Master d'Informatique 1ème année – Université Paris-Sud 11

Mise à niveau en OCaml

10 septembre 2009

Louis Mandel
louis.mandel@lri.fr

D'après des transparents de Jean-Christophe Filliâtre et de Valérie Ménissier-Morain

Pourquoi ce cours ?

Mise à niveau en OCaml OCaml est utilisé en :

- cours de compilation : TD, examens
- projet de compilation : écriture d'un compilateur en OCaml
- Interrogation de OCaml : lundi 28 septembre

À faire :

- lire le poly "Initiation à la programmation fonctionnelle"
- lire le poly "Formation au langage Caml" et faire les exercices
- regarder le manuel d'OCaml
- lire
  `http://caml.inria.fr/resources/doc/guides/guidelines.fr.html`

Page web du projet :

`http://www.lri.fr/~mandel/enseignement/projet-compilation`

Premier programme

Compilateur

- fichier source `hello.ml` :

```
(* Ceci est notre premier programme *)
let _ = Printf.printf "hello world!\n";;
```

- compilation :

```
--> ocamlc -o hello hello.ml
```

- exécution :

```
--> ./hello
```

Premier programme

Boucle d'interaction

- exécution du "toplevel" :

```
--> ocaml
        Objective Caml version 3.11.1
#
```

- interaction :

```
# let _ = Printf.printf "hello world!\n";;
hello world!
- : unit = ()
```

Remarque : le caractère # est le prompt inséré automatiquement par le toplevel

Survol du langage

Déclarations : constantes

```
# let une_constante = 1;;
val une_constante : int = 1
# let _ = 1;;
- : int = 1
# let _ =
    let une_constante_locale = 1.5 in
    une_constante_locale +. 2.7;;
- : float = 4.2
```

Les phrases OCaml se terminent par un `;;` qui est optionnel.

Attention : il y a deux `let` qui sont différents.

Les erreurs de typages

```
# let _ = 1 + "coucou";;
Error: This expression has type string but an expression was expected of type
         int
```

La cohérence des types est vérifiée à la compilation

Déclarations : fonctions

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

```
# let _ = carre (5);;
- : int = 25
# let _ = carre 25;;
- : int = 625
```

Déclarations : références

```
# let une_variable = ref true;;
val une_variable : bool ref = {contents = true}
# let _ = une_variable := false;;
- : unit = ()
# let _ = !une_variable;;
- : bool = false
# let _ = une_variable := 4012;;
Error: This expression has type int but an expression was expected of type
         bool
```

Types produits

```
# let tuple = (9, 9.0, '9', "9");;
val tuple : int * float * char * string = (9, 9., '9', "9")
```

```
# let (prem, sec) = (false, "aaa");;
val prem : bool = false
val sec : string = "aaa"
```

Les parenthèses sont optionnelles

```
# let prem, sec = false, "aaa";;
val prem : bool = false
val sec : string = "aaa"
```

Types produits et polymorphisme

```
# let first = fun (x, y) -> x;;
val first : 'a * 'b -> 'a = <fun>
# let _ = first  (1, 2);;
- : int = 1
# let _ = first ("aaa", "bbb");;
- : string = "aaa"
```

Types produits

```
# let somme = fun (x, y) -> x + y;;
val somme : int * int -> int = <fun>
# let diagonale = fun x -> (x, x);;
val diagonale : 'a -> 'a * 'a = <fun>
# let _ = (diagonale 2, diagonale true);;
- : (int * int) * (bool * bool) = ((2, 2), (true, true))
```

Types enregistrements

```
# type personne = { nom: string; age: int; };;
type personne = { nom : string; age : int; }
# let lambda = { nom = "LA"; age = 21; };;
val lambda : personne = {nom = "LA"; age = 21}
```

```
# let anniv1 = fun x -> { nom = x.nom; age = x.age + 1; };;
val anniv1 : personne -> personne = <fun>
# let anniv2 = fun { nom = n; age = a; } ->
    { nom = n; age = a + 1; };;
val anniv2 : personne -> personne = <fun>
# let _ = anniv1 lambda, lambda;;
- : personne * personne = ({nom = "LA"; age = 22}, {nom = "LA"; age = 21})
```

Types sommes

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

```
# type carte =
    | Dame of couleur
    | Petite_carte of int * couleur;;
type carte = Dame of couleur | Petite_carte of int * couleur
```

```
# let dame_pique = Dame Pique;;
val dame_pique : carte = Dame Pique
```

Types sommes et filtrage de motifs

```
# let points c =
    match c with
    | Dame _ -> 20
    | Petite_carte (10, _) -> 10
    | Petite_carte (v, coul) ->
        if v = 9 && coul = Coeur then 11 else 0;;
val points : carte -> int = <fun>
```

```
# let _ = points dame_pique;;
- : int = 20
# let _ = points (Petite_carte (7, Trefle));;
- : int = 0
```

Programmation impérative

```
# let _ =
    for i = 0 to 9 do
      Printf.printf "%i" i
    done;;
0123456789- : unit = ()
# let t = [| 1; 2; 3; |];;
val t : int array = [|1; 2; 3|]
# let _ =
    let i = ref 0 in
    while !i < Array.length t do
      Printf.printf "%i; " t.(!i);
      i := !i + 1
    done;;
1; 2; 3; - : unit = ()
```

Le langage OCaml plus en détails

Types de base et constantes

- `unit` : unique valeur `()`
- `char` : caractères (`'a'`, `'\080'`, `'\n'`, `'\t'`, ...)
- `int` : entiers relatifs (`1`, `-2`, `4012`, ...)
  - ▷ opérateurs : `+`, `-`, `*`, `/`, `mod`, ...
- `float` : flottants (`0.5`, `-1.2`, `3.7e5`, `15.3e-10`, ...)
  - ▷ opérateurs : `+.`, `-.`, `*.`, `/.`, `sqrt`, `log`, `sin`, ...
- `bool` : booléens (`true`, `false`)
  - ▷ opérateurs : `&&`, `||`, `not`, ...
- `string` : chaînes de caractères (`"bonjour"`, ...)
  - ▷ opérateurs : `^` (concaténation), ...

Les déclarations globales

$$\begin{array}{rcl} \textit{declaration} & ::= & \texttt{let}\ \textit{pattern}\ \texttt{=}\ \textit{expression}\ [\texttt{;;}] \\ & | & \texttt{type}\ \textit{ident}\ \texttt{=}\ \textit{type-definition}\ [\texttt{;;}] \end{array}$$

Un programme est une suite de déclaration de valeurs et de types

- exemple :

```
let x = 1 + 2;;
let _ = Printf.printf "%i" x;;
let y = x * x;;
let _ = Printf.printf "%i" y;;
let a, b = ("bonjour", 6);;
type point = { x: int; y: int; };;
let origine = { x = 0; y = 0; };;
```

- les « ;; » sont optionnels.

Notion de variable

`let` x = *expression* `;;` introduit une variable globale x

- Différences avec la notion usuelle de variable :
  - nécessairement initialisée
  - type pas déclaré mais inféré
  - contenu non modifiable

Références

Une variable modifiable s'appelle une référence

- introduite par un `ref`
- exemple

```
# let x = ref 1;;
val x : int ref = {contents = 1}
# let _ = x := !x + 1;;
- : unit = ()
# let _ = print_int !x;;
2- : unit = ()
```

Expressions/Instructions

Pas de distinctions entre expressions et instructions

- il n'y a que des expressions
- exemples
  - en Caml

```
# let a = if !x = 1 then 1 else 2;;
```

  - en C

```
int a = (x == 1) ? 1 : 2;
```

  - en Caml

```
# let b = x := 42; Printf.printf "%d" !x; 2;;
```

  - en C

```
int b = (x = 42, printf("%d", x), 2);
```

Expressions

Toutes les expressions ont un type et s'évaluent vers une valeur de ce type

- l'expression : `1.5 +. 2.0`
  - a le type `float`
  - s'évalue vers la valeur `3.5`
- l'expression : `x := 3`
  - a le type `unit`
  - s'évalue vers la valeur `()`
- l'expression : `if 6 mod 2 = 0 then 'a' else 'b'`
  - a le type `char`
  - s'évalue vers la valeur `'a'`
- l'expression : `if 6 mod 2 = 0 then 'a' else ()`
  - n'est pas bien typée

Expressions

expression ::= *constante*

| *expression* `+` *expression*

| *expression* `-` *expression*

| ...

| `ref` *expression*

| `if` *expression* `then` *expression* `else` *expression*

| *expression* `;` *expression*

| ...

Variables locales

$$\begin{array}{rcl} \textit{expression} & ::= & \ldots \\ & \mid & \texttt{let}\ \textit{pattern}\ \texttt{=}\ \textit{expression}\ \texttt{in}\ \textit{expression} \end{array}$$

```
# let global =
    let local = 1 in local + 1;;
val global : int = 2
# let _ = global + local;;
Error: Unbound value local
```

- Comme une variable globale :
  - nécessairement initialisée
  - type pas déclaré mais inféré
  - contenu non modifiable
  - mais portée limitée à l'expression qui suit le `in`

Portée des variables

- En C

```
{  int x = 1;
   printf("x = %d; ", x);
   { int x = 2; printf("x = %d; ", x); }
   printf("x = %d; ", x);
}
```

- En Caml

```
# let _ =
    let x = 1 in
    Printf.printf "x = %d; " x;
    (let x = 2 in Printf.printf "x = %d; " x);
    Printf.printf "x = %d; " x;;
x = 1; x = 2; x = 1; - : unit = ()
```

let/in

Autres exemples

```
# let x = 3;;
val x : int = 3
# let _ = let x = 1 in x + 1;;
- : int = 2
# let _ = x + 1;;
- : int = 4
# let _ =
    let aux = 6 * 24 + 3868 in
    (aux / 2, aux + 1);;
- : int * int = (2006, 4013)
```

L'expression `let` x = e_1 `in` e_2

- a le type et la valeur de e_2
- dans l'environnement où x à le type et la valeur de e_1

Définitions simultanées

```
# let _ =
    let a = 1 in
    let b = "hello" in
    let a = b in
    let b = a in
    (a, b);;
- : string * string = ("hello", "hello")
```

```
# let _ =
    let a = 1 in
    let b = "hello" in
    let a = b and b = a in
    (a, b);;
- : string * int = ("hello", 1)
```

Portée statique

```
# let const = 3;;
val const : int = 3
# let f x = x + const;;
val f : int -> int = <fun>
# let _ = f 1;;
- : int = 4
# let const = 6;;
val const : int = 6
# let _ = f 1;;
- : int = 4
```

Les listes

Une liste est une suite (ordonnée) de valeurs de même type

```
# let liste_vide = [];;
val liste_vide : 'a list = []
# let liste_a_3_elements = [ 1; 2; 3; ];;
val liste_a_3_elements : int list = [1; 2; 3]
# let liste_de_booleen = [ false; 1 = 2; ];;
val liste_de_booleen : bool list = [false; false]
# let liste_de_liste = [ [ 1; 3; ]; []; [ 2; 4; ] ];;
val liste_de_liste : int list list = [[1; 3]; []; [2; 4]]

# let probleme = [ true; "faux"; ];;
Error: This expression has type string but an expression was expected of type
         bool
```

Les listes : Cons

L'opérateur prédéfini `::`

- se lit « cons » (prononcer le s final)
- est un opérateur infixe
- prend en argument
  - un élément `x`
  - une liste `l`
- `x :: l` retourne la liste où `x` est suivi de `l`

```
# let _ = 1 :: [ 2; 3; ];;
- : int list = [1; 2; 3]
# let _ = 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
# let _ = 'a' :: 'b' :: [] = [ 'a'; 'b' ];;
- : bool = true
```

Les listes : polymorphisme

La liste vide peut être considérée comme une liste d'entiers, de flottants, ...

```
# let _ = 1 :: [];;
- : int list = [1]
# let _ = 1.5e-10 :: [];;
- : float list = [1.5e-10]
# let _ = 1 :: [ 1.5e-10 ];;
Error: This expression has type float but an expression was expected of type
         int
# let _ = [];;
- : 'a list = []
```

Le type 'a se lit souvent α (alpha)

Les listes : filtrage de motifs

expression ::= ...

| `match` *expression* `with` {`|` *pattern* `->` *expression*}$^{+}$

```
# let premier_element =
    match liste_a_3_elements with
    | [] -> 0
    | x :: xs -> x;;
val premier_element : int = 1
```

Les listes : filtrage de motifs

```
# let deuxieme_element =
    match liste_a_3_elements with
    | [] -> 0
    | x :: xs ->
        begin match xs with
        | [] -> 0
        | y :: ys -> y
        end;;
val deuxieme_element : int = 2
```

```
# let deuxieme_element =
    match liste_a_3_elements with
    | [] -> 0
    | x :: [] -> 0
    | x :: y :: ys -> y
  ;;
val deuxieme_element : int = 2
```

Le langage OCaml plus en détails :

---

les fonctions

Syntaxe

$$\begin{array}{rcl} \textit{expression} & ::= & \ldots \\ & \mid & \texttt{fun}\ \textit{pattern}\ \texttt{->}\ \textit{expression} \end{array}$$

Les fonctions sont des expressions

```
# let abs =
    fun x ->
      if x < 0 then -x else x;;
val abs : int -> int = <fun>
```

Il y a du « sucre syntaxique » pour définir les fonctions

```
# let abs x =
    if x < 0 then -x else x;;
val abs : int -> int = <fun>
```

Comparaison avec C

En C

```
int successeur (int x) {
  return (x + 1);
 }
```

En Caml

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

- corps = expression à rendre (pas de `return` explicite)
- type inférés : l'utilisateur n'a pas à donner le type des arguments ou du résultat
- les parenthèses ne sont pas nécessaires lors de l'appel

```
# let _ = successeur 1;;
- : int = 2
```

Comparaison avec C

Comme en C, il n'y a que des fonctions (pas de procédures) :
elles rendent toutes un résultat

```
# let r = ref 3;;
val r : int ref = {contents = 3}
# let set v =
    r := v;;
val set : int -> unit = <fun>
# let _ = set 5;;
- : unit = ()
# let _ = !r;;
- : int = 5
```

Les fonctions qui font seulement un effet de bord retournent la valeur ()

Fonction « sans argument »

```
# let reset () =
    r := 0;;
val reset : unit -> unit = <fun>
# let _ = reset ();;
- : unit = ()
```

La fonction `reset` prend la valeur de type `unit` en argument

```
# let reset2 _ =
    r := 0;;
val reset2 : 'a -> unit = <fun>
# let _ = reset2 "coucou";;
- : unit = ()
```

La fonction `reset2` prend une valeur de n'importe quel type en argument

Fonctions « à plusieurs arguments »

Les paramètres ne sont pas entre parenthèses

```
# let f x y z =
    if x then y + 1 else z - 1;;
val f : bool -> int -> int -> int = <fun>
```

Les arguments ne sont pas entre parenthèses

```
# let _ = f true 2 3;;
- : int = 3
```

Fonction qui prend un seul paramètre de type produit

```
# let f (x, y, z) =
    if x then y + 1 else z - 1;;
val f : bool * int * int -> int = <fun>
```

Fonctions « à plusieurs arguments »

Une fonction à plusieurs arguments est en fait

plusieurs fonctions à un argument

```
# let f =
    fun x ->
     (fun y ->
       (fun z -> if x then y + 1 else z - 1));;
val f : bool -> int -> int -> int = <fun>
```

Il y a aussi du sucre syntaxique pour écrire

```
# let f =
    fun x y z ->
      if x then y + 1 else z - 1;;
val f : bool -> int -> int -> int = <fun>
```

Fonctions locales

Fonction locale à une expression

```
# let _ =
    let carre n =
      n * n
    in
    carre 3 + carre 4 = carre 5;;
- : bool = true
```

Fonction locale à une fonction

```
# let pythagore x y z =
    let carre n =
      n * n
    in
    carre x + carre y = carre z;;
val pythagore : int -> int -> int -> bool = <fun>
```

Fonctions locales

```
# let fresh_number =
    let cpt = ref 0 in
    fun () ->
      cpt := !cpt + 1;
      !cpt;;
val fresh_number : unit -> int = <fun>
# let _ = fresh_number ();;
- : int = 1
# let _ = fresh_number ();;
- : int = 2
```

La valeur de la référence `cpt` ne peut être connue et modifiée que par la fonction `fresh_number`

Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle même

- il faut au moins un cas sans appel récursif si on veut que la fonction termine

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

En math : $\begin{cases} u_0 & = & 1 \\ u_n & = & n * u_{n-1} \end{cases}$

Fonctions mutuellement récursives

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

Grâce à la construction `let/and`, les fonctions `pair` et `impair` sont chacune définie en fonction de l'autre

Fonctions récursives : listes

```
# let rec somme l =
    match l with
    | [] -> 0
    | x :: xs -> x + somme xs
  ;;
val somme : int list -> int = <fun>
```

```
# let rec succ_list l =
    match l with
    | [] -> []
    | x :: xs -> (x + 1) :: succ_list xs
  ;;
val succ_list : int list -> int list = <fun>
```

Fonctions récursives terminales

```
# let rec somme_aux acc l =
    match l with
    | [] -> acc
    | x :: xs -> somme_aux (x + acc) xs;;
val somme_aux : int -> int list -> int = <fun>
# let somme l =
    somme_aux 0 l;;
val somme : int list -> int = <fun>
```

Un appel est récursif terminal s'il n'y a pas de calculs après l'appel récursif

- exécution efficace
- mémoire constante

Fonctions polymorphes

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

D'autres fonctions sur les listes

```
# let rec print_string_list l =
    match l with
    | [] -> ()
    | x :: xs -> print_string x; print_string_list xs
  ;;
val print_string_list : string list -> unit = <fun>
```

```
# let rec print_int_list l =
    match l with
    | [] -> ()
    | x :: xs -> print_int x; print_int_list xs
  ;;
val print_int_list : int list -> unit = <fun>
```

Fonctions d'ordre supérieur

Les fonctions peuvent prendre des fonctions en paramètre

```
# let rec iter f l =
    match l with
    | [] -> ()
    | x :: xs -> f x; iter f xs
  ;;
val iter : ('a -> 'b) -> 'a list -> unit = <fun>
# let print_string_list l =
    iter print_string l;;
val print_string_list : string list -> unit = <fun>
# let print_int_list l =
    iter print_int l;;
val print_int_list : int list -> unit = <fun>
```

Itérateurs : `map`

Appliquer une fonction à chaque élément d'une liste

```
# let rec map f l =
    match l with
    | [] -> []
    | x :: xs -> f x :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- `map f [ a1; ... ; an ] = [ f a1; ... ; f an ]`

```
# let succ_list l = map (fun x -> x + 1) l;;
val succ_list : int list -> int list = <fun>
```

Itérateurs : `fold_right`

Combiner les éléments d'une liste de la droite vers la gauche

```
# let rec fold_right f l acc =
    match l with
    | [] -> acc
    | x :: xs -> f x (fold_right f xs acc);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

- `fold_right f [ a1; a2; ... ; an ] acc =`
  `f a1 (f a2 ( ... (f an acc)...))`

```
# let somme l = fold_right (fun x y -> x + y) l 0;;
val somme : int list -> int = <fun>
```

Itérateurs : `fold_left`

Combiner les éléments d'une liste de la gauche vers la droite

```
# let rec fold_left f acc l =
    match l with
    | [] -> acc
    | x :: xs -> fold_left f (f acc x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

- `fold_left f [ b1; b2; ... ; bn ] acc =`
  `f ( ... (f (f acc b1) b2)...) bn`

```
# let somme_aux acc l = fold_left (+) acc l;;
val somme_aux : int -> int list -> int = <fun>
# let somme = fold_left (+) 0;;
val somme : int list -> int = <fun>
```

A faire : lire l'interface du module `List`

Application partielle

Appliquer une fonction qu'à ses premiers arguments (pas tous)

- le résultat d'un application partielle est une fonction

```
# let plus x y = x + y;;
val plus : int -> int -> int = <fun>
# let plus_1 = plus 1;;
val plus_1 : int -> int = <fun>
# let _ = plus_1 4011;;
- : int = 4012
```

Application partielle

On peut aussi faire des calculs avant de retourner une fonction

```
# let f x =
    Printf.printf "x = %d " x;
    (fun y -> Printf.printf "y = %d " y);;
val f : int -> int -> unit = <fun>
# let f1 = f 1;;
x = 1 val f1 : int -> unit = <fun>
# let _ = f1 2;;
y = 2 - : unit = ()
# let _ = f1 3;;
y = 3 - : unit = ()
# let _ = f 4 5;;
x = 4 y = 5 - : unit = ()
```

Application partielle

```
# let fresh_number_from n =
    let cpt = ref (n-1) in
    fun () -> cpt := !cpt + 1; !cpt;;
val fresh_number_from : int -> unit -> int = <fun>
# let fresh = fresh_number_from 10;;
val fresh : unit -> int = <fun>
# let _ = fresh ();;
- : int = 10
# let _ = fresh ();;
- : int = 11
# let _ = fresh_number_from 100 ();;
- : int = 100
# let _ = fresh ();;
- : int = 12
```

Différence avec les pointeurs de fonctions

```
# let f x =
    let x2 = x * x in
    fun y -> x2 + y * y;;
val f : int -> int -> int = <fun>
# let f5 = f 5;;
val f5 : int -> int = <fun>
# let f10 = f 10;;
val f10 : int -> int = <fun>
```

Les fonctions `f5` et `f10` ont chacune un état

- on parle de fermetures

Polymorphisme

```
# let f x = x;;
val f : 'a -> 'a = <fun>
# let _ = f 3;;
- : int = 3
# let _ = f true;;
- : bool = true
# let _ = f print_int;;
- : int -> unit = <fun>
# let _ = f print_int 2;;
2- : unit = ()
```

Polymorphisme

OCaml infère toujours le type le plus général possible

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

Récapitulatif

- fonctions = valeurs comme les autres
  - ▷ locales, anonymes, arguments d'autres fonctions, ...
- partiellement appliquées
- polymorphes
- l'appel de fonction ne coûte pas cher

Le langage OCaml plus en détails :

---

les types enregistrements

Types enregistrements

$$
\begin{array}{rcl}
\textit{declaration} & ::= & \ldots \\
& | & \texttt{type}\ \textit{ident}\ \texttt{=}\ \texttt{\{}\ \{\textit{label}\ \texttt{:}\ \textit{type-expression}\ \texttt{;}\}^{+}\ \texttt{\}}\ [\texttt{;}\,\texttt{;}] \\
\textit{expression} & ::= & \ldots \\
& | & \texttt{\{}\ \{\textit{label}\ \texttt{=}\ \textit{expression}\ \texttt{;}\}^{+}\ \texttt{\}}
\end{array}
$$

- Les types enregistrements sont comme les structures de C
- Il faut déclarer le type enregistrement

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

- Allocation et initialisation simultanées

```
# let x = { re = 1.0; im = -1.0; };;
val x : complexe = {re = 1.; im = -1.}
```

Types enregistrements

- Accès aux valeurs des champs d'un enregistrement
  - directement

```
# let _ = x.im;;
- : float = -1.
```

  - par filtrage

```
# let _ = match x with
          | { re = a; im = b; } -> a ;;
- : float = 1.
# let norm { re = a; im = b; } = sqrt (a *. a +. b *. b);;
val norm : complexe -> float = <fun>
# let { im = b } = x;;
val b : float = -1.
```

Types enregistrements

Déclaration de types polymorphes

```
# type ('a, 'b) paire = { gauche: 'a; droite: 'b; };;
type ('a, 'b) paire = { gauche : 'a; droite : 'b; }
# type ('a, 'b, 'c) triplet =
    { gauche: 'a; milieu: 'b; droite: 'c; };;
type ('a, 'b, 'c) triplet = { gauche : 'a; milieu : 'b; droite : 'c; }
```

```
# let _ = { gauche = 1; milieu = "a"; droite = 1.5; };;
- : (int, string, float) triplet = {gauche = 1; milieu = "a"; droite = 1.5}
```

Il ne peut pas y avoir plusieurs types enregistrements avec les mêmes noms de champs

```
# let _ = { gauche = 1; droite = 1.5; };;
Error: Some record field labels are undefined: milieu
```

Types enregistrements modifiables en place

```
# type personne = { nom: string;
                    mutable age: int; };;
type personne = { nom : string; mutable age : int; }
# let p = { nom = "Toto"; age = 23; };;
val p : personne = {nom = "Toto"; age = 23}
```

Les champs déclarés `mutable` peuvent être modifiés

```
# let _ = p.age <- p.age + 1;;
- : unit = ()
# let _ = p.age;;
- : int = 24
```

Les références

Les références sont simplement des enregistrements avec un champ mutable

```
# type 'a ref = { mutable contents: 'a};;
type 'a ref = { mutable contents : 'a; }
# let ref x = { contents = x };;
val ref : 'a -> 'a ref = <fun>
# let (!) r = r.contents;;
val ( ! ) : 'a ref -> 'a = <fun>
# let (:=) r x = r.contents <- x;;
val ( := ) : 'a ref -> 'a -> unit = <fun>
```

Le langage OCaml plus en détails :

---

les tableaux

Les tableaux

$$
\begin{array}{lcl}
\textit{expression} & ::= & \ldots \\
& | & \texttt{[|}\ \{\textit{expression}\ \texttt{;}\}^{+}\ \texttt{|]} \\
& | & \textit{expression}\ \texttt{.}\ \texttt{(}\ \textit{expression}\ \texttt{)} \\
& | & \textit{expression}\ \texttt{<-}\ \textit{expression}
\end{array}
$$

Les tableaux sont nécessairement initialisés

```
# let a = [| 1; 2; 3; 4; |];;
val a : int array = [|1; 2; 3; 4|]
```

```
# let a = Array.create 10 0;;
val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

Les tableaux

Les indices des tableaux commencent à 0

```
# let _ = a.(1) <- 1024;;
- : unit = ()
# let _ = a;;
- : int array = [|0; 1024; 0; 0; 0; 0; 0; 0; 0; 0|]
```

Les accès dans les limites des tableaux sont vérifiés dynamiquement

```
# let _ = a.(-1);;
Exception: Invalid_argument "index out of bounds".
```

Tri par insertion

```
# let tri_insertion a =
    let swap i j =
      let t = a.(i) in
      a.(i) <- a.(j); a.(j) <- t
    in
    for i = 1 to Array.length a - 1 do
      (* insérer l'élément i dans 0..i-1 *)
      let j = ref (i - 1) in
      while !j >= 0 && a.(!j) > a.(!j + 1) do
        swap !j (!j + 1);
        decr j
      done
    done;;
val tri_insertion : 'a array -> unit = <fun>
```

Le langage OCaml plus en détails :

---

les types sommes

Types sommes

$$\textit{declaration} \quad ::= \quad \ldots$$
$$\mid \quad \texttt{type}\ \textit{ident}\ \texttt{=}\ \{\texttt{|}\ \textit{u-ident}\ [\texttt{of}\ \textit{type-expression}]\}^{+}\ [\texttt{;;}]$$

Les types sommes permettent de faire l'union de plusieurs types

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

```
# type saison = Printemps | Ete | Automne | Hiver;;
type saison = Printemps | Ete | Automne | Hiver
```

Types sommes

Les types `unit` et `list` sont des types sommes

```
# type my_unit = Unit;;
type my_unit = Unit
```

```
# type 'a my_list =
    | Nil
    | Cons of 'a * 'a list;;
type 'a my_list = Nil | Cons of 'a * 'a list
```

Le type `option` est aussi prédéfini

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

Types sommes

Les types sommes sont adaptés pour définir des Arbres de Syntaxe Abstraite

```
# type formule =
    | Vrai
    | Faux
    | Conjonction of formule * formule
  ;;
type formule = Vrai | Faux | Conjonction of formule * formule
```

```
# let _ = Vrai;;
- : formule = Vrai
```

```
# let _ = Conjonction (Vrai, Faux);;
- : formule = Conjonction (Vrai, Faux)
```

Filtrage

Pour récupérer la valeur d'un type somme, il faut utiliser du filtrage

```
# let rec evalue e =
    match e with
    | Vrai -> true
    | Faux -> false
    | Conjonction (e1, e2) -> evalue e1 && evalue e2
  ;;
val evalue : formule -> bool = <fun>
```

Filtrage

les motifs de filtrage peuvent être imbriqués

```
# let rec evalue e =
    match e with
    | Vrai -> true
    | Faux -> false
    | Conjonction (Faux, e2) -> false
    | Conjonction (e1, Faux) -> false
    | Conjonction (e1, e2) -> evalue e1 && evalue e2
  ;;
val evalue : formule -> bool = <fun>
```

Filtrage

les motifs de filtrage peuvent être omis ou regroupés

```
# let rec evalue e =
    match e with
    | Vrai -> true
    | Faux -> false
    | Conjonction (Faux, _) | Conjonction (_, Faux) -> false
    | Conjonction (e1, e2) -> evalue e1 && evalue e2
  ;;
val evalue : formule -> bool = <fun>
```

Le langage OCaml plus en détails :

---

les exceptions

Exceptions

$$
\begin{array}{rcl}
\textit{declaration} & ::= & \ldots \\
& | & \texttt{exception}\ \textit{u-ident}\ [\texttt{of}\ \textit{type-expression}]\ [\texttt{;;}] \\
\textit{exception} & ::= & \ldots \\
& | & \texttt{raise}\ \textit{expression} \\
& | & \texttt{try}\ \textit{expression}\ \texttt{with}\ \{\texttt{|}\ \textit{pattern}\ \texttt{->}\ \textit{expression}\}^{+}
\end{array}
$$

- Interrompre l'exécution d'une fonction s'il n'y a pas de réponse pertinente à donner
- Trois actions
  - déclarer une exception
  - déclencher, lever, signaler une exception
  - traiter, rattraper une exception

Exceptions

```
# exception Liste_vide;;
exception Liste_vide
```

```
# let hd l =
    match l with
    | [] -> raise Liste_vide
    | x :: _ -> x;;
val hd : 'a list -> 'a = <fun>
```

- Il n'y a pas de « bonne » valeur à rendre si la liste est vide
- `raise` ne casse pas le polymorphisme

```
# let  _ = raise ;;
- : exn -> 'a = <fun>
```

Exceptions

```
# exception Non_trouve;;
exception Non_trouve
# let rec assoc x l =
    match l with
    | [] -> raise Non_trouve
    | (y, v) :: l' -> if x = y then v else assoc x l'
  ;;
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
# let assoc_or_0 x l =
    try assoc x l
    with Non_trouve -> 0;;
val assoc_or_0 : 'a -> ('a * int) list -> int = <fun>
```

Exceptions prédéfinies

Quelques exceptions prédéfinies :

```
exception Division_by_zero
exception Invalid_argument of string
exception Failure of string
exception Assert_failure of (string * int * int)
exception Not_found
```

Lever des exceptions prédéfinies

```
# let _ = 1/0;;
Exception: Division_by_zero.
# let _ = failwith "ne doit pas se produire";;
Exception: Failure "ne doit pas se produire".
```

Assertions

`assert` permet de faire des vérifications dynamique de propriétés

```
# let racine x =
    assert (x >= 0.0);
    sqrt x;;
val racine : float -> float = <fun>
```

et d'identifier des cas absurdes

```
# let _ =
    try List.assoc 1 [ (4, "a"); (0, "b"); (1, "c"); (2, "c"); ]
    with Not_found -> assert false;;
- : string = "c"
```

Modules et Foncteurs

Génie logiciel

Lorsque les programmes deviennent trop gros il faut

- découper un unités : modularité
- occulter le représentation de certaines données : encapsulation
- éviter au mieux la duplication de code

En OCaml : fonctionnalités apportées par les modules

Fichiers et modules

Chaque fichier est un module

Si `arith.ml` contient

```
let pi = 3.141592;;
let round x = floor (x +. 0.5);;
```

alors on le compie avec

```
--> ocamlc -c arith.ml
```

Utilisation dans un autre module `main.ml` :

```
let x = float_of_string (read_line ());;
let _ = Printf.printf "%f \n" (Arith.round (x /. Arith.pi));;
```

```
--> ocamlc -c main.ml
--> ocamlc -o test arith.cmo main.cmo
```

Encapsulation

On peut restriendre les valeurs exportées avec une interface

Dans le fichier `arith.mli` :

```
  val round: float -> float
```

```
--> ocamlc -c arith.mli
```

```
--> ocamlc -c main.ml
File "main.ml", line 2, characters 49-57:
Error: Unbound value Arith.pi
```

Encapsulation

Une interface peut restreindre la visibilité de la définition d'un type

Dans le fichier `ensemble.ml` :

```
type t = int list
let ajoute x l = x :: l
let appartient = List.mem
```

Dans le fichier `ensemble.mli` :

```
type t
val ajoute: int -> t -> t
val appartient: int -> t -> bool
```

Le type `t` est un type abstrait

Compilation séparée

La compilation d'un fichier ne dépend que des interfaces des fichiers utilisés

- moins de recompilation quand le code change mais pas son interface

Langage de modules

Les modules ne sont pas restreints aux fichiers

```
# module M = struct
    let c = 100
    let f x = c * x
  end;;
module M : sig val c : int val f : int -> int end
```

Langage de modules

On peut définir des modules dans des modules

```
# module A = struct
    let a = 2
    module B = struct
      let b = 3
      let f x = a * b * x
    end
    let f x = B.f (x + 1)
  end;;
```

```
module A :
  sig
    val a : int
    module B : sig val b : int val f : int -> int end
    val f : int -> int
  end
```

Langage de modules

On peut également définir des signatures de modules

```
# module type S = sig
    val f: int -> int
  end;;
module type S = sig val f : int -> int end
```

On peut mettre une contrainte sur le type du module

```
# module M : S = struct
    let a = 2
    let f x = a * x
  end;;
module M : S
# let _ = M.a;;
Error: Unbound value M.a
```

Récapitulation

- modularité par découpage du code en unité appelées modules
- encapsulation de types et de valeurs : types abstraits
- vraie compilation séparée
- organisation de l'espace de nommage

Foncteurs

Foncteur = module paramétré par un ou plusieurs modules

Exemple : table de hachage générique

- il faut paramétrer par rapport à
  - la fonction de hachage
  - la fonction d'égalité

Table de hachage générique : première solution

Passer les fonctions de hachage et d'égalité en argument :

```
type 'a t
val create: int -> 'a t
val add: ('a -> int) -> 'a t -> 'a -> unit
val mem:
  ('a -> int) -> ('a -> 'a -> bool) -> 'a t -> 'a -> bool
```

Table de hachage générique : deuxième solution

Passer les fonctions de hachage et d'égalité à la création :

```
type 'a t
val create: ('a -> int) -> ('a -> 'a -> bool) -> int -> 'a t
val add: 'a t -> 'a -> unit
val mem: 'a t -> 'a -> bool
```

Implantation :

```
# type 'a t =
    { hash: 'a -> int;
      eq: 'a -> 'a -> bool;
      data: 'a list array; };;
```

```
# let create h eq n =
    { hash = h; eq = eq; data = Array.create n []; };;
```

Table de hachage générique : « la bonne solution »

```
# module type S = sig
    type elt
    val hash: elt -> int
    val eq: elt -> elt -> bool
  end;;
```

```
# module type H = functor (X: S) ->
    sig
      type t
      val create: int -> t
      val add: t -> X.elt -> unit
      val mem: t -> X.elt -> bool
    end;;
```

Table de hachage générique : « la bonne solution »

```
# module H (X: S) = struct
    type t = X.elt list array
    let create n = Array.create n []
    let add t x =
      let i = (X.hash x) mod (Array.length t) in
      t.(i) <- x :: t.(i)
    let mem t x =
      let i = (X.hash x) mod (Array.length t) in
      List.exists (X.eq x) t.(i)
  end;;
```

Foncteurs : utilisation

```
# module Entiers = struct
    type elt = int
    let hash x = abs x
    let eq x y = x = y
  end;;
module Entiers :
  sig type elt = int val hash : int -> int val eq : 'a -> 'a -> bool end
# module Hentiers = H(Entiers);;
module Hentiers :
  sig
    type t = Entiers.elt list array
    val create : int -> 'a list array
    val add : Entiers.elt list array -> Entiers.elt -> unit
    val mem : Entiers.elt list array -> Entiers.elt -> bool
  end
```

Foncteurs : application

- Structures de données paramétrées par d'autres structures de données
  - `Hashtbl.Make` : table de hachage
  - `Set.Make` : ensemble finis codés par des arbres binaires équilibrés
  - `Map.Make` : tables d'association codées par des arbres binaires équilibrés
- Algorithmes paramétrés par des structures de données
  - exemple : algorithme de Dijkstra de recherche de plus court chemin écrit indépendamment de la structure de graphes

Algorithme de Dijkstra « générique »

```
# module type Dijkstra =
    functor
      (G : sig
             type graph
             type sommet
             val voisins: graph -> sommet -> (sommet * int) list
           end) ->
    sig
      val plus_court_chemin:
        G.graph -> G.sommet -> G.sommet -> G.sommet list * int
    end;;
```

Exemple : évaluateur d'expressions arithmétiques

Syntaxe abstraite des expressions

On définit le type `expr` suivant :

```
# type expr =
    | Cst of int
    | Sum of expr * expr
    | Diff of expr * expr
    | Prod of expr * expr
    | Quot of expr * expr
  ;;
```

On souhaite maintenant écrire une fonction d'évaluation

```
  val eval_expr: expr -> int
```

Code de la fonction `eval_expr`

La fonction est définie par récurrence sur la structure des expressions

- fonction récursive définie par filtrage sur l'expression à évaluer

```
# let rec eval_expr e =
    match e with
    | Cst x -> x
    | Sum (e1, e2) -> (eval_expr e1) + (eval_expr e2)
    | Diff (e1, e2) -> (eval_expr e1) - (eval_expr e2)
    | Prod (e1, e2) -> (eval_expr e1) * (eval_expr e2)
    | Quot (e1, e2) -> (eval_expr e1) / (eval_expr e2)
  ;;
val eval_expr : expr -> int = <fun>
```

Extension de la syntaxe abstraite

On étend maintenant nos expressions arithmétiques avec des variables locales pour stocker les résultats intermédiaires

```
# type expr =
    | Cst of int
    | Sum of expr * expr
    | Diff of expr * expr
    | Prod of expr * expr
    | Quot of expr * expr
    | Var of string
    | Letin of string * expr *expr
  ;;
```

Il faut alors ajouter un environnement à la fonction `eval_expr`

```
  val eval_expr: (string * int) list -> expr -> int
```

Code de la fonction `eval_expr`

```
# let rec eval_expr env e =
    match e with
    | Cst x -> x
    | Var x -> List.assoc x env
    | Sum (e1, e2) -> (eval_expr env e1) + (eval_expr env e2)
    | Diff (e1, e2) -> (eval_expr env e1) - (eval_expr env e2)
    | Prod (e1, e2) -> (eval_expr env e1) * (eval_expr env e2)
    | Quot (e1, e2) -> (eval_expr env e1) / (eval_expr env e2)
    | Letin (x, e1, e2) ->
        let v = eval_expr env e1 in
        eval_expr ((x, v) :: env) e2
  ;;
val eval_expr : (string * int) list -> expr -> int = <fun>
```

Gestion des erreurs

```
# exception Varundef of string;;
```

```
# let rec eval_expr env e =
    match e with
    | Cst x -> x
    | Var x ->
       (try List.assoc x env
        with Not_found -> raise (Varundef x))
    | Sum (e1, e2) -> (eval_expr env e1) + (eval_expr env e2)
    | Diff (e1, e2) -> (eval_expr env e1) - (eval_expr env e2)
    | Prod (e1, e2) -> (eval_expr env e1) * (eval_expr env e2)
    | Quot (e1, e2) -> (eval_expr env e1) / (eval_expr env e2)
    | Letin (x, e1, e2) ->
       let v = eval_expr env e1 in
       eval_expr ((x, v) :: env) e2 ;;
```

Instructions

On ajoute des instructions pour manipuler des variables globales

```
# type instr =
    | Set of string * expr
    | Unset of string
    | Print of expr
  ;;
```

Ces variables serons stockées dans un environnement `genv` de type :

```
val genv : (string * int) list ref
```

Il ne reste plus qu'à écrire une fonction d'évaluation `eval_instr` :

```
val eval_instr : instr -> unit
```

Code de la fonction `eval_instr`

```
# let genv = ref [];;
val genv : '_a list ref = {contents = []}
# let eval_instr i =
    match i with
    | Set (x, e) -> genv := (x, eval_expr [] e) :: !genv
    | Unset x -> genv := List.remove_assoc x !genv
    | Print e -> Printf.printf "= %d" (eval_expr [] e)
  ;;
val eval_instr : instr -> unit = <fun>
```

Code de la fonction `eval_expr`

```
# let rec eval_expr env e =
    match e with
    | Cst x -> x
    | Var x ->
       (try List.assoc x env
        with Not_found ->
          (try List.assoc x !genv
           with Not_found -> raise (Varundef x)))
    | Sum (e1, e2) -> (eval_expr env e1) + (eval_expr env e2)
    | Diff (e1, e2) -> (eval_expr env e1) - (eval_expr env e2)
    | Prod (e1, e2) -> (eval_expr env e1) * (eval_expr env e2)
    | Quot (e1, e2) -> (eval_expr env e1) / (eval_expr env e2)
    | Letin (x, e1, e2) ->
        let v = eval_expr env e1 in eval_expr ((x, v) :: env) e2 ;;
val eval_expr : (string * int) list -> expr -> int = <fun>
```

Evaluateur modulaire

On peut rendre cet évaluateur indépendant du choix d'implantation de l'environnement global à l'aide d'un foncteur

La signature `GlobalEnv` définit l'interface d'un module implantant un environnement de variables globales pour des expressions arithmétiques entières

```
# module type GlobalEnv = sig
    type t
    val nouveau: int -> t
    val ajoute: t -> string * int -> unit
    val trouve: t -> string -> int
    val supprime: t -> string -> unit
  end;;
```

Code de l'évaluateur

```
# module Eval (Genv : GlobalEnv) = struct
    let genv = Genv.nouveau 31
    exception Varundef of string
    let rec eval_expr env e =
      match e with
      | Cst x -> x
      | Var x ->
         (try List.assoc x env with Not_found ->
            (try Genv.trouve genv x
             with Not_found -> raise (Varundef x)))
      | Sum (e1, e2) -> (eval_expr env e1) + (eval_expr env e2)
      | Diff (e1, e2) -> (eval_expr env e1) - (eval_expr env e2)
      | Prod (e1, e2) -> (eval_expr env e1) * (eval_expr env e2)
      | Quot (e1, e2) -> (eval_expr env e1) / (eval_expr env e2)
      | Letin (x, e1, e2) ->
          let v = eval_expr env e1 in eval_expr ((x, v) :: env) e2
```

```
    let eval_instr i =
      match i with
      | Set (x, e) -> Genv.ajoute genv (x, eval_expr [] e)
      | Unset x -> Genv.supprime genv x
      | Print e -> Printf.printf "= %d" (eval_expr [] e)
  end;;
module Eval :
  functor (Genv : GlobalEnv) ->
    sig
      val genv : Genv.t
      exception Varundef of string
      val eval_expr : (string * int) list -> expr -> int
      val eval_instr : instr -> unit
    end
```

Une implantation de `GlobalEnv` avec des listes

```
# module AssocEnv : GlobalEnv = struct
    type t = (string * int) list ref
    let nouveau _ = ref []
    let trouve t s = List.assoc s !t
    let ajoute t e = t := e :: !t
    let supprime t s = t := List.remove_assoc s !t
  end;;
module AssocEnv : GlobalEnv
```

Une implantation de `GlobalEnv` avec des tables de hachage

```
# module HashEnv : GlobalEnv = struct
    type t = (string, int) Hashtbl.t
    let nouveau = Hashtbl.create
    let trouve = Hashtbl.find
    let ajoute t (x, v) = Hashtbl.add t x v
    let supprime = Hashtbl.remove
  end;;
module HashEnv : GlobalEnv
```

Quelques points forts du langage Ocaml

- Le langage repose sur un ensemble réduit de concepts
- Facilité de définir des structures de données
- Filtrage de motifs
- Inférence de types
- Modularité : polymorphisme, ordre supérieur, foncteurs
- Gestion automatique de la mémoire : Glaneur de Cellules
- Code compilé efficace