}$$

```
let rec somme l = match l with  
  LV -> 0  
  | CONS(e,l0) -> e + (somme l0);;  
val somme : liste -> int = <fun>  
  
# somme l;;  
- : int = 6
```

29

Type somme récursif polymorphe : Listes

Une α -liste est soit la liste vide LV , soit une liste $cons(e, \ell)$ composée d'un 1er élément e de type α suivie de la α -liste ℓ des éléments suivants.

```
type 'a liste = LV | CONS of 'a * 'a liste;;  
# let l1 = CONS(2,CONS(1,(CONS(3,LV))));;  
val l1 : int liste = CONS (2, CONS (1, CONS (3, LV)))  
# let l2 = CONS('a',CONS('z',LV));;  
val l2 : char liste = CONS ('a', CONS ('z', LV))
```

Fonction récursive : longueur d'une liste

$$longueur(\ell) = \begin{cases} 0 & \text{si } \ell = LV \\ 1 + longueur(\ell') & \text{si } \ell = cons(e, \ell') \end{cases}$$

```
let rec longueur l = match l with  
  LV -> 0 | CONS(e,l0) -> 1 + (longueur l0);;  
val longueur : 'a liste -> int = <fun>  
# longueur l2;;  
- : int = 2
```

30

Listes

Le type polymorphe `list` des listes est prédéfini en OCaml. C'est un type :

- à deux *constructeurs* :

1. `[]` la liste vide

2. `::` infix tel que $e :: l = l' \begin{cases} \text{hd}(l') = e \\ \text{tl}(l') = l \end{cases}$

Fonctions d'accès aux composants d'une liste :

`hd : 'a list -> 'a` `tl : 'a list -> 'a list`

- *polymorphe* c.a.d. paramétré par une variable de type
- *récuratif* : une *liste* est composée d'un premier élément et de la *liste* des éléments suivants

31

Listes : syntaxe

Il existe une syntaxe abrégée pour les listes :

$[e_1; e_2; \dots; e_n] = e_1 :: (e_2 :: \dots (e_n :: []) \dots)$

`[];`

- : `'a list = []`

`(3*6)::[];`

- : `int list = [18]`

`1::2::3::[];`

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

`[5.2; 6.-.5.4; 3.7];;`

- : `float list = [5.2; 0.6; 3.7]`

`[[1;2]; []; [3;4;5]];`

- : `int list list = [[1; 2]; []; [3; 4; 5]]`

`[0.4; 3; 5];;`

This expression has type int list but is here used with type float list

32

Listes : filtrage (1)

Liaison filtrante, on peut lier plusieurs éléments d'une liste simultanément :

```
# let prod3 [i; j; k] = i*j*k;;  
Warning: this pattern-matching is not exhaustive  
val prod3 : int list -> int = <fun>
```

Fonction récursive sur les listes : test d'appartenance

```
val mem : 'a -> 'a list -> bool
```

mem a l is true if and only if a is equal to an element of l.

```
let rec mem a l = match l with  
  [] -> false  
  | h::t -> if a=h then true else (mem a t);;
```

mem est une fonction prédéfinie du langage OCaml.

Listes : filtrage (2)

```
val append : 'a list -> 'a list -> 'a list
```

Catenate two lists. Same function as the infix operator @. Not tail-recursive (length of the first argument). The @ operator is not tail-recursive either.

```
let rec append l1 l2 = match l1 with  
  [] -> l2  
  | h::t -> h::(append t l2);;
```

```
# List.append [1;2;3] [6;7;8];;  
- : int list = [1; 2; 3; 6; 7; 8]
```

append est une fonction prédéfinie du langage OCaml.

Listes : fonctionnelles (1)

```
val map : ('a -> 'b) -> 'a list -> 'b list
List.map f [a1; ...; an] applies function f to a1, ..., an, and builds
the list [f a1; ...; f an] with the results returned by f. Not
tail-recursive.
```

$$\text{map} : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$$
$$\text{map } f [e_1; e_2; \dots; e_n] = [f(e_1); f(e_2); \dots; f(e_n)]$$

```
let rec map f l = match l with
  [] -> []
  | (a::r) -> (f a) :: (map f r);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# int_of_char;;
- : char -> int = <fun>
# List.map int_of_char ['a'; 'z'; 'A'; 'Z'];;
- : int list = [97; 122; 65; 90]
```

35

Listes : fonctionnelles (2)

```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_left f a [b1; ...; bn] is f (... (f (f a b1) b2) ...) bn.
```

$$\text{fold_left } f b [a_1; a_2; \dots; a_n] = (f(\dots(f(f b a_1)a_2)\dots)a_n)$$

```
# let rec fold_left f e l = match l with
  [] -> e
  | (a::r) -> (fold_left f (f e a) r);;
# List.fold_left (function x -> function y -> x-y) 0 [1;2;3];;
- : int = -6
```

36

Listes : fonctionnelles (3)

```
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.fold_right f [a1; ...; an] b is f a1 (f a2 (... (f an b) ...)).
```

Not tail-recursive.

$$\text{fold_right } f [a_1; a_2; \dots; a_n] b = f a_1 (f a_2 (\dots (f a_n b) \dots))$$

```
# let rec fold_right f l e = match l with
  [] -> e
  | (a::r) -> f a (fold_right f r e);;

# List.fold_right (function x -> function y -> x-y) [1;2;3] 0;;
- : int = 2
```

37

Listes : fonctionnelles (4)

```
val exists : ('a -> bool) -> 'a list -> bool
exists p [a1; ...; an] checks if at least one element of the list satisfies
the predicate p. That is, it returns (p a1) || (p a2) || ... || (p an).
```

$$(\text{exists } p l) = \text{true} \Leftrightarrow \exists x \in l p(x)$$

```
let rec exists p l = match l with
  [] -> false
  | h::t -> if (p h) then true else (exists p t);;

# List.exists (function x -> (x mod 2)=0) [1;3];;
- : bool = false
# List.exists (function x -> x>0) [1;3];;
- : bool = true
```

38

Listes : fonctionnelles (5)

```
val for_all : ('a -> bool) -> 'a list -> bool
```

exists p [a1; ...; an] checks if all elements of the list satisfies the predicate p. That is, it returns (p a1) && (p a2) && ... && (p an).

$$(\text{for_all } p \ l) = \text{true} \Leftrightarrow \forall x \in l \ p(x)$$

```
let rec forall p l = match l with
  [] -> true
  | h::t -> if (p h) then (for_all p t) else false;;

# List.for_all (function x -> (x mod 2)=0) [1;3];;
- : bool = false
# List.for_all (function x -> x>0) [1;3];;
- : bool = true
```

39

Listes : fonctionnelles (6)

```
val flatten : 'a list list -> 'a list
```

Same as concat. (Concatenate a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result). Not tail-recursive (length of the argument + length of the longest sub-list).

```
let rec flatten ll = match ll with
  [] -> []
  | h::t -> (append h (flatten t));;

# List.flatten [[1;2];[2;4;5]];
- : int list = [1; 2; 2; 4; 5]
```

40

Listes : fonctionnelles (7)

```
val filter : ('a -> bool) -> 'a list -> 'a list
filter p l returns all the elements of the list l that satisfy the predicate
p. The order of the elements in the input list is preserved.
let rec filter p l = match l with
  [] -> []
  | h::t -> if (p h) then h::(filter p t) else (filter p t);;

# List.filter (function x -> (x mod 2)=0) [1;2;3;4];;
- : int list = [2; 4]
```

41

Listes : fonctionnelles (8)

```
val combine : 'a list -> 'b list -> ('a * 'b) list
Transform a pair of lists into a list of pairs:
combine [a1; ...; an] [b1; ...; bn] is [(a1,b1); ...; (an,bn)]. Raise
Invalid_argument if the two lists have different lengths. Not
tail-recursive.
let rec combine l1 l2 = match (l1,l2) with
  ([],[]) -> []
  | ((h1::t1),(h2::t2)) -> (h1,h2)::(combine t1 t2)
  | _ -> raise (Invalid_argument "List.combine");;

# List.combine ['a';'b';'c'] [1;2;3];;
- : (char * int) list = [('a', 1); ('b', 2); ('c', 3)]
```

42

Liste d'associations

Liste de couples (*key, value*)

```
val assoc : 'a -> ('a * 'b) list -> 'b
```

assoc a l returns *the value associated with key a in the list of pairs l*.
That is, `assoc a [...; (a,b); ...] = b` if (a,b) is the leftmost binding of a in list l. Raise `Not_found` if there is no value associated with a in the list l.

```
let rec assoc c l = match l with
  [] -> raise Not_found
  | (k,v)::t -> if c=k then v else (assoc c t);;
```

Exemple.

```
# let v = [("p1",true);("p2",true);("p3",false)];;
val v:(string * bool) list = [("p1",true); ("p2",true); ("p3",false)]

# List.assoc "p2" v;;           # List.assoc "p4" v;;
- : bool = true                Exception: Not_found.
```

43

Exceptions

Traiter les cas d'erreur ou les cas exceptionnels :

- Déclencher (lever) un exception arrête le calcul en cours : `raise`.

```
let rec assoc c l = match l with
  [] -> raise Not_found
  | (k,v)::t -> if c=k then v else (assoc c t);;

# List.assoc;;           - : 'a -> ('a * 'b) list -> 'b = <fun>
# Not_found;;           # raise;;
- : exn = Not_found     - : exn -> 'a = <fun>
```

- Il est possible de rattraper une exception : `try ... with`.

```
let valeur_de_verite p v =
  try List.assoc p v with Not_found -> failwith "undefined symbol";;
valeur_de_verite : 'a -> ('a * 'b) list -> 'b = <fun>
# let v = [("p1",true);("p2",true);("p3",false)];;
# valeur_de_verite "p2" v;;   - : bool = true
# valeur_de_verite "p4" v;;   Exception: Failure "undefined symbol".
```

44