Le langage Objective Caml

Université Pierre et Marie Curie

Mise à niveau, Maîtrise

Sept. 2003

— o —

Caractéristiques

- langage fonctionnel,
- typé statiquement,
- polymorphe paramétrique,
- avec inférence de types,
- muni d'un mécanisme d'exceptions
- et de traits impératifs,
- possèdant un système de modules paramétrés
- et un modèle objet,
- possiblement indépendant de l'architecture machine.

Histoire

- l'ancêtre : ML (*meta-langage*) de LCF (80)
- machine abstraite : la CAM ; Curien, Cousineau
- spécifications : Standard ML (84 - Milner)
- premières implantations
  - CAML - Suarez - Weis - Mauny (87)
  - SML/NJ - Mc Queen - Appel (ATT - 88)
- nouvelles implantations : Caml-light (90) Leroy - Doligez
- modules paramétrés : CSL (95)
- extension objet (96)

Mises en œuvre (1/3)

Compilateur de byte-code

```
$ cat p.ml
let f x = x+1
print_int (f 1999); print_newline()
$ ocamlc -i -o p.exe p.ml
val f : int -> int

$ file p.exe
p.exe: a /usr/local/bin/ocamlrun script text
$ ./p.exe
2000
```

Mises en œuvre (2/3)

Compilateur natif (pour Intel, Sparc, HP-pa, PowerPC, etc.)

```
$ ocmalopt -o p.exe p.ml
$ file pp.exe
p.exe: ELF 32-bit LSB executable, Intel 80386,
version 1 (FreeBSD), dynamically linked, not stripped
$ ./p.exe
2000
```

Mises en œuvre (3/3)

Boucle d'interaction (byte code à la volée)

```
$ ocaml
        Objective Caml version 3.06

# let f x = x+1 ;;
val f : int -> int = <fun>
# print_int (f 1999); print_newline() ;;
2000
- : unit = ()
# #quit ;;
$
```

Plan du 1ier jour

1. Types et opérateurs numériques
2. Définir et utiliser des fonctions (sur les nombres)
3. Manipuler les nombres
4. Autres types de base simples
5. Types et structures de données paramétrés
6. Manipuler ces types
7. Types utilisateurs
8. Des expressions (synthèse)
9. Des programmes

Types et opérateurs numériques

Les nombres

- Entiers :
  - type `int`
  - valeurs $[-2^{30}, 2^{30} - 1]$
    (sur machine 32 bits)
- Flottants :
  - type `float`
  - valeurs mantisse 53 bits, exposant $[-1022, 1023]$
    (norme IEEE 754)

Les nombres - notation (1/3)

On utilise la boucle d'interaction `ocaml`

Entiers, décimale

```
# 1 ;;
- : int = 1
```

- le `#` est l'invite ;
- `1` est l'expression ;
- le `;;` provoque l'évaluation ;
- on obtient en retour le type (`int`) et la valeur (`1`)

Les nombres - notation (2/3)

Entiers, négatifs, hexadécimale, octale, binaire

```
# -1 ;;
- : int = -1
# 0xffffffff ;;
- : int = -1
# 0o10 ;;
- : int = 8
# 0b10 ;;
- : int = 2
```

Les nombres - notation (3/3)

Flottants, décimale pointée, exponentielle

```
# 1.0 ;;
- : float = 1.000000
# 1. ;;
- : float = 1.000000
# 0.1 ;;
- : float = 0.100000
# 1.e-1 ;;
- : float = 0.100000
```

Opérations sur les nombres (quelques)

Entiers		Flottants	
`+`	addition	`+.`	addition
`-`	soustraction	`-.`	soustraction
`*`	multiplication	`*.`	multiplication
`/`	division entière	`/.`	division
`mod`	modulo	`**`	exponentiation

Opérateurs infixes à précédance

```
# 1 + 2 * 3 mod 4;;
- : int = 3
```

Définition et utilisation de fonctions

Définir une fonction

Déclaration, type inféré

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

Notez: pas de *return*

Application, préfixe

```
# cube 3 ;;
- : int = 27
```

Appliquer une fonction

Attention à la priorité

```
# cube 3+1 ;;
- : int = 28
# cube (3+1) ;;
- : int = 64
# cube cube 3 ;;
This function is applied to too many arguments
# cube (cube 3) ;;
- : int = 19683
```

L'application associe *à gauche*

Définir une fonction récursive

Construction alternative

`if` *test* `then` *expr1* `else` *expr2*

Déclaration récursive *explicite*

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

Définir une fonction récursive partielle

Erreurs et exceptions

```
# let rec fact n =
    if n < 0 then failwith "undefined"
    else if n = 0 then 1
    else n * (fact (n-1))
;;
# fact (-1)
Uncaught exception: Failure "undefined".
```

Remarquez les parenthèses autour de `-1`

Définir des fonctions mutuellements récursives

Construction `let rec ... and ...`

```
# let rec even n =
    if n=0 then true
    else odd (n-1)

  and odd n =
    if n=1 then true
    else even (n-1)
;;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
```

Définir: divers

Fonction sans argument ou constante

```
# let x = 3 ;;
val x : int = 3
```

Définition locale, construction `let ... in`

```
# let puiss4 n =
    let n2 = n*n in n2*n2 ;;
val puiss4 : int -> int = <fun>
# puiss4 2 ;;
- : int = 16
```

Manipuler les nombres

Calculs sur les nombres (1/3)

Calculs entiers *modulo*

```
# max_int ;;
- : int = 1073741823
# min_int ;;
- : int = -1073741824
# max_int+1;;
- : int = -1073741824
```

Erreur (exception) ou valeurs spéciales

```
# 1 / 0 ;;
Uncaught exception: Division_by_zero.
# 1.0 /. 0.0 ;;
- : float = Inf
```

Calculs sur les nombres (2/3)

Opérateurs fortement typés :

```
# 1.0 / 0 ;;
This expression has type float but is here used with type int
# 1.0 /. 0 ;;
This expression has type int but is here used with type float
```

Conversion explicite (lire le type) :

```
# float_of_int ;;
- : int -> float = <fun>
# (float_of_int 1) /. 2.0 ;;
- : float = 0.500000
```

Calculs sur les nombres (3/3)

Attention aux débordements (non spécifié) :

```
# (float_of_int max_int) ** 2. ;;
- : float = 1152921502459363328.000000
# int_of_float ;;
- : float -> int = <fun>
# int_of_float ((float_of_int max_int) ** 2.) ;;
- : int = 0
```

Affichages

```
# print_int ;;
- : int -> unit = <fun>
# print_float ;;
- : float -> unit = <fun>
# print_newline ;;
- : unit -> unit = <fun>
# print_int 1; print_newline();
  print_float 1.0; print_newline() ;;
1
1.
- : unit = ()
```

Affichages (commentaires)

`unit` en Objective Caml ≈ `void` en C

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

Le type `unit` contient la seule valeur `()`

Opérateur de mise en séquence (d'évaluations) `;` (le point virgule)

```
# print_int (2+3); 2+3 ;;
5- : int = 5
```

La séquence est une expression dont la valeur est la dernière

Autres types et opérateurs de bases

Caractères (1/2)

Type `char`

Notations

```
# 'A' ;;
- : char = 'A'
# '\n' ;;
- : char = '\n'
# '\010' ;;
- : char = '\n'
```

Caractères (2/2)

Code ASCII, ISO 8859-1 standard.

```
# int_of_char 'A' ;;
- : int = 10
# char_of_int 65 ;;
- : char = 'A'
# char_of_int 0 ;;
- : char = '\000'
# char_of_int (-1) ;;
Uncaught exception: Invalid_argument "char_of_int".
# char_of_int 256 ;;
Uncaught exception: Invalid_argument "char_of_int".
```

Chaînes de caractères (1/3)

Type `string`

Notation

```
# "Hello" ;;
- : string = "Hello"
# "" ;;
- : string = ""
```

Concaténation (opérateur infixe) :

```
# "Hello" ^ " (O'Caml) " ^ "world\n" ;;
- : string = "Hello (O'Caml) world\n"
```

Chaînes de caractères (2/3)

Opérateur typé, fonction de conversion :

```
# 'H' ^ "ello" ;;
This expression has type char but is here used with type string
# "2000" + 1 ;;
This expression has type string but is here used with type int
# int_of_string ;;
- : string -> int = <fun>
# (int_of_string "2000") + 1 ;;
- : int = 2001
# int_of_string "2 000" ;;
Uncaught exception: Failure "int_of_string".
# string_of_char ;;
Unbound value string_of_char
```

Chaînes de caractères (3/3)

Bibliothèque (module) `String`

Création

```
# String.make 12 '.' ;;
- : string = "............"
# String.create 12 ;;
- : string = "\216\141\013\008H\253\t\008\008Z\006\008"
```

Taille maximale (module `Sys`)

```
# Sys.max_string_length ;;
- : int = 16777211
# String.create (Sys.max_string_length + 1) ;;
Uncaught exception: Invalid_argument "String.create".
```

Affichages

Standard

```
# print_char ;;
- : char -> unit = <fun>
# print_string ;;
- : string -> unit = <fun>
```

Bibliothèque Printf

```
# Printf.printf "%d %f %c %s \n" 1 1. '1' "1" ;;
1 1.000000 1 1
- : unit = ()
```

Les booléens

Type `bool`

Deux constantes :

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

Opérateurs

`not`	négation		
`&&`	conjonction	`&`	synonyme
`||`	disjonctions	`or`	synonyme

Les opérateurs binaires sont infixes et séquentiels

Connecteurs *vs* opérateurs logiques

<table>
<tr><th colspan="2">Booléens</th><th colspan="2">Entiers</th></tr>
<tr><td><code>not</code></td><td>négation</td><td><code>lnot</code></td><td>négation logique</td></tr>
<tr><td><code>&&</code></td><td>conjonction</td><td><code>land</code></td><td>conjonction logique (bit à bit)</td></tr>
<tr><td><code>||</code></td><td>disjonctions</td><td><code>lor</code></td><td>disjonction logique (bit à bit)</td></tr>
</table>

```
# lnot false ;;
This expression has type bool but is here used with type int
# lnot 0 ;;
- : int = -1
# lnot 0xffffffff;;
- : int = 0
```

Relations

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

Les relations sont des opèrateurs *polymorphes* mais homogènes

```
# 0 < 1 ;;
- : bool = true
# false < true ;;
- : bool = true
# 0 < true ;;
This expression has type int but is here used with type bool
```

Types paramétrés

Produits cartésiens (1/3)

Structures de données *polymorphes*

Constructeur de type : `*` (l'étoile)

Constructeur de valeur : `,` (la virgule)
infixe, polymorphe, hétérogène

```
# 1, "mai" ;;
- : int * string
# "may", 1 ;;
- : string * int = "may", 1
```

Produits cartésiens (2/3)

Accesseurs : `fst` et `snd` (polymorphes)

```
# fst ;;
- : 'a * 'b -> 'a = <fun>
# snd ;;
- : 'a * 'b -> 'b = <fun>
```

Notez l'écriture des *variables* de type

Remarques :

`'a * 'b -> 'a` $\equiv$ `('a * 'b) -> 'a`

`'a * 'b -> 'a` $\not\equiv$ `'a * ('b -> 'a)`

Produits cartésiens (3/3)

Attention à la syntaxe

```
# fst (1,"mai") ;;
- : int = 1
# fst 1, "mai" ;;
This expression has type int but is here used with type 'a * 'b
```

Objective Caml est un langage fonctionnel

```
# int_of_char, char_of_int ;;
- : (char -> int) * (int -> char) = <fun>, <fun>
# (fst (int_of_char, char_of_int)) 'A' ;;
- : int = 65
```

Apparté: qu'est-ce qu'une fonction binaire ? (1/2)

```
# let pol1 (n, m) = 3*n*n + 2*m + 1 ;;
val pol1 : int * int -> int = <fun>
# pol1 (1, 2) ;;
- : int = 8
# pol1 1 2 ;;
This function is applied to too many arguments
```

Un couple est *une* valeur

```
# let pol1' c = 3*(fst c)*(fst c) + 2*(snd c) + 1 ;;
val pol1' : int * int -> int = <fun>
```

Apparté: qu'est-ce qu'une fonction binaire ? (2/2)

Ça n'existe pas !

```
# let pol2 n m = 3*n*n + 2*m + 1 ;;
val pol2 : int -> int -> int = <fun>
# pol2 1 ;;
- : int -> int = <fun>
# pol2 1 2 ;;
- : int = 8
```

Car

1. `int -> int -> int` ≡ `int -> (int -> int)`
2. `pol2 1 2` ≡ `((pol2 1) 2)`

Couples *vs* n-uplets

```
# 1, "mai", 2001 ;;
- : int * string * int = 1, "mai", 2001
# (1, "mai"), 2001 ;;
- : (int * string) * int = (1, "mai"), 2001
# (1, "mai", 2001) = ((1, "mai"), 2001)  ;;
This expression has type int * string * int but is here used
with type (int * string) * int
# (1, "mai", 2001) = (1, ("mai", 2001)) ;;
This expression has type int * string * int but is here used
with type int * (string * int)
```

Listes polymorphes homogènes (1/3)

Type `'a list`

- constructeurs
  - liste vide, notée `[]`
  - ajout d'un élément en tête, noté `::` (infixe)
- accesseurs (module `List`)
  - élément en tête, `List.hd : 'a list -> 'a`

    `List.hd (x::xs)` ≡ `x`
  - suite de la liste, `List.tl : 'a list -> 'a list`

    `List.tl (x::xs)` ≡ `xs`

Listes polymorphes homogènes (2/3)

Notation, construction

```
# [];;
- : 'a list = []
# [1; 2; 3] ;;
- : int list = [1; 2; 3]
# 0::[1; 2; 3] ;;
- : int list = [0; 1; 2; 3]
# 0::1::2::3::[] = [0; 1; 2; 3] ;;
- : int list = [1; 2; 3]
```

Concaténation

```
# [0; 1] @ [2; 3] ;;
- : int list = [0; 1; 2; 3]
```

Listes polymorphes homogènes (3/3)

Listes *homogènes*

```
# [1; "mai"] ;;
This expression has type string but is here used with type int
```

Objective Caml est un langage fonctionnel

```
# [String.uppercase; String.lowercase] ;;
- : (string -> string) list = [<fun>; <fun>]
# (List.hd [String.uppercase; String.lowercase]) "hello" ;;
- : string = "HELLO"
```

Manipuler des listes

Filtrage (1/7)

Destructurer ce qui a été construit

Définition par cas de constructeur: opérateur **match**

```
# let rec sum ns =
    match ns with
      [] -> 0
    | n::ns' -> n + (sum ns') ;;
val sum : int list -> int = <fun>
# sum [6; 60; 600] ;;
- : int = 666
```

Filtrage (2/7)

Motifs de filtrage : constructeurs et variables

Un motif fabrique une *liaison*

```
# let head_and_tail xs =
    match xs with
      [] -> failwith "empty"
    | x::xs' -> x, xs' ;;
val head_and_tail : 'a list -> 'a * 'a list = <fun>
# head_and_tail [0; 1; 2] ;;
- : int * int list = 0, [1; 2]
```

Filtrage (3/7)

Liaisons inutiles: motif universel _ (souligné)

```
# let head_only xs =
    match xs with
      [] -> failwith "empty"
    | x::_ -> x ;;
# head_only [0; 1; 2] ;;
- : int = 0
```

Filtrage (4/7)

L'évaluation du filtrage est *séquentielle*

```
# let bad_head_only xs =
    match xs with
      _ -> failwith "first pattern"
    | x::_ -> x ;;
Warning: this match case is unused.
# bad_head_only [0; 1; 2] ;;
Uncaught exception: Failure "first pattern".
```

Filtrage (5/7)

Du travail en profondeur

```
# let rec pairing xs =
    match xs with
      [] -> []
    | [x] -> failwith "should be of even length"
    | x1::x2::xs' -> (x1,x2)::(pairing xs') ;;
```

Alternative

```
# let rec pairing xs =
    match xs with
      [] -> []
    | x1::x2::xs' -> (x1,x2)::(pairing xs')
    | _ -> failwith "should have even elements" ;;
```

Filtrage (6/7)

Du travail encore plus en profondeur

```
# let rec rem_zero nss =
    match ns with
      [] -> []
    | 0::ns' -> rem_zero ns'
    | n::ns' -> n::(rem_zero ns') ;;
val rem_zero : int list -> int list = <fun>
# rem_zero [1; 0; 2; 0; 0; 3] ;;
- : int list = [1; 2; 3]
```

Notez: les (notations de) constantes entières sont des motifs

Filtrage (7/7)

Un motif est *linéaire*

```
# let rec rem_dup xs =
    match xs with
      x::x::xs' -> rem_dup (x::xs')
    | x::xs' -> x::(rem_dup xs')
    | _ -> xs ;;
This variable is bound several times in this matching
```

Le motif `x::x::xs'` contient deux occurences de la variable `x`

Fonctionnelles (1/3)

Objective Caml est un langage fonctionnel

Appliquer un même traitement

```
val iter :  ('a -> unit) -> 'a list -> unit
   List.iter f [a1; ...; an] ≡ f a1; f a2; ...; f an; ()
```

```
# let print_int_list =
    let print_fun n = Printf.printf"(%d)" n in
      List.iter print_fun
;;
val print_int_list : int list -> unit = <fun>
# print_int_list [0; 1; 2; 3] ;;
(0)(1)(2)(3)- : unit = ()
```

Fonctionnelles (2/3)

Appliquer un même traitement et reconstruire

```
val map :  ('a -> 'b) -> 'a list -> 'b list
     List.map f [a1; ...; an] ≡ [f a1; ...; f an]
```

```
# let rev_pairs xys =
    let f (x,y) = (y,x) in
      List.map f xys
;;
val rev_pairs : ('a * 'b) list -> ('b * 'a) list = <fun>
# rev_pairs [1,true; 2,false; 3,true; 4,false] ;;
- : (bool * int) list = [true, 1; false, 2; true, 3; false, 4]
```

Fonctionnelles (3/3)

Composition

```
val fold_right :  ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

```
List.fold_right f [a1; ...; an] b
 ≡ f a1 (f a2 (...  (f an b) ...))
```

```
# let sum ns = List.fold_right (+) ns 0 ;;
val sum : int list -> int = <fun>
# sum [1; 2; 3; 4] ;;
- : int = 10
```

Remarque: `(+)` est la version préfixée de l'opérateur `+`

```
let (+) x y = x + y ;;
```

Types utilisateurs

Types union (1/3)

Définir son propre ensemble de constructeurs

```
# type int_or_float =
    Int_value of int
  | Float_value of float ;;
```

Remarquez: initiales des constructeurs *obligatoirement* en majuscules

Les valeurs

```
# Int_value 1 ;;
- : int_or_float = Int_value 1
# Float_value 1.5 ;;
- : int_or_float = Float_value 1.500000
```

Types union (2/3)

Filtrage

```
# let add x y =
    match x, y with
      Int_value n1, Int_value n2 -> Int_value (n1 + n2)
    | Float_value x1, Float_value x2 -> Float_value (x1 +. x2)
    | _ -> failwith "type error" ;;
```

Remarquez: la construction du couple pour le filtrage

Type union (3/3)

Objective Caml est un langage fonctionnel

```
# type fun_and_arg =
    Arg  of int
  | Fun of (int -> int) ;;
[...]
# let apply t =
    match t with
      Fun f, Arg x -> f x
    | _ -> failwith"some this wrong"
;;
val apply : fun_and_arg * fun_and_arg -> int = <fun>
# apply (Fun succ, Arg 1) ;;
- : int = 2
```

Type récursifs (1/2)

Arbres binaires

Version *monomorphe*

```
# type int_btree =
    Empty
  | Node of int_btree * int * int_btree ;;
```

Version paramétrique

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

Type récursifs (2/2)

Parcours générique

```
let rec btree_fold f a t =
  match t with
    Empty -> a
  | Node(t1, x, t2)
    -> f x (btree_fold f a t1) (btree_fold f a t1)
;;
val btree_fold :
  ('a -> 'b -> 'b -> 'b) -> 'b -> 'a btree -> 'b = <fun>
```

Types mutuellement récursifs (1/3)

Arbres et forêts

```
# type 'a tree =
    Leaf
  | Branch of 'a * 'a forest

  and 'a forest =
    Empty
  | Cons of 'a tree * 'a forest
;;
```

Types mutuellement récursifs (2/3)

Parcours générique

```
# let rec tree_fold f a g b t =
    match t with
      Leaf -> a
    | Branch (x, ts) -> f x (forest_fold f a g b ts)
  and forest_fold f a g b ts =
    match ts with
      Empty -> b
    | Cons (t, ts')
      ->  g (tree_fold f a g b t) (forest_fold f a g b ts')
;;
```

Types mutuellement récursifs (3/3)

Type des parcours génériques

```
val tree_fold :
  ('a -> 'b -> 'c) -> 'c -> ('c -> 'b -> 'b) -> 'b
   -> 'a tree -> 'c = <fun>
val forest_fold :
  ('a -> 'b -> 'c) -> 'c -> ('c -> 'b -> 'b) -> 'b
   -> 'a forest -> 'b = <fun>
```

Aux limites de l'abstraction fonctionnelle

Types produits: enregistrements (1/5)

Nommer les champs d'un n-uplet

```
# type fiche =
   { nom: string;
     prenom: string;
     naissance: int*int*int } ;;
```

Oublier leur ordre

```
# let f1 = { prenom = "Gilberte";
             naissance = 01,01,01;
             nom = "Albertine" } ;;
val f1 : fiche =
  {nom="Albertine"; prenom="Gilberte"; naissance=1, 1, 1}
```

Types produits: enregistrements (2/5)

Accés, notation pointée

```
# f1.nom ;;
- : string = "Albertine"
```

Accés, par filtrage

```
# let get_nom f =
    match f with
      {nom=x} -> x
;;
val get_nom : fiche -> string = <fun>
# get_nom f1 ;;
- : string = "Albertine"
```

Types produits: enregistrements (3/5)

Champs fonctionnels

```
# type t = { value: int; next: int -> int } ;;
[...]
# let e0 = { value = 0; next = succ } ;;
val e0 : t = {value=0; next=<fun>}
```

Construction ... `with` ...

```
# let e1 = { e0 with value = e0.next e0.value } ;;
val e1 : t = {value=1; next=<fun>}
```

Types produits: enregistrements (4/5)

Types (naturellement) récursifs

```
type t = { value: int; next: t -> t } ;;
[...]
# let e1 =
    let succ_t e = { e with value = succ e.value } in
      { value = 2; next = succ_t } ;;
val e1 : t = {value=2; next=<fun>}
# let e2 =
    let square_t e = { e with value =  e.value * e.value } in
      { value = 2; next = square_t } ;;
val e2 : t = {value=2; next=<fun>}
```

Types produits: enregistrements (5/5)

Une succession qui dépend de l'argument initial

```
# let next_t e = e.next e ;;
val next_t : t -> t = <fun>
# let e1' = next_t e1 ;;
val e1' : t = {value=3; next=<fun>}
# let e1'' = next_t e1' ;;
val e1' : t = {value=4; next=<fun>}
# let e2' = next_t e2 ;;
val e2' : t = {value=4; next=<fun>}
# let e2'' = next_t e2' ;;
val e2'' : t = {value=16; next=<fun>}
```

Des expressions

L'application

Syntaxe: juxtaposition simple

$$e_1 \ e_2 \ \ldots e_n$$

Valeur: dépend du type

si $e_1 : \tau_2$ `->` τ_1 et si $e_2 : \tau_2$ alors $e_1 \ e_2 : \tau_1$

Évaluation (ordre d'): argument(s) d'abord

```
# let f x = print_string" puis j'oublie mon argument\n" ;;
val f : 'a -> unit = <fun>
# f (print_string"je suis l'argument") ;;
je suis l'argument puis j'oublie mon argument
- : unit = ()
```

L'abstraction

Syntaxe: liaison

`fun x -> e`

Valeur: *fermeture*

Fige l'évaluation de `e` en attente de la valeur de `x`

```
# fun x -> failwith"j'ai oublie mon argument" ;;
- : 'a -> 'b = <fun>
# (fun x -> failwith"j'ai oublie mon argument") "Go" ;;
Uncaught exception: Failure "j'ai oublie mon argument".
```

Type:

de type τ_1 `->` τ_2, si `e`:τ_2 en supposant `x`:τ_1

La séquence (structure de contrôle (1/3))

Syntaxe:

$$e_1 ; \; e_2 ; \; \ldots ; \; e_n$$

Valeur: valeur de e_n

Évaluation (ordre d'): de droite à gauche

```
# print_string"je vaux "; 2001 ;;
je vaux - : int = 2001
# failwith"je m'arrete la"; 2001 ;;
Uncaught exception: Failure "je m'arrete la".
```

L'alternative (structure de contrôle (2/3))

Syntaxe:

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

Valeur: celle de e_2 ou e_3

Évaluation (ordre d'): e_1, puis e_2 ou e_3

```
# if true then 20/20 else 0/0 ;;
- : int = 1
# (if true then 20/20 else 0/0) + 10 ;;
- : int = 11
```

Le filtre (structure de contrôle (3/3))

Syntaxe:

match e with p_1 -> e_1 | ...| p_n -> e_n

Valeur: celle du premier e_i tel que p_i filtre e

Évaluation (ordre d'): e, puis, et seulement lui, le e_i sélectionné

```
# match failwith"je ne ressemble a rien" with _ -> 0/0 ;;
Uncaught exception: Failure "je ne ressemble a rien".
# (match true with false -> 0/0 | true -> 20/20) + 10 ;;
- : int = 11
```

Définition locale

Syntaxe:

$$\texttt{let x = } \mathtt{e}_1 \texttt{ in } \mathtt{e}_2$$

Valeur: celle de $\mathtt{e}_2$ quand `x` vaut celle de $\mathtt{e}_1$

Évaluation (ordre d'): $\mathtt{e}_1$ d'abord, puis $\mathtt{e}_2$

```
# let x = print_string"je vaux " in 1 ;;
je vaux - : int = 1
# let x = failwith"je ne vaux rien" in 1 ;;
Uncaught exception: Failure "je ne vaux rien".
```

Des programmes

Déclarations globales

Syntaxe:

```
let x = e
```

Effet: attribut à `x` la valeur de `e`

Variations

```
# let a = 3 and b = 5 and c = 7 ;;
[...]
# let fun_poly = fun x -> fun y -> a*x*x + b*y + c ;;
val fun_poly : int -> int -> int = <fun>
# fun_poly 1 2 ;;
- : int = 20
```

Schéma de programmes

```
(* Commentaire de programme *)
(* Declarations globales *)
let x1 = e1
⋮
let xn = en
(* Expression principale *)
e(n+1)
```

1. Les déclarations créent *l'environnement* d'évaluation
2. L'expression principale déclanche l'évaluation

Plan du 2ème jour : 1ère partie

- Exceptions
- Noyau impératif
  - entrées/sorties
  - valeurs physiquement modifiables
  - structures de contrôle

Plan du 2ème jour : 2ème partie

- Modèle mémoire
  - copie et partage
  - récupérateur automatique de mémoire

- Inférence de types
  - mécanisme d'inférence de types
  - polymorphisme paramétrique
  - variables de types faibles

Exceptions

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

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

- effectuer une rupture de calcul jusqu'à un récupérateur (exceptions)

Déclarations d'exceptions

Syntaxe: `Exception E1`;; ou `Exception E1 of t1`;;

Valeur de type `exn`

`exn`: type somme *monomorphe et extensible*

```
# exception A_MOI;;
exception A_MOI
# A_MOI;;
- : exn = A_MOI
# exception Depth of int;;
exception Depth of int
# Depth 4;;
- : 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

```
# raise A_MOI;;
Uncaught exception: A_MOI
# let x = 18;;
val x : int = 18
# if (x = 0) then raise A_MOI else x;;
- : int = 18
```

Déclarations et déclenchements d'exception (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);;
```

Déclarations et déclenchements d'exception (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 d'exception (3)

Filtrage de motifs incomplet:

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

Déclarations et déclenchements d'exception (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: this pattern-matching ...
val mult_aux : int list -> int = <fun>
```

Récupération d'exceptions

Syntaxe: `try` *expr* `with` *filtrage*

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

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

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 → rupture du calcul
- style de programmation : exemple prcdent (`filter`)

Attention au coût du `try`

Programmation impérative

Programmation impérative

- modèle plus proche des machines réelles
- tout est dans `X := X + 1`
  - exécution d'une instruction (action) qui modifie l'état mémoire
  - passage à une nouvelle instruction dans le nouvel état mémoire
- modèle des langages Fortan, Pascal, C, Ada, ...

Entrées/sorties

Canaux:

- types : `in_channel` et `outchannel`
- fonctions : `open_in` : `string`→ `in_channel` (`close_in`)
  `open_out` : `string`→`out_channel` (`close_out`)
- exception : *End_of_file*
- canaux prédéfinis : *stdin*, *stdout* et *stderr*
- fonctions de lecture et d'écriture sur les canaux
- organisation et accès séquentiels
- type `open_flag` pour les modes d'ouverture

Principales fonctions d'ES

Fonction		Type
input	:	in_channel → string → int → int → int
input_line	:	in_channel → string
output	:	out_channel → string → int → int → unit
output_string	:	out_channel → string → unit
read_line	:	unit → string
read_int	:	unit → int
print_string	:	string → unit
print_int	:	int → unit
print_newline	:	unit → unit

Exemple : C+/C- (1/2)

Séquence, effet de bord

```
# let rec cpcm n =
    print_string "taper un nombre : ";
    let i = read_int () in
      if i = n then print_string "BRAVO\n\n"
      else begin
           if i < n then print_string "C+\n"
           else print_string "C-\n";
           cpcm n
           end ;;
val cpcm : int -> unit = <fun>
```

Exemple : C+/C- (2/2)

Exécution

```
# cpcm 64;;
taper un nombre : 88
C-
taper un nombre : 44
C+
taper un nombre : 64
BRAVO

- : unit = ()
#
```

Valeurs physiquement modifiables

- valeurs structurées dont une partie peut être physiquement (en mémoire) modifiée;
- vecteurs, enregistrements à champs modifiables, chaînes de caractères, références

$\Rightarrow$ nécessite de contrôler l'ordre du calcul!!!

Attention: l'ordre d'évaluation des arguments n'est pas spécifié.

Vecteurs (1)

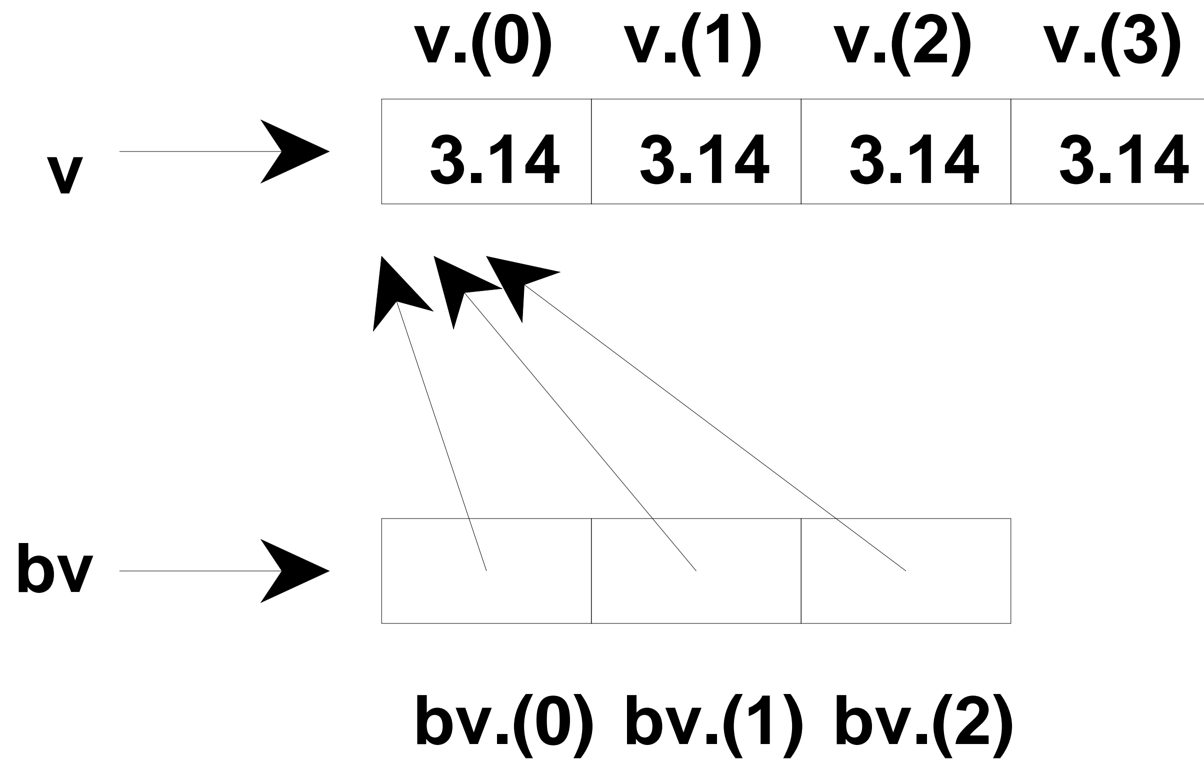
- regroupent un nombre connu d'éléments de même type
- création : `Array.create` : `int` → `'a` → `'a array`,
- longueur : `Array.length` : `'a array`→ `int`
- accès : $e_1.(e_2)$
- modification : $e_1.(e_2)$`<-`e_3

Vecteurs (2)

```
# let v = Array.create 4 3.14;;
val v : float array = [|3.14; 3.14; 3.14; 3.14|]
# v.(1);;
- : float = 3.14
# v.(8);;
Exception: Invalid_argument "Array.get".
# v.(0) <- 100.;;
- : unit = ()
# v;;
- : float array = [|100.; 3.14; 3.14; 3.14|]
```

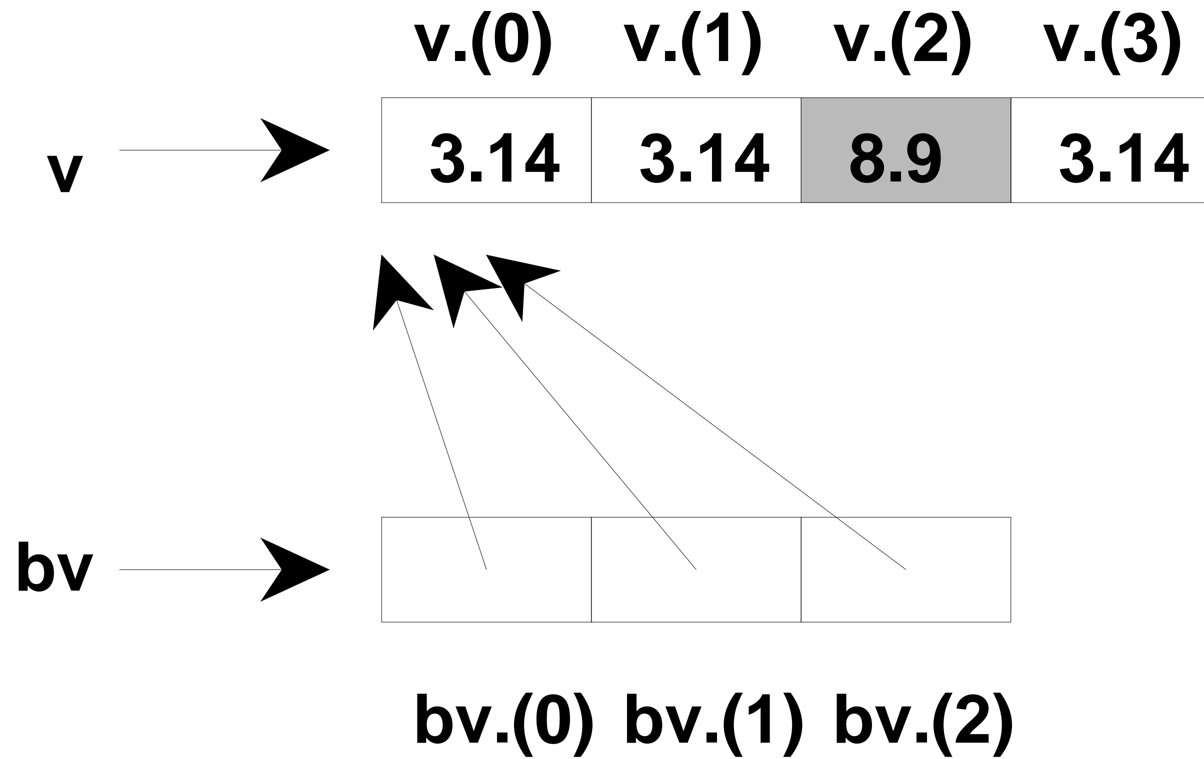
Représentation mémoire (1)

```
# let bv = Array.create 3 v;;
```



Représentation mémoire (2)

```
# v.(2) <- 8.9;;
```



Enregistrements à champs mutables

- indication à la déclaration de type d'un champs est "mutable"
- accès identique $e_1.f_i$, modification $e_1.f_i$<-e_2

type t ={*f1 : t1;* ***mutable*** *f2:t2; ...; fn:tn*} ;;

```
# type point = {mutable x : float; mutable y : float};;
type point = { mutable x: float; mutable y: float }
# let p = {x=1.; y=1.};;
val p : point = {x=1; y=1}
# p.x <- p.x +. 1.0;;
- : unit = ()
# p;;
- : point = {x=2; y=1}
```

Chaînes de caractères

- les chaînes sont des valeurs modifiables (fonction `input`)
- accès : $e_1.[e_2]$
- modification : $e_1.[e_2]$`<-`e_3

```
# let s  = "bonjour";;
val s : string = "bonjour"
# s.[3];;
- : char = 'j'
# s.[3]<-'-';;
- : unit = ()
# s;;
- : string = "bon-our"
```

Références

- sous-cas historique utilisant maintenant des records mutables
  - `type 'a ref = {mutable contents:'a}`
  - `!`e_1 $\equiv$ e_1.contents
  - e_1`:=`e_2 $\equiv$ e_1.contents`<-`e_2

```
# let incr x = x := !x + 1;;
val incr : int ref -> unit = <fun>
# let z = ref 3;;
val z : int ref = {contents=3}
# incr z;;
- : unit = ()
# z;;
- : int ref = {contents=4}
# (ref 3) := 2;;
```

Structures de contrôle

- séquentielle : $e1$; e_2; ...; e_n
  regroupée : (...) ou `begin` ... `end`
  le type de la séquence est le type de e_n
- conditionnelle : `if` c_1 `then` e_2 (e_2 de type `unit`)
- itératives :
  - `while` c `do` e `done`
  - `for` v=e_1 [`down`]`to` e_2 `do` e_3 `done`

La conditionnelle et les boucles sont des expressions de type `unit`

Exemple : somme de 2 vecteurs

```
#let somme a b =
  let al = Array.length a  and bl = Array.length b in
  if al <> bl then failwith "somme"
  else  if al = 0 then a
    else
      let c = Array.create al a.(0) in
        for i=0 to al-1 do
          c.(i) <- a.(i) + b.(i)
        done;
        c;;
val somme : int array -> int array -> int array = <fun>
# somme [|1; 2; 3|] [| 9; 10; 11|];;
- : int array = [|10; 12; 14|]
```

Types et valeurs modifiables

Introduction des traits impératifs (1)

Danger des modifications physiques en présence du polymorphisme :

casser le système de type:

```
let x = ref [] in
  x := 0::!x;
  x := true::!x
```

Il faut l'interdire !

Introduction des traits impératifs (2)

⇒ 2 types d'expressions :

- expression non expansive : variables, constructeurs, abstraction
- expression expansive : application, ref

Seules ces dernières peuvent :

- engendrer une exception
- transgresser le système de types

⇒ utiliser des variables de type faibles (`'_a`)

Introduction des traits impératifs (2)

```
# let x = ref [];;
val x : '_a list ref = {contents = []}
# x := 0::!x;;
- : unit = ()
# x;;
- : int list ref = {contents = [1]}
#
```

On ne peut plus faire `x := true::!x`

Application non généralisée (1)

```
# let id = function x -> x;;
val id : 'a -> 'a = <fun>
# let i = id id;;
val i : '_a -> '_a = <fun>
# i 3;;
- : int = 3
# i;;
- : int -> int = <fun>
# i true;;
This expression has type bool
but is here used with type int
```

Application non généralisée (2)

Solution: ajouter un paramètre

```
# let j y = id id y;;
val j : 'a -> 'a = <fun>
# j 3;;
- : int = 3
# j true;;
- : bool = true
```