Langage OCaml

Table des matières

1 Introduction

1.1 Présentation du langage OCaml

Le langage OCaml a été développé à l'INRIA (Institut National de Recherche en Informatique et Automatique) à partir des années 80. Sans faire d'historique ni de typologie détaillée des langages de programmation, on peut donner rapidement quelques caractéristiques de OCaml. Il s'agit d'un langage :

- *compilé* : les fichiers source doivent d'abord être traduits en langage machine avant d'être exécutés, comme en C, C++, Java. Python, au contraire, est *interprété* : le code est (plus ou moins) lu ligne par ligne à l'exécution par un programme appelé *interpréteur*. Comme OCaml est dotée d'une REPL (*Read Eval Print Loop*, environnement « interactif » comme celui dont on dispose pour Python) que nous utiliserons systématiquement en TD, nous ne verrons pas trop la différence.
- *typé statiquement* : une variable ne peut pas changer de type au cours de l'exécution. Python est typé dynamiquement, ce qui est assez caractéristique des langages de script (Python, Ruby, Perl, JavaScript...) ; la plupart des langages compilés (C, Java, C++) sont typés statiquement, avec quelques subtilités ;
- *fortement typé*, c'est-à-dire que toutes les erreurs de type seront détectées à la compilation. En Python, ces erreurs seront détectées à l'exécution ; en C, elles ne seront pas forcément détectées.
- *fonctionnel*, c'est-à-dire que les fonctions sont des valeurs comme les autres, qui peuvent être renvoyées comme résultat d'un calcul, prises en argument... C'est également possible en Python, mais c'est moins pratique et moins « idiomatique ». Un langage fonctionnel met également l'accent sur la récursion plus que sur l'itération et privilégie les valeurs non mutables (essentiellement, fonctions et variables sont utilisées d'une manière proche de ce que l'on fait en mathématiques et pas comme des cases mémoire dans lesquelles on écrit à notre gré). Python n'est pas du tout fonctionnel en ce sens là, les exemples classiques sont plutôt LISP (dans une certaine mesure), Standard ML (très proche de OCaml) et Haskell.
- *impur* : la mutation (modification de la valeur d'une variable) est quand même possible dans certains cas. Haskell est l'exemple le plus connu de langage (presque) pur. Notez quand même qu'en comparaison de Python, OCaml est une vestale...
- *strict* : les arguments d'une fonction sont évalués avant de faire l'appel. Si vous n'avez jamais fait de Haskell ni de λ-calcul, vous n'avez sûrement jamais réalisé que l'on pouvait faire autrement. Haskell et son ancêtre Miranda sont les seuls langages un tant soit peu connus à être *paresseux* (le contraire de strict dans ce contexte).

— *orienté objet*, ou plutôt « doté d'une système d'objets dont quasiment personne ne se sert » : c'est de là que vient le « O » de OCaml. Dans la plupart des langages actuellement populaires, les objets jouent un rôle important (Python, C++, C#, Java, etc), dans les autres, la notion n'existe pas (C, Haskell).

Caml est un langage « presque confidentiel mais pas tout-à-fait » : il est principalement utilisé dans l'univers de la recherche et pour créer des outils manipulant des programmes éventuellement écrits dans un autre langage (compilateurs, analyse statique, génération automatique de code, etc). On peut citer deux variantes assez populaires de Caml :

— F#, qui a été créé par Microsoft et fait partie de l'environnement `.Net` : il y a quelques différences par rapport à OCaml mais il est immédiat d'apprendre l'un si l'on connaît l'autre ;

— Reason, créé très récemment par Facebook. Ici, on ne peut même pas vraiment parler de variante : le langage est exactement le même et utilise le même compilateur, seule la syntaxe a été modifiée de manière à être moins déroutante pour des développeurs habitués à JavaScript ou PHP, qui sont les langages les plus utilisés chez Facebook.

1.2 Opérateurs de base

Les types les plus simples en Caml sont `int` (type des entiers), `float` (type des nombres à virgule flottante, l'« équivalent » informatique des réels), `bool` (les booléens, c'est-à-dire `true` et `false`), `char` (type des caractères) et `string` (type des chaînes de caractères).

On dispose des opérateurs arithmétiques usuels sur les entiers :

```
# 3 + 4;;
- : int = 7
# 12 - 3 * 7;;
- : int = -9
# 5 / 2 ;;
- : int = 2
# 17 mod 5;;
- : int = 2
```

et sur les flottants :

```
# 3.1 +. 5.4;;
- : float = 8.5
# 5. /. 2. ;;
- : float = 2.5
# 2. ** 8. ;; (* puissance, défini uniquement sur les flottants *)
- : float = 256.
```

Il faut bien remarquer qu'il y a deux variantes distinctes des opérateurs, l'une pour les entiers et l'autre pour les flottants. Si l'on essaie de mélanger les deux types, on obtiendra une erreur :

```
# 2.5 + 3;;
Characters 0-3:
2.5 + 3;;
^^^
Error: This expression has type float but an expression was expected of type
int
# 2.5 +. 3;;
Characters 7-8:
2.5 +. 3;;
       ^
Error: This expression has type int but an expression was expected of type
float
```

Les opérateurs de comparaison sont, eux, *polymorphes* : on utilise le même opérateur pour comparer deux entiers, deux flottants...[1]

```
# 2 < 3;;
- : bool = true
# 3.4 >= 2.5;;
- : bool = true
# (4, 8) = (4, 9);;
- : bool = false
```

1. mais pas un entier et un flottant.

```
# (4, 8) <> (4, 9);;
- : bool = true
```

Sur les booléens, les opérations de base sont le « et », le « ou » et la négation :

```
# true && false;; (* "et" logique *)
- : bool = false
# true || false;; (* "ou" logique *)
- : bool = true
# not false;; (* négation *)
- : bool = true
```

Tout texte compris entre `(*` et `*)` est ignoré : on parle de *commentaires*.

1.3 Variables

On peut définir des variables *globales* (qui existent dans l'ensemble du programme) :

```
# let x = 3;;
val x : int = 3
# x + x;;
- : int = 6
# let y = x;;
val y : int = 3
# let x = 4;;
val x : int = 4
# y;;
- : int = 3
```

et également des variables *locales* (dont la portée est limitée) :

```
# let a = 4 in a * a + 1;;
- : int = 17
# a;;
Characters 0-1:
a;;
^
Error: Unbound value a
```

Une variable locale peut *masquer* une variable globale :

```
# let a = 5;;
val a : int = 5
# let a = 4 in a * a;;
- : int = 16
# a;;
- : int = 5
```

Très souvent, on enchaînera les définitions de variables locales (mais normalement ce sera à l'intérieur d'une définition de fonction, pas dans la boucle interactive) :

```
# let a = 4 in let b = 7 in a * b;;
- : int = 28
# let a = 4 in let b = a * a in a + b;;
- : int = 20
```

En OCaml, une variable « normale » ne *change jamais de valeur* une fois définie (on dit qu'elle n'est *pas mutable*). C'est assez différent de ce qui se passe dans la plupart des langages, mais très similaire à la notion mathématique de variable [2]. Nous aurons l'occasion de revenir largement sur ce point dans la suite du cours.

1.4 Définitions de fonctions

Un exemple minimal :

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

2. « Let x = ... » est la traduction anglaise de « Soit $x = \ldots$ », et ce choix n'a pas été fait par hasard.

Comme l'opérateur `*` ne marche que sur les entiers, Caml sait que `x` doit être un entier, et comme le résultat de la multiplication de deux entiers est un entier, Caml sait que le résultat sera un entier. Il en déduit le type de la fonction : `int -> int`. On dit que Caml a *inféré le type* de la fonction `carre`.

On notera immédiatement une différence importante avec Python : il n'y a pas de `return`. En Caml, tout est une expression, et en particulier la valeur de retour d'une fonction est simplement la valeur de l'expression définissant la fonction une fois que l'on a substitué les valeurs concrètes des arguments formels.

En Caml, l'application de la fonction f à l'argument x se note simplement `f x` (et non $f(x)$ comme en mathématiques). Si l'on veut appeler cette fonction, il suffit donc de taper

```
# carre 10;;
- : int = 100
```

Attention aux priorités :

```
# carre 3 + 5;;
- : int = 14
# carre (3 + 5);;
- : int = 64
```

En Caml, l'application de fonction est prioritaire sur tout le reste : `f x y` signifie « $f(x)$, le tout appliqué à y ».

Quand une fonction prend plusieurs arguments, la technique « naturelle » en Caml est d'en écrire une version *curryfiée* (rien à voir avec les épices : c'est en référence à Haskell Curry, logicien, qui a également donné son nom au langage Haskell) :

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

Si l'on veut ajouter le carré de 4 et celui de $2 + 3$ (en utilisant les fonctions que nous venons de définir), il faut alors écrire :

```
# ajoute (carre 4) (carre (2 + 3));;
- : int = 41
```

On peut définir des variables locales à l'intérieur d'une fonction (c'est même le principal intérêt des variables locales...) :

```
# let f a b =
    let x = carre (a + b) in
    let y = carre (a - b) in
    carre (x + y) - carre (x - y);;
val f : int -> int -> int = <fun>
# f 3 7;;
- : int = 6400
```

1.5 Fonctions récursives

En informatique, une fonction est dite *récursive* si elle s'appelle elle-même. Il y a beaucoup de choses à dire sur la récursivité, et nous y reviendrons longuement, mais pour aujourd'hui on peut se contenter de quelques exemples très simples.

On rappelle la définition classique de la factorielle :

$$0! = 1 \qquad \text{et} \qquad \forall n \in \mathbb{N},\ (n+1)! = (n+1) \times n!$$

En mathématiques, on dirait qu'il s'agit d'une définition *par récurrence* ; en informatique, on parlera plutôt de *définition inductive*, voire *récursive*.

Pour changer temporairement de langage, il y a en Python deux manières naturelles d'écrire une fonction « factorielle » :

```
1 def fact_iter(n):
2     f = 1
3     for k in range(1, n + 1):
4         f = f * k
5     return f
6
7 def fact_rec(n):
```

```
    if n == 0:
        return 1
    return n * fact_rec(n - 1)
```

La première version est dite *itérative* : elle utilise une boucle (et ce qui serait en Caml une variable *mutable* `f`). La deuxième version, très proche de la définition mathématique, est dite récursive.

Ces deux versions peuvent être traduites en Caml, mais la première, qui utilise les aspects impératifs de Caml, ne nous intéresse pas pour l'instant.

```
(* Vous pouvez sauter cette partie pour l'instant. *)
let fact_iter n =
  let f = ref 1 in
  for k = 1 to n do
    f := !f * k
  done;
  !f
(* Fin de la partie à ignorer. *)

let rec fact_rec n =
  if n = 0 then 1
  else n * fact_rec (n - 1)
```

On notera qu'en Caml, une définition récursive utilise `let rec` au lieu de `let`.

1.6 Listes

Le type `list` est un type de base en OCaml. Il permet de représenter des listes de valeurs de type arbitraire mais homogène. Cela signifie que l'on peut définir une liste d'entiers ou une liste de booléens, mais pas une liste contenant à la fois des entiers et des booléens.

Attention : le type `list` de OCaml est *très* différent du type `list` de Python. Ce que l'on appelle liste en Python ressemble plutôt à ce que l'on appellerait tableau en OCaml (et dans la plupart des langages).

On peut définir une liste en mettant les éléments entre crochets et en les séparant par des points-virgules :

```
# let l = [];;
val l : 'a list = []
# let m = [1];;
val m : int list = [1]
# let s = [3; 5; 0];;
val s : int list = [3; 5; 0]
```

Étant donnée une liste, on peut en construire une nouvelle en ajoutant un élément en tête :

```
# let k = 3 :: l ;;
val k : int list = [3]
# let j = 7 :: k ;;
val j : int list = [7; 3]
```

Tous les éléments d'une liste doivent avoir le même type (uniquement des entiers, ou alors uniquement des flottants, ou alors…) :

```
# let w = [3.2; 5.1];;
val w : float list = [3.2; 5.1]
# let t = [3.2 ; 5];;
Characters 15-16:
let t = [3.2 ; 5];;
                ^
Error: This expression has type int but an expression was expected of type
float
```

Essentiellement, il y a deux types de listes : la liste vide, et les autres. Une liste non vide est constituée d'un premier élément, appelé *tête* ou *head* et de la liste (éventuellement vide) de ses autres éléments, appelée *queue* ou *tail*. Le *pattern matching* (qu'on traduit parfois en français par *filtrage*, ou *filtrage par motif*) est la manière standard d'opérer sur des listes :

```
let rec somme l =
  match l with
  | [] -> 0
  | x :: xs -> x + somme xs
```

On voit ici le schéma fondamental de pattern matching sur les listes : soit la liste est vide (c'est ici, comme souvent, le cas de base de la récursion), soit elle s'écrit `head :: tail`, avec `head` un élément de type `'a` et `tail` une liste de type `'a list`.

Il arrive quand même parfois que ces deux cas ne suffisent pas. Ici, on souhaite calculer le maximum des éléments d'une liste, en utilisant la fonction prédéfinie `max` (qui permet d'obtenir le maximum de deux nombres).

```
let rec max_liste l =
  match l with
  | [] -> failwith "liste vide"
  | [x] -> x
  | x :: xs -> max x (max_liste xs)
```

Dans ce filtrage, les motifs ne sont pas mutuellement exclusifs : une liste de la forme `[x]` est également de la forme `x :: xs` avec `xs = []`. Cependant, les motifs sont essayés dans l'ordre : ainsi, une liste à un seul élément sera acceptée par la deuxième clause et n'arrivera pas jusqu'à la troisième. En revanche, une liste contenant au moins deux éléments sera refusée par les deux premiers motifs et sera donc traitée par la troisième clause.

2 OCaml Fonctionnel

2.1 Types de base

2.1.1 Types numériques

Type int Le type `int` est celui des entiers (*integer* en anglais).

Opérateur	Signification	Exemples
`+`	Addition d'entiers	`2 + 3 = 5`
`*`	Multiplication d'entiers	`2 * 3 = 6`
`-`	Soustraction / opposé	`3 - 7 = -4` ou `-(2 + 3) = -5`
`/`	« Quotient » de la division euclidienne	`17 / 5 = 3` `(-17) / 5 = -3`
`mod`	« Reste » de la division euclidienne	`17 mod 5 = 2` `-17 mod 5 = -2`
`abs`	Valeur absolue	`abs (-2) = 2`

Remarque

⇨ Pour la division, `a / b` fait un arrondi vers 0, et ne correspond donc à la division euclidienne mathématique que si a et b sont de même signe. `a mod b` est défini de telle manière que l'identité `a = (a / b) * b + (a mod b)` soit systématiquement vérifiée, ce qui signifie que `a mod b` est négatif si `a` est négatif.

La plupart de ces opérateurs sont *infixes* (il faut les placer entre leurs deux arguments). Cependant, l'addition par exemple est en fait une fonction de type `int -> int -> int`. On peut y accéder en plaçant l'opérateur entre parenthèses.

```
# ( + );;
- : int -> int -> int = <fun>
# ( + ) 12 23;;
- : int = 35
```

Les entiers sont représentés sur 64 bits sur une machine 64 bits, mais OCaml utilise un de ces bits comme un « drapeau » : les entiers représentables varient donc entre -2^{62} et $2^{62}-1$ sur une machine 64 bits. Ces valeurs sont stockées dans les constantes `min_int` et `max_int`.

```
# min_int;;
- : int = -4611686018427387904
# max_int;;
- : int = 4611686018427387903
# 1 lsl 62 - 1 (* 2^62 - 1 *);;
- : int = 4611686018427387903
```

Tous les calculs sont effectués modulo 2^{63}.

```
# max_int + 1 = min_int;;
- : bool = true
```

Type float Le type des nombres dits *à virgule flottante* (ou flottants pour faire simple), représentés sur 64 bits. La représentation interne d'un flottant a été vue en informatique de tronc commun.

Opérateur	Signification	Exemples
`+.`	Addition de flottants	`2.1 +. 3. = 5.1`
`*.`	Multiplication de flottants	`2.1 *. 3. = 6.2`
`-.`	Soustraction / opposé	`3. -. 7.1 = -4.1` `-.(2. +. 3.2) = -5.2`
`/.`	Division de flottants	`17. /. 5. = 3.4`
`**`	Puissance	`2.3 ** 4.1 = 30.41...`
`exp`	Exponentielle	`exp 1. = 2.718...`
`log`	Logarithme népérien	`log 2. = 0.69...`
`floor`	« Partie entière »	`floor (-2.3) = -3.`
`ceil`	Arrondi par excès	`ceil 3.1 = 4.`
`abs_float`	Valeur absolue	`abs_float (-3.2) = 3.2`

Attention, le résultat de la « partie entière » est un nombre flottant. Les fonctions trigonométriques usuelles (directes, inverses, hyperboliques) sont aussi fournies.

Contrairement à la plupart des langages, OCaml ne fait jamais de conversion automatique des entiers vers les flottants, ni évidemment des flottants vers les entiers. Les fonctions de conversion de type, aussi appelées *coercitions*, sont :

Fonction	Type	Comportement	Exemples
`float_of_int`	`int -> float`	Injection canonique	`float_of_int 4 = 4.`
`int_of_float`	`float -> int`	Arrondi vers 0	`int_of_float 3.7 = 3` `int_of_float (-3.7) = -3`

2.1.2 Booléens

Le type booléen ne contient que deux valeurs : `true` et `false`.

Opérateur	Signification	Exemples
`not`	Négation	`not true = false`
`&&`	« Et » séquentiel	`true && false = false`
`||`	« Ou » séquentiel	`true || false = true`

`&&` et `||` sont *paresseux* (ou *séquentiels*), ce qui signifie qu'ils n'évaluent leur second argument que si nécessaire. Ainsi, `true || x` renvoie `true` immédiatement sans évaluer l'expression `x`, ce qui peut être important si déterminer la valeur de `x` demande un calcul long, et crucial si le calcul de `x` ne termine pas ou résulte en une erreur.

```
# (2 < 3) || (1 / 0 = 0);;
- : bool = true
# (2 < 3) && (1 / 0 = 0);;
Exception: Division_by_zero.
# (1 / 0 = 0) || (2 <> 3);;
Exception: Division_by_zero.
# (2 > 3) || (1 / 0 = 0);;
Exception: Division_by_zero.
# (2 > 3) && (1 / 0 = 0);;
- : bool = false
```

On dispose également d'opérateurs de comparaison polymorphes. Ils permettent de comparer deux valeurs d'un même type, quel que soit ce type.

Opérateur	Signification	Exemples
`=`	Égalité structurelle	`[[3]; [1; 2]] = [[3]; [1; 2]]` renvoie `true`
`<>`	Différence structurelle	`[[3]; [1; 2]] <> [[3]; [1; 4]]` renvoie `true`
`<`	Inférieur strict	`[[3]; [1; 2]] < [[3]; [1; 4]]` donne `true`
`<=`	Inférieur ou égal	`[[3]; [1; 2]] <= [[4]; [1; 1]]` donne `true`

Tester l'égalité structurelle de deux valeurs simples (entiers, flottants, etc) se fait en temps constant. Ce n'est bien sûr plus le cas si, par exemple, on compare deux listes : on peut avoir à parcourir toute la structure.

Remarque

⇨ *Attention*, les opérateurs `==` et `!=` existent également, mais leur sens est différent. Ils testent l'égalité *physique*, c'est-à-dire égalité des pointeurs. Nous ne nous en servirons pour ainsi dire jamais.

2.1.3 Caractères et chaînes de caractères

En OCaml, les types caractère (`char`) et chaîne de caractère (`string`) sont deux types différents. Une chaîne de caractère se comporte essentiellement comme un tableau de `char`, excepté qu'une chaîne n'est pas mutable. Pour définir une constante littérale de type `char`, on mettra le caractère entre apostrophes, et pour une constante de type `string` entre guillemets.

```
# 'a';;
- : char = 'a'
# "a";;
- : string = "a"
# "abc";;
- : string = "abc"
```

Les fonctions d'affichage sont par essence impératives, mais si vous en avez besoin :

Fonction	Signification	Exemples
`print_string`	Afficher une chaîne	`print_string "abc"` affiche `"abc"`
`print_int`	Afficher un entier	`print_int (2 + 3)` affiche `"5"`
`print_float`	Affiche un flottant	`print_float (acos (-1.))` affiche `3.14159265359`

On dispose également de `int_of_string`, `string_of_int`, `float_of_string` et `string_of_float` :

```
# float_of_string "2.3e5";;
- : float = 230000.
# string_of_float (2.**10.);;
- : string = "1024."
# int_of_string "12.34";;
Exception: Failure "int_of_string".
```

2.1.4 Type unit

Le type `unit` est essentiellement celui des *actions*. Il n'y a qu'une valeur de ce type : `()`. Nous reviendrons en détail sur ce sujet lorsque nous évoquerons les aspects impératifs de OCaml.

2.2 Types composites

2.2.1 Type liste

Nous avons déjà utilisé le type `list`, et nous reviendrons dessus de manière plus poussée ultérieurement. Voilà quand même une table de référence pour quelques fonctions usuelles.

Fonction	Type	Signification
`@` (infixe)	`'a list -> 'a list -> 'a list`	Concaténation
`List.hd`	`'a list -> 'a`	Renvoie l'élément de tête
`List.tl`	`'a list -> 'a list`	Renvoie la queue
`List.rev`	`'a list -> 'a list`	Renvoie la liste « miroir »
`List.length`	`'a list -> int`	Renvoie la longueur de la liste
`List.map`	`('a -> 'b) -> 'a list -> 'b list`	*cf* TD
`List.fold_left`	`('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`	*cf* TD
`List.fold_right`	`('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`	*cf* TD
`List.iter`	`('a -> unit) -> 'a list -> unit`	*cf* TD

Remarques

- ⇨ Nous n'avons pas mentionné « `::` » car il s'agit d'un constructeur et non d'une fonction. C'est cependant, et de loin, ce qu'on utilise le plus souvent.
- ⇨ Notez bien que les fonctions `List.hd` et `List.tl` sont très rarement utiles : on préférera le pattern matching.
- ⇨ Attention, la fonction `List.length` a une complexité linéaire en la longueur de la chaîne.

2.2.2 Types produit

Couples Si `'a` et `'b` sont des types, alors `'a * 'b` est un type, dit *produit* de `'a` et de `'b`. C'est le type des couples dont la première composante est de type `'a` et la deuxième de type `'b`.

```
# let a = (1.2, 2.5);;
val a : float * float = (1.2, 2.5)
# let c = (2, [1; 7]);;
val c : int * int list = (2, [1; 7])
# let d = (3.17, c);;
val d : float * (int * int list) = (3.17, (2, [1; 7]))
```

Deux fonctions sont prédéfinies pour accéder aux composantes : `fst : 'a * 'b -> 'a` et `snd : 'a * 'b -> 'b`.

```
# fst d;;
- : float = 3.17
# snd d;;
- : int * int list = (2, [1; 7])
# fst (snd d);;
- : int = 2
```

Le pattern matching marche comme on peut s'y attendre : les fonctions `norme` et `norme2` définies ci-dessous sont parfaitement équivalentes.

```
let norme vecteur =
  match vecteur with
  (x,y) -> sqrt (x**2. +. y**2.)

let norme2 vecteur =
  sqrt ((fst vecteur)**2. +. (snd vecteur)**2.)
```

Cependant, il est possible de le faire directement sur les arguments.

```
let norme3 (x, y) =
  sqrt (x**2. +. y**2.)
```

De manière similaire, on peut utiliser des couples pour définir simultanément deux variables.

```
# let (a, b) = 12, "bonjour";;
val a : int = 12
val b : string = "bonjour"
```

2.2.3 Tuples

Les produits de plus de deux types fonctionnent exactement de la même manière que les couples, à ceci près qu'on ne dispose pas de fonctions prédéfinies pour accéder aux différents éléments : il faut utiliser le pattern matching.

```
# let fst_triplet (x, y, z) = x;;
val fst_triplet : 'a * 'b * 'c -> 'a = <fun>
# let a, b, c = 2.3, 5, [];;
val a : float = 2.3
val b : int = 5
val c : 'a list = []
```

2.3 Types flèches

Si `'a` et `'b` sont des types, `'a -> 'b` est le type des fonctions qui prennent un argument de type `'a` et renvoient un argument de type `'b`. Les types `'a -> ('b -> 'c)` et `('a -> 'b) -> 'c` sont bien sûr différents, et pour éviter d'avoir trop de parenthèses on prend la convention suivante.

$$\text{« 'a -> 'b -> 'c »} \quad := \quad \text{« 'a -> ('b -> 'c) »}$$

Avec les notations mathématiques usuelles, `'a -> 'b` est essentiellement l'ensemble $\mathcal{F}(A,B) = B^A$. On en déduit que

$$\text{« 'a -> ('b -> 'c) »} \quad = \quad \text{« 'a -> ('b -> 'c) »} \quad \approx \quad \mathcal{F}(A,\mathcal{F}(B,C)) \quad = \quad \left(C^B\right)^A$$

En abusant allègrement des règles de calcul sur les puissances, on continue. On a

$$\left(C^B\right)^A \quad \approx \quad C^{A\times B}$$

et donc

$$\text{« 'a -> 'b -> 'c »} \quad \approx \quad \text{« ('a * 'b) -> 'c ».}$$

C'est bien sûr abusif, tant en mathématiques qu'en OCaml, mais on a bien une bijection canonique à défaut d'une égalité. L'application φ définie ci-dessous réalise bien une bijection de $\mathcal{F}(A\times B, C)$ dans $\mathcal{F}(A,\mathcal{F}(B,C))$.

$$\begin{array}{rccl} \varphi : & \mathcal{F}(A\times B,C) & \longrightarrow & \mathcal{F}(A,\mathcal{F}(B,C)) \\ & f & \longmapsto & \varphi(f) : \quad A \longrightarrow \mathcal{F}(B,C) \\ & & & \qquad\quad x \longmapsto [\varphi(f)](x) : \quad B \longrightarrow C \\ & & & \qquad\qquad\qquad\quad y \longmapsto f(x,y) \end{array}$$

C'est le principe de la *curryfication* (en l'honneur du logicien Haskell Curry) qui est la manière standard de définir des fonctions de plusieurs arguments en OCaml. Une fonction `f : 'a -> 'b -> 'c` peut être vue

- soit comme une fonction qui prend un argument de type `'a` et un argument de type `'b`, et renvoie un résultat de type `'c`.
- soit comme une fonction qui prend un argument de type `'a` et renvoie une fonction de type `'b -> 'c`.

Pour définir une fonction en OCaml, on dispose de trois possibilités.

```
let f x_1 ... x_n = expression

let f = (fun x_1 ... x_n -> expression)

let f = function
  | pattern_1 -> expression_1
  | pattern_2 -> expression_2
  ...
  | pattern_n -> expression_n
```

La première forme est celle que nous utiliserons le plus souvent. La seconde est utile pour définir des fonctions *anonymes* (donc sans la partie `let f =`) que l'on utilise immédiatement, souvent comme argument à une fonction d'ordre supérieur.

```
# let compose f g = (fun x -> f (g x));;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let f = compose (fun x -> x*x) (fun x -> x + 1);;
val f : int -> int = <fun>
# f 4;;
- : int = 25
```

La troisième permet de gagner quelques caractères quand on définit une fonction d'un seul argument qui commence immédiatement par un `match` sur l'argument, mais on s'en passe très bien.

```
let rec longueur = function
  | [] -> 0
  | hd :: tl -> 1 + longueur tl
```

Cette définition est parfaitement équivalente à ls définition suivante.

```
let rec longueur l =
  match l with
  | [] -> 0
  | hd :: tl -> 1 + longueur tl
```

Remarquez qu'une fonction récursive doit nécessairement être définie à l'aide de `let rec`.

S'habituer au parenthésage de OCaml peut prendre un peu de temps, mais les règles sont en fait très logiques. Considérons les quelques exemples suivants.

```
(* 1. *) a b
(* 2. *) a (b c)
(* 3. *) a b c
(* 4. *) a (b c) (d e)
(* 5. *) a (b c) (d (e f) g)
```

Dans chaque cas, la première chose à faire est de compter le nombre de « paquets » auxquels on applique la fonction a. Chacun de ces paquets doit être vu comme un argument que l'on passe à a.

1. C'est le cas le plus simple : on applique a à b.

   ```
   # sqrt 2.;;
   - : float = 1.41421356237309515
   ```

   Bien sûr, si `a` attend plus d'un argument, le résultat peut encore être une fonction. Il s'agit alors d'une *application partielle*. Par exemple, puisque `max` est de type `int -> int -> int`.

   ```
   # max 32;;
   - : int -> int = <fun>
   # let g = max 32;;
   val g : int -> int = <fun>
   # g 12;;
   - : int = 32  (* car max 32 12 = 32 *)
   # g 45;;
   - : int = 45  (* car max 32 45 = 45 *)
   ```

2. On ne passe toujours qu'un seul argument à `a`. Cet argument, `b c`, est lui-même le résultat de l'application de `b` à `c`. `a` doit donc être une fonction. On demande tout simplement de calculer $(a \circ b)(c) = a(b(c))$. Un exemple concret simple.

   ```
   # log (exp 12.);;
   - : float = 12.
   ```

3. On passe à `a` deux arguments : `b` et `c`. C'est équivalent à `(a b) c` qui est une application partielle de `a` à `b` puis une application de la fonction ainsi obtenue à `c`.

   ```
   # max 12 34;;
   - : int = 34
   ```

4. On passe encore deux arguments à `a` qui sont `b c` et `d e`. `b` doit nécessairement être une fonction d'au moins un argument, tout comme `d`.

   ```
   (* renverse la liste, puis ajoute 5 à chaque élément *)
   # List.map (( + ) 5) (List.rev [1; 2; 3; 4]);;
   - : int list = [9; 8; 7; 6]
   ```

5. On passe toujours deux arguments à `a` qui sont `(b c)` et `(d (e f) g)`, un argument à `b`, un argument à `e` et deux arguments à `d`.

   ```
   # max (log 2.) (min (exp 1.) 1.5);;
   - : float = 1.5
   (* max(ln(2), min(exp(1), 1.5)) = max(ln(2), 1.5) = 1.5 *)
   ```

2.4 Filtrage

Le *filtrage par motif*, que l'on appelle plus couramment *pattern matching*, fournit une syntaxe extrêmement pratique pour effectuer une disjonction de cas sur la *forme* d'une expression. Cette technique est utilisée constamment lorsqu'on programme en OCaml, en particulier lorsqu'on opère sur des types sommes que l'on verra plus loin.

2.4.1 Forme générale d'un match

```
match expr with
  | filtre_1 -> valeur_1
  | ...
  | filtre_k -> valeur_k
```

- `expr` est une expression quelconque, d'un certain type `'a`. En pratique, ce sera le plus souvent quelque chose de simple :
  - soit une variable : `match x with`.
  - soit un couple ou un triplet de variables : `match x, y with`.
  - soit le résultat d'une application de fonction : `match f x with`.
  - soit parfois une combinaison des cas précédents : `match x, f y, g y with`.
- `filtre_1`,...,`filtre_k` suivent la syntaxe des filtres que l'on va définir ci-dessous, et doivent bien sûr tous correspondre au type `'a` de `expr`.
- `valeur_1`,...,`valeur_k` sont des expressions d'un même type `'b`. L'expression `valeur_i` peut faire intervenir les variables présentes dans `filtre_i`, et bien entendu celles qui étaient définies avant le `match`.
- L'expression complète `match expr with | filtre_1 -> ... | filtre_k -> valeur_k` est de type `'b`, le type commun à `valeur_1`,...,`valeur_k`.

Les filtres sont essayés un par un, dans l'ordre, et le premier qui accepte l'expression `expr` est utilisé : à ce moment, on « passe à droite de la flèche », et la valeur du `match` est obtenue en remplaçant dans `valeur_i` les variables du filtre par ce qui a été capturé.

2.4.2 Forme des filtres

Un filtre peut faire intervenir

- Des *constructeurs de type*. Nous verrons plus tard de quoi il s'agit, mais visuellement c'est très simple : c'est quelque chose qui commence par une majuscule. Il y a en plus le cas particulier du « `::` » pour les listes et de la virgule pour les tuples.
- Des *constantes littérales* comme `:1`, `72`, `3.14`, `'z'`, `true`, `[]`, `"toto"`, `[4; 8; 6]`, etc.
- Des *variables*, qui seront *systématiquement considérées comme fraîches*. Autrement dit, si le filtre contient une variable `x` et qu'il existe déjà une variable portant ce nom, *ces deux variables n'ont rien à voir entre elles.* Attention, un filtre est nécessairement *linéaire*, ce qui signifie qu'une même variable ne peut apparaître qu'une seule fois dans un filtre donné.
- Des « `_` » (*underscore*), qui acceptent tout, comme des variables, mais ne capturent rien. Ils ne peuvent donc être utilisés à droite de la flèche.

2.4.3 Test d'exhaustivité

Le compilateur émet systématiquement un *warning* lorsqu'il n'est pas capable d'assurer qu'un `match` est exhaustif : c'est une fonctionnalité extrêmement puissante, qui permet d'éliminer un nombre incalculable d'erreurs dans les programmes. Il peut arriver que le cas « oublié » soit en fait un cas d'erreur, pour lequel aucune réponse n'est valable. C'est le cas du maximum d'une liste vide, par exemple. Dans ce cas, il est conseillé de rajouter un cas d'erreur explicite, en utilisant `failwith` suivi d'une chaîne de caractères.

```
let rec max_liste lst =
  match lst with
  | [] -> failwith "max non défini"
  | [x] -> x
  | hd :: tl -> max hd (max_liste tl)
```

2.4.4 Clause gardée

On peut rajouter un *garde* à un filtre, avec la syntaxe `filtre when condition -> valeur`. Il faut que `condition` soit un booléen. Si le filtre accepte l'expression de départ, la condition est évaluée. Si cette dernière s'évalue en `true`, alors on « passe à droite de la flèche », sinon on passe au filtre suivant.

Remarque

⇨ On peut toujours éviter l'utilisation de `when` à l'aide de `if...then...else` à droite de la flèche, mais cela donne parfois un code moins clair. Les deux codes suivants sont presque équivalents.

```
match expr with
| filtre when condition -> valeur1
| filtre when not condition -> valeur2
```

```
match expr with
| filtre ->
    if condition then valeur1
    else valeur2
```

La seule différence est que OCaml ne peut pas garantir l'exhaustivité si toutes les clauses correspondant à un motif sont gardées. Ainsi, le code suivant générera un *warning*.

```
let rec nb_positifs lst =
  match lst with
  | [] -> 0
  | hd :: tl when hd >= 0 -> 1 + nb_positifs tl
  | hd :: tl when hd < 0 -> nb_positifs tl
```

On préférera dans ce cas utiliser l'une des deux solutions suivantes.

```
let rec nb_positifs lst =
  match lst with
  | [] -> 0
  | hd :: tl when hd >= 0 -> 1 + nb_positifs tl
  | hd :: tl (* hd < 0 ici *) -> nb_positifs tl

let rec nb_positifs lst =
  match lst with
  | [] -> 0
  | hd :: tl ->
    if hd >= 0 then 1 + nb_positifs tl
    else nb_positifs xs
```

2.4.5 Clause multiple

On peut regrouper plusieurs clauses en une.

```
match expr with
  | filtre_1 | filtre_2 -> ...
  | filtre_3 -> ...
  | ...
```

Attention, dans ce cas, toutes les clauses regroupées doivent contenir exactement les mêmes variables.

2.4.6 match avec un seul cas

Il est possible d'écrire un `match` avec une seule clause, mais dans ce cas on préférera systématiquement le remplacer par un simple `let`. Par exemple, le code suivant

```
match couple with
| a, b -> a + b
```

est parfaitement équivalent à

```
let a, b = couple in a + b
```

On privilégiera la deuxième solution.

2.4.7 Exemples et erreurs à éviter

On commence par quelques exemples de bon usage de « pattern matching ».

Exemples

⇨ La fonction suivante prend en entrée deux listes `[x0; ...; xn]` et `[y0; ...; yn]`, de même type et de même longueur, et renvoie `[max x0 y0; ...; max xn yn]`.

```
(* 'a list -> 'a list -> 'a list *)
let rec max_liste lst1 lst2 =
  match lst1, lst2 with
  | [], [] -> []
  | hd1 :: tl1, hd2 :: tl2 -> (max hd1 hd2) :: max_liste tl1 tl2
  | _ -> failwith "longueurs différentes"
```

⇨ La fonction suivante fait la même chose mais ne suppose plus que les listes sont de même longueur. Lorsqu'une liste est épuisée, on prend simplement les éléments de l'autre.

```
(* 'a list -> 'a list -> 'a list *)
let rec max_liste lst1 lst2=
  match lst1, lst2 with
  | [], [] -> []
  | hd1 :: tl1, [] -> hd1 :: max_liste tl1 []
  | [], hd2 :: tl2 -> hd2 :: max_liste [] tl2
  | hd1 :: tl1, hd2 :: tl2 -> (max hd1 hd2) :: max_liste tl1 tl2
```

⇨ Les fonctions suivantes sont d'autres versions de `max_liste`, ayant la même spécification que dans l'exemple précédent, avec un code un peu plus concis.

```
let rec max_liste lst1 lst2 =
  match lst1, lst2 with
  | [], _ -> lst2
  | _, [] -> lst1
  | hd1 :: tl1, hd2 :: tl2 -> (max hd1 hd2) :: max_liste tl1 tl2

let rec max_liste lst1 lst2 =
  match lst1, lst2 with
  | [], lst | lst, [] -> lst
  | hd1 :: tl1, hd2 :: tl2 -> (max hd1 hd2) :: max_liste tl1 tl2
```

⇨ Il est parfois nécessaire d'imbriquer plusieurs `match`. Dans ce cas, il faut délimiter le `match` interne par un `begin ... end`. Par exemple, la fonction suivante prend en argument une liste de listes `[u0; ...; un]` et renvoie la liste obtenue en prenant à chaque fois le premier élément de u_i si cette liste est non vide et en l'ignorant si elle est vide.

```
(* a' list list -> 'a list *)
let rec tetes lst =
  match lst with
  | [] -> []
  | hd :: tl ->
    begin match hd with
      | [] -> tetes tl
      | x :: _ -> x :: tetes tl
    end
```

Ces `match` imbriqués sont un peu lourds, et on peut *très souvent les éviter*. Ici, on écrirait plutôt

```
let rec tetes lst =
  match lst with
  | [] -> []
  | [] :: tl -> tetes tl
  | (x :: _) :: tl -> x :: tetes tl
```

Voici maintenant quelques erreurs classiques à éviter, et les manières de les corriger.

Exemples

⇨ *Les variables sont fraîches.* Considérons la fonction suivante, dont le but est de déterminer si l'élément `x` apparaît dans la liste `lst`.

```
(* Code faux         *)
let rec appartient x lst =
  match lst with
  | [] -> false
  | x :: tl -> true
  | y :: tl -> appartient x tl
(* Fin du code faux *)
```

À la compilation, on obtient un *warning* nous indiquant que le dernier cas ne sera jamais utilisé. En effet, la variable `x` apparaissant dans le deuxième cas n'a *rien à voir avec l'argument* `x` *de la fonction*. Le deuxième cas se lit essentiellement : « si la liste est de la forme `x :: tl` pour un certain `x` et un certain `tl`, alors `true` », or toute liste non vide est de cette forme. Dans un cas comme celui-ci, on utilisera soit une clause gardée (avec un `when`), soit un `if ... then ... else` à droite de la flèche.

```
let rec appartient x lstlet rec appartient x lstlet rec appartient x lst
let rec appartient x lst   | y :: tl when y = x -> true
let rec appartient x lst   | _ :: tl -> appartient x tl
```

```
let rec appartient x lst =
  match u with
  | [] -> false
  | y :: tl -> if y = x then true else appartient x tl
```

Dans la deuxième version, le `if ... then ... else` est en fait maladroit et peut être simplifié. On obtient alors

```
let rec appartient x lst =
  match lst with
  | [] -> false
  | y :: tl -> y = x || appartient x tl
```

⇨ *Un entier n'a pas de forme.* En mathématiques, il est tout à fait raisonnable d'écrire : « si n est de la forme $2p$ avec p entier, alors… ». En revanche, informatiquement, un entier n'a essentiellement pas de forme : c'est juste un entier. Par exemple, si l'on souhaite compter le nombre d'entiers pairs dans une liste d'entiers

```
(* Code faux !!!     *)
let rec nombre_pairs lst =
  match lst with
  | [] -> 0
  | (2 * p) :: tl -> 1 + nombre_pairs tl
  | _ :: tl -> nombre_pairs tl
(* Fin du code faux *)

(* Code correct *)
let rec nombre_pairs lst =
  match lst with
  | [] -> 0
  | n :: tl when n mod 2 = 0 -> 1 + nombre_pairs tl
  | _ :: tl -> nombre_pairs tl

(* ou bien *)
let rec nombre_pairs lst =
  match lst with
  | [] -> 0
  | n :: tl ->
    if n mod 2 = 0 then 1 + nombre_pairs tl
    else nombre_pairs tl

(* ou encore *)
let rec nombre_pairs lst =
  match lst with
  | [] -> 0
  | n :: tl -> nombre_pairs tl + if n mod 2 = 0 then 1 else 0
```

Comme un entier n'a pas de structure, il est assez rarement intéressant de faire un `match` sur un entier. Par exemple, la fonction suivante renvoie $-1, 0$ ou 1 suivant le signe de son argument, et elle est correcte, mais on préférera un `if ... then ... else`.

```
(* Code correct mais maladroit *)
let signe n =
  match n with
  | n when n > 0 -> 1
  | n when n = 0 -> 0
  | _ -> -1

(* code à privilégier *)
let signe n =
  if n > 0 then 1
  else if n = 0 then 0
  else -1
```

2.5 Définition de types

2.5.1 Syntaxe

Pour nommer un nouveau type en OCaml, on utilise le mot-clé `type`.

```
type p3D = float * float * float
type v3D = float * float * float
type sphere = p3D * float
```

Pour des types comme ceux-ci, qui « existent déjà », l'intérêt peut sembler limité. Cependant la lisibilité du code peut se trouver grandement améliorée.

```
let vect3D ((x1, y1, z1) : p3D) ((x2, y2, z2) : p3D) : v3D =
  ((x2 -. x1), (y2 -. y1), (z2 -. z1))

let norme3D ((x, y, z) : v3D) = sqrt (x**2. +. y**2. +. z**2.)

let dist3D (a : p3D) (b : p3D) = norme3D (vect3D a b)

let in_sphere (a : p3D) ((centre, r) : sphere) =
  (dist3D a centre) <= r
```

Remarquez les *annotations de type* : on peut quand on déclare une fonction utiliser la syntaxe

```
let f (x_1 : t_1) ... (x_n : t_n) : u = ...
```

pour spécifier que les arguments `x_1`, ..., `x_n` doivent respectivement être de type `t_1`, ..., `t_n` et que le résultat doit être de type `u`. Ici, le type inféré pour `in_sphere` est

```
# in_sphere;;
- : p3D -> sphere -> bool = <fun>
```

à comparer avec le résultat suivant obtenu si l'on n'annote rien.

```
# in_sphere_2;;
- : float * float * float -> (float * float * float) * float -> bool = <fun>
```

2.5.2 Types sommes

Les types produits correspondent à un produit cartésien $A \times B$ et les types flèches à un ensemble d'applications $A \to B$ (ou B^A). La troisième opération algébrique fondamentale sur les types est l'union disjointe : les types correspondant à cette opération sont appelés *types sommes*, ou *types union*.

Types énumérés finis Un type ne contenant qu'un nombre fini de valeurs peut être défini avec la syntaxe suivante.

```
type enumeration = Constructeur_1 | Constructeur_2 | ... | Constructeur_n
```

On dit que c'est un *type énuméré*. Les *constructeurs de type* doivent obligatoirement commencer par une majuscule, et ce sont les seuls noms, avec les modules, qui peuvent commencer par une majuscule en OCaml.

On peut par exemple définir un type `signe` à trois valeurs.

```
type signe = Negatif | Nul | Positif
```

On traduit ensuite la règle sur le signe d'un produit.

```
let signe_produit s1 s2 =
  match s1, s2 with
  | Nul, _ -> Nul
  | _, Nul -> Nul
  | Positif, _ -> s2
  | Negatif, Positif -> Negatif
  | Negatif, Negatif -> Positif
```

On peut ensuite utiliser cette fonction pour déterminer le signe d'un produit d'entiers.

```
let signe_of_int n =
  if n = 0 then Nul
  else if n < 0 then Negatif
  else Positif

let signe_produit_int n p =
  signe_produit (signe_of_int n) (signe_of_int p)
```

```
# signe_produit Negatif Negatif;;
- : signe = Positif
# signe_produit_int 10_000_000_000 10_000_000_000;;
- : signe = Positif
# signe_of_int (10_000_000_000 * 10_000_000_000);;
- : signe = Negatif (* Pourquoi ? *)
```

On peut remarquer que l'inférence de type et le pattern matching fonctionnent sans problème sur un type défini de cette manière.

Parmi les types de base de OCaml, l'un est essentiellement un type énuméré et peut être réimplémenté sans aucune difficulté : le type `bool`.

```
type booleen = Vrai | Faux

let et a b =
  match a, b with
  | Vrai, Vrai -> Vrai
  | _ -> Faux
```

Attention, la définition de la fonction `et` n'est en fait pas équivalente à celle du `&&` de OCaml puisqu'elle évalue systématiquement ses deux arguments, alors qu'on a vu que l'opérateur `&&` est séquentiel. C'est la même chose pour `ou`.

Constructeurs paramétriques Si l'on veut définir un type contenant une infinité de valeurs, il faut pouvoir paramétrer les constructeurs par des valeurs d'un type préexistant. Par exemple, définissons un type permettant de représenter les sous-espaces vectoriels de $\mathbb{R}^2$, et une fonction permettant de calculer l'intersection de deux tels sous-espaces. On va oublier un instant que tester l'égalité de deux flottants est dangereux en raison des erreurs d'arrondi.

```
(* Un sev de R^2 est soit réduit à zéro,
   soit une droite définie par un vecteur directeur (un couple de flottants),
   soit le plan tout entier. *)
type sev =
  | Zero
  | Droite of float * float
  | Plan

let intersection e f  =
  match e, f with
```

```
  | Plan, _ -> f
  | _, Plan -> e
  | Droite (x1, y1), Droite (x2, y2) when x1 *. y2 -. x2 *. y1 = 0. -> e
    (* On vient de faire un test d'égalité sur les flottants : c'est trés mal *)
  | _ -> Zero
```

2.5.3 Types paramétriques

Certains types de OCaml sont dits *paramétriques* : par exemple, une `'a list` contient des éléments qui ont tous le même type, mais ce type est quelconque.

Nous aurons souvent à définir de tels types, typiquement lorsque nous définirons de nouvelles structures de données susceptibles de contenir des données de type quelconque. Pour l'instant, nous nous contenterons de donner un exemple pour illustrer la syntaxe.

```
(* tirage avec ordre (Liste) ou sans ordre (Combinaison) *)
type 'a tirage =
  | Liste of 'a list
  | Combinaison of 'a list

(* Pour ignorer l'ordre quand on détermine si deux combinaisons sont égales,
 * le plus simple est de les trier avant de les comparer *)
let egal_tirages x y =
  match x, y with
  | Liste u, Liste v -> u = v
  | Combinaison u, Combinaison v -> (List.sort compare u = List.sort compare v)
  | _ -> false
```

```
# egal_tirages (Liste ["a"; "b"; "x"]) (Liste ["x"; "b"; "a"]);;
- : bool = false
# egal_tirages (Combinaison ["a"; "b"; "x"]) (Combinaison ["x"; "b"; "a"]);;
- : bool = true
```

2.5.4 Type 'a option

Le type `option` est prédéfini en OCaml par

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

C'est un type très utile pour définir des fonctions qui peuvent ne pas avoir de résultat à renvoyer. Attention, une valeur de type `'a option` n'est pas de type `'a` : il faut « déconstruire » l'option si l'on veut récupérer le `'a`. Par exemple, les deux fonctions suivantes renvoient :

— `Some i` si `i` est le premier indice d'apparition de l'élément `x` dans la liste `lst`.
— `None` si `x` n'apparaît pas dans `lst`.

```
  let rec find x lst =
    match lst with
    | [] -> None
    | hd :: _ when hd = x -> Some 0
    | _ :: tl ->
      begin
        match find x tl with
        | None -> None
        | Some n -> Some (n + 1)
      end
```

```
let find x lst =
  let rec aux x lst n =
    match lst with
    | [] -> None
    | hd :: _ when hd = x -> Some n
    | hd :: tl -> aux x tl (n + 1) in
  aux x lst 0
```

Quand on utilise ces fonctions, il faut « déballer » l'option.

```
(* minimum de (index x u) et (index x v) (ou None) *)
let premier_x x u v =
  match (index x u, index x v) with
  | Some n, Some p -> Some (min n p)
  | Some n, None -> Some n
  | None, Some p -> Some p
  | None, None -> None
```

```
# premier_x 7 [1; 3; 5; 7] [2; 3; 1; 4; 8; 7];;
- : int option = Some 3
# premier_x 8 [1; 3; 5; 7] [2; 3; 1; 4; 8; 7];;
- : int option = Some 4
# premier_x (-12) [1; 3; 5; 7] [2; 3; 1; 4; 8; 7];;
- : int option = None
```

2.5.5 Types enregistrement

Un type enregistrement (*record* en anglais) est essentiellement un type produit dans lequel chaque composante a un *nom*. La syntaxe est

```
type enregistrement = {nom_1 : type_1 ; ... ; nom_n : type_n};;
```

et, si `x` est de type `enregistrement`, on accède ensuite à la valeur du champ `nom_1` de `x` par `x.nom_1`.

Remarques

- ⇨ Les règles qui déterminent ce qui se passe quand plusieurs types enregistrement différents ont des champs qui portent le même nom sont un peu complexes : le plus simple est d'éviter cette situation.
- ⇨ L'ordre des champs n'a pas d'importance, seul le nom compte. Par exemple, dans l'exemple qui suit, sur les nombres complexes, `{re = 1.0 ; im = 0.0}` et `{im = 0.0 ; re = 1.0}` sont indiscernables.

Exemple élémentaire : nombres complexes Si l'on veut définir un type pour représenter des complexes, il est assez logique de procéder ainsi

```
type complexe = {re : float ; im : float}
let conjugue z = {re = z.re ; im = -.z.im}
```

```
# let i = {re = 0.0; im = 1.0};;
val i : complexe = {re = 0.; im = 1.}
# let j = {im = (sqrt 3.) /. 2.; re = -0.5};;
val j : complexe = {re = -0.5; im = 0.8660254037844386597}
(* L'ordre dans lequel on remplit les champs n'a pas d'importance :
   OCaml regarde les noms *)
# conjugue i;;
- : complexe = {re = 0.; im = -1.}
# conjugue j;;
- : complexe = {re = -0.5; im = -0.8660254037844386597}
```

On peut aussi faire du pattern matching sur des enregistrements même si ce n'est qu'assez rarement utile.

```
let norme z =
  match z with
  {re = x ; im = y} -> sqrt (x**2. +. y**2.)
```

```
# norme j = 1.;;
- : bool = false
# norme j;;
- : float = 0.99999999999999889
(* Les tests d'égalité sur les flottants, ca ne marche pas ! *)
```

Dans ce cas, tout comme pour les types produits, il est plus pratique de « déconstruire » l'objet directement.

```
let norme {re = x; im = y} =
  sqrt (x**2. +. y**2.)
```

En réalité, le mieux est souvent de ne pas déconstruire, mais d'accéder aux champs quand on en a besoin.

```
let norme z =
  sqrt (z.re ** 2. +. z.im ** 2.)
```

Un exemple classique : le jeu de cartes On veut définir un type permettant de représenter des cartes à jouer.
- Une carte est caractérisée par sa *couleur* et sa *valeur*.
- Les couleurs possibles sont pique, cœur, carreau et trèfle.
- Les valeurs possibles sont as, roi, dame, valet ainsi que les entiers de 2 à 10.

Une possibilité pour le type

```
type couleur = Pique | Coeur | Carreau | Trefle
type valeur = As | Roi | Dame | Valet | Mineure of int
type carte = {co : couleur ; va : valeur}
(* Le 8 de trefle sera donc {co = Trefle ; va = Mineure 8}
   et le roi de coeur {co = Coeur ; va = Roi} *)
```

Et deux exemples de fonctions très simples opérant sur ce type.

```
let pique_ou_as carte =
  match carte with
  | {co = Pique} -> true (* on n'est pas obligé de "matcher" tous les champs *)
  | {va = As} -> true
  | _ -> false

(* Ici, le pattern matching n'apporte pas grand-chose *)
let trefle_ou_8 carte =
  (carte.co = Trefle) || (carte.va = Mineure 8)
```

3 OCaml impératif

3.1 Type `unit`

`unit` est l'un des types de base de OCaml : celui des *actions*. Il n'y a qu'une seule valeur de type `unit`, c'est `()`.

```
# let x = ();;
val x : unit = ()
```

Une fonction de type `t -> unit`, où `t` est un type, ne renvoie rien (techniquement, une telle fonction renvoie `()`, un peu comme une fonction sans `return` en Python qui renvoie `None`). La fonction `print_string`, par exemple, est de type `string -> unit` : si on l'applique à une chaîne en écrivant `print_string "Hello!"`, on n'obtient pas une « valeur » qui pourrait être réutilisée dans un calcul ultérieur, mais une *action* : l'affichage de `Hello!` à l'écran. En C (et dans de nombreux langages inspirés du C), une telle fonction aurait `void` comme type de retour.

Une fonction peut aussi être de type `unit -> 'a` pour un certain type `'a`. Cela signifie que la fonction ne prend pas de paramètre (significatif) en entrée. Par exemple, la fonction `print_newline : unit -> unit` qui affiche une ligne vide et qui sert donc surtout à revenir à la ligne entre deux affichages.

```
# print_newline;;
- : unit -> unit = <fun>
# print_newline ();;
- : unit = ()
(* On remarquera que même si print_newline ne prend "pas d'argument", il faut
   quand même lui passer l'argument () pour effectuer l'appel. *)
```

Il arrive souvent que l'on souhaite exprimer quelque chose comme « dans ce cas, fais cela, dans les autres cas, ne fais rien ». C'est tout à fait possible, en utilisant `()` pour dire « ne fais rien ».

```
let affiche_si_pair n =
  match n mod 2 with
    | 0 -> print_int n
    | _ -> ()

(* ou bien *)
let affiche_si_pair n =
```

```
  if n mod 2 = 0 then
    print_int n
  else
    ()
```

Ce dernier cas se présente assez souvent : c'est pour cela que dans la plupart des langages, la branche `else` est optionnelle dans un `if ... then ... else`. En OCaml, elle est en général obligatoire, sauf si le type de retour est `unit`.

```
(* Code faux *)
let abs n = if n < 0 then -n
(* erreur de type : c'est logique, qu'est censé valoir abs 3 ? *)

(* Code correct *)
let affiche_si_negatif n = if n < 0 then print_int n
(* strictement équivalent a :
   let affiche si_pair n = if n < 0 then print_int n else () *)
```

L'idée est simplement que `if cond then exp1 else exp2` est une expression dotée d'une valeur et donc d'un type. Ce type est celui, nécessairement commun, de `exp1` et `exp2`. Dans le cas où `exp1` est de type `unit`, on dispose d'une valeur par défaut `()` si `exp2` n'est pas spécifié . Si `exp1` a un autre type, ce n'est pas le cas.

3.2 Variables mutables

En mathématiques, si l'on écrit « *soit x l'unique réel vérifiant...* » ou « *soit f la fonction qui à un réel x associe...* », il est sous-entendu que l'objet ainsi défini ne changera jamais de valeur. On peut éventuellement réutiliser le *nom* f pour désigner ultérieurement une autre fonction, une fois qu'on n'aura plus besoin de la première fonction f, mais il s'agira alors d'un autre objet : les occurrences de f qui suivent la première définition, mais précèdent la deuxième, feront toujours référence à la première définition. Il en est de même en OCaml, comme le montre l'exemple suivant.

```
# let x = 3;;
val x : int = 3
# let y = x + 2;;
val y : int = 5
# let f n = x * n;;
val f : int -> int = <fun>
# let x = 7;;
val x : int = 7
# y;;
- : int = 5
(* Pas trés surprenant, le résultat serait le même dans la plupart des langages :
   quand "let y = x + 2" a été executé, x a été remplacé par sa valeur.
   La valeur de y n'est pas affectée par une redéfinition ultérieure de x. *)
# f 3;;
- : int = 9
(* Plus surprenant : le "x" dans la définition de f se référe
   toujours à la définition de x en vigueur au moment du "let f =" *)
```

Il est cependant possible de définir des variables pouvant changer de valeur : il faut juste le spécifier.

3.2.1 Champs mutables dans un enregistrement

Quand on définit un type enregistrement, il est possible de spécifier que certains champs sont modifiables après la création à l'aide du mot-clé `mutable`. Définissons par exemple un type `compteur` qui contiendra trois champs entiers : `debut` et `fin` contiendront respectivement les valeurs initiales et finales du compteur, alors que `courant` contiendra la valeur courante. Seul ce dernier champ est susceptible de changer durant la « vie » du compteur, ce qui nous conduit à la définition de type suivante.

```
type compteur = {debut : int ; fin : int ; mutable courant : int}
```

Une fois créé un `compteur`, on peut modifier la valeur de son champ `courant` avec l'opérateur `<-`.

```
# let c = {debut = 1; fin = 5; courant = 1};;
val c : compteur = {debut = 1; fin = 5; courant = 1}
# c.courant <- 4;;
- : unit = ()
```

```
# c;;
- : compteur = {debut = 1; fin = 5; courant = 4}
# c.debut <- 2;;
Characters 0-12:
  c.debut <- 2;;
  ^^^^^^^^^^^^
Error: The record field label debut is not mutable
```

On peut maintenant définir une fonction `incremente : compteur -> unit` qui tente d'incrémenter le compteur et lève une exception si le compteur est déjà à sa valeur finale.

```
let incremente c =
  if c.courant < c.fin then
    c.courant <- c.courant + 1
  else
    failwith "valeur maximale atteinte"
```

Une telle fonction est dite *impure*, ou *avec effets de bord*. La différence fondamentale avec les fonctions *pures* vues jusqu'à maintenant est que `incremente` modifie l'« univers » quand on l'appelle. En particulier, cela signifie que deux appels successifs à `incremente` avec le même argument peuvent avoir un comportement différent.

```
# let test = {debut = 1 ; fin = 5 ; courant = 3};;
val test : compteur = {debut = 1; fin = 5; courant = 3}
# incremente test;;
- : unit = ()
# test;;
- : compteur = {debut = 1; fin = 5; courant = 4}
# incremente test;;
- : unit = ()
# test;;
- : compteur = {debut = 1; fin = 5; courant = 5}
# incremente test;;
Exception: Failure "valeur maximale atteinte".
```

3.2.2 Une syntaxe allégée : les références

Très souvent, on souhaite simplement créer une variable « mutable » d'un certain type `t` préexistant. Il est bien sûr possible définir un type enregistrement *ad hoc*.

```
type 'a modifiable = {mutable valeur : 'a}
```

On peut alors procéder comme suit.

```
# let x = {valeur = 3};;
val x : int modifiable = {valeur = 3}
# x.valeur <- 17;;
- : unit = ()
# x.valeur <- x.valeur * 2;;
- : unit = ()
# x;;
- : int modifiable = {valeur = 34}
```

Cependant, cette manière de procéder est quelque peu lourde d'un point de vue syntaxique. OCaml propose donc un *syntactic sugar* : le type prédéfini `'a ref`. Il s'agit du même type (aux noms près) que le `'a modifiable` défini plus haut.

```
type 'a ref = {mutable contents : 'a};;
```

L'avantage est que l'on dispose de deux opérateurs prédéfinis `!` (préfixe) et `:=` (infixe).

Opérateur	Type	Signification	Syntaxe
`ref`	`'a -> 'a ref`	Création d'une référence	`ref x` équivaut à `{contents = x}`
`!`	`'a ref -> 'a`	Dé-référencement	`!x` équivaut à `x.contents`
`:=`	`'a ref -> 'a -> unit`	Affectation	`x := y` équivaut à `x.contents <- y`

Un exemple d'utilisation.

```
# let x = ref 3.;;
val x : float ref = {contents = 3.}
# x +. 1.2;;
Characters 0-1:
  x +. 1.2;;
  ^
Error: This expression has type float ref
       but an expression was expected of type float
# !x +. 1.2;;
- : float = 4.2
# x := (!x)**3.;;
- : unit = ()
# x;;
- : float ref = {contents = 27.}
```

Si `x` est une référence et que l'on écrit `let y = x`, on fait « pointer » `y` vers la même case mémoire que `x` (on parle d'égalité *physique*). Cela signifie que toute modification ultérieure du contenu de `x` affectera `y`, et inversement.

```
# let x = ref 0;;
val x : int ref = {contents = 0}
# let y = x;;
val y : int ref = {contents = 0}
# x := 2;;
- : unit = ()
# y;;
- : int ref = {contents = 2}
# y := -5;;
- : unit = ()
# x;;
- : int ref = {contents = -5}
# let x = ref 2;;
val x : int ref = {contents = 2}
(* On définit un nouveau x qui ne pointe plus vers la même case *)
# y;;
- : int ref = {contents = -5}
(* Le nouveau x et y ne sont donc plus liés. *)
```

3.3 L'opérateur ;

En OCaml, le `;` est un opérateur de *séquencement* : on peut considérer que `instr_1 ; instr_2` signifie « exécuter l'instruction 1, puis l'instruction 2 ». Plus précisément, `expression_1 ; expression_2` signifie :

- Calculer `expression_1`. Ce calcul peut comporter des effets de bord et donc résulter en un affichage, la modification d'une variable mutable, etc.
- Jeter le résultat de ce calcul s'il y en a un.
- Calculer `expression_2`. Ici aussi, il peut y avoir des effets de bord) ; le résultat de cette évaluation donne la valeur de l'expression complète `expression_1; expression_2`.

Techniquement, `;` est un opérateur infixe de type `'a -> 'b -> 'b`. Cependant, le type `'a` sera presque toujours `unit` (OCaml émet d'ailleurs un *warning* si ce n'est pas le cas). Un exemple typique d'utilisation est le suivant :

```
let incremente c =
  if c.courant < c.fin then
    begin
      c.courant <- c.courant + 1;
      print_string "La nouvelle valeur est ";
      print_int c.courant
    end
  else
    failwith "valeur maximale atteinte"
```

```
# c;;
- : compteur = {debut = 1; fin = 5; courant = 3}
```

```
# incremente c;;
La nouvelle valeur est 4- : unit = ()
# c;;
- : compteur = {debut = 1; fin = 5; courant = 4}
```

Remarquez que le `begin ... end` est *indispensable* (on peut cependant remplacer le `begin` par une parenthèse ouvrante et le `end` par une parenthèse fermante) : le code `if a then b; c else d` est lu `(if a then b); c else d`, ce qui résulte en une erreur de syntaxe. La question de savoir ce qui fait ou non partie du corps d'un `then` ou d'un `else` est une *source d'erreurs récurrentes*. Si c'est possible, un bon réflexe est de demander régulièrement à votre éditeur de texte de réindenter le code, ce qui permet de mettre en évidence les différences éventuelles entre ce que l'on voulait dire et ce que le compilateur va comprendre.

```
(* Code mal indenté : ne fait pas ce qu'on pourrait croire *)
if condition then
  instruction_1
else
  instruction_2;
  instruction_3

(* Code correctement indenté : on voit que instruction_3 ne fait pas partie
   du if/then/else et sera exécutée dans tous les cas. *)
if condition then
  instruction_1
else
  instruction_2;
instruction_3
```

3.4 Boucles

3.4.1 Boucle `for`

OCaml propose deux variantes de la boucle `for`, selon que l'indice croît ou décroît.

```
for compteur = start to finish do
  expression
done
```

```
for compteur = start downto finish do
  expression
done
```

`compteur` est considéré comme une variable de type `int` locale à la boucle et ne peut pas être modifié « à la main » dans le corps de la boucle. `start` et `finish` doivent être de type `int` et `expression` de type `unit`. Notez bien que les deux bornes de la boucle sont **incluses**.

Calculons par exemple la somme des entiers et des carrés de `deb` à `fin` (inclus).

```
let sommes deb fin =
  let s = ref 0 in
  let t = ref 0 in
  for i = deb to fin do
    s := !s + i;
    t := !t + i*i
  done;
  (!s, !t)

(* val sommes : int -> int -> int * int = <fun>
   # sommes 1 5;;
   - : int * int = (15, 55)                    *)
```

OCaml ne possède pas d'équivalents de `break`, `continue` et autres : une boucle `for` est forcément exécutée en entier, et l'on ne sort d'une boucle `while` que quand la condition devient fausse.

3.4.2 Boucle `while`

La syntaxe d'une boucle `while` en OCaml est

```
while condition do
  expression
done;
```

où `condition` est de type `bool` et `expression` de type `unit`. Par exemple, une fonction `inverse_fact : int -> int` telle que `inverse_fact n` renvoie le premier entier k tel que $k! \geqslant n$.

```
let inverse_fact n =
  let k = ref 0 in
  let f = ref 1 in
  while !f < n do
    k := !k + 1;
    f := !f * !k
  done;
  !k
```

3.4.3 Itération sur les éléments d'une structure de données

En Python, il est souvent bien plus pratique d'itérer directement sur les éléments d'une liste plutôt que de passer par les indices. On peut même en profiter pour déconstruire les éléments le cas échéant.

```
def affiche(liste):
  s = 0
  for (x, y) in liste:
    print("Somme : ", x + y)
    s = s + x + y
  print("Total : ", s)
```

Cette fonction prend une liste de couples de nombres et affiche la somme de chacun des couples puis la somme totale. La manière typique d'écrire cela en OCaml serait

```
let affiche liste =
  let s = ref 0 in
  let affiche_couple (x, y) =
    print_endline ("Somme : " ^ string_of_int (x + y));
    s := !s + x + y in
  List.iter affiche_couple liste;
  print_endline ("Total : " ^ string_of_int !s)
```

La boucle `for (x, y) in liste:` a été remplacée par un appel à `List.iter`. La spécification de cette fonction est

- `List.iter : ('a -> unit) -> 'a list -> unit`
- `List.iter f [a1; ...; an]` équivaut à `begin f a1; ...; f an; () end`. Autrement dit, `List.iter` appelle la fonction f successivement sur chacun des éléments de la liste, ce qui revient exactement à une boucle `for` dont le corps serait constitué de f.

Notez que le code est plus lourd en OCaml, mais principalement à cause de la méthode choisie pour l'affichage. En réalité, on écrirait plutôt

```
open Printf
(* Pour ecrire f plutot que Printf.f pour les fonctions du module Printf *)

let affiche_bis liste =
  let s = ref 0 in
  let f (x, y) = printf "Somme : %d\n" (x + y); s := !s + x + y in
  List.iter f liste;
  printf "Total : %d\n" !s
```

3.5 Tableaux

3.5.1 Type `'a array`

Les tableaux sont la principale alternative aux listes pour stocker un nombre quelconque d'éléments d'un même type. Pour un tableau `t` de n éléments, les indices varient de 0 à $n-1$ et l'on peut accéder directement à l'élément

d'indice `i` par `t.(i)`. De plus, un tableau est une structure mutable dont on peut modifier l'élément d'indice `i` par `t.(i) <- nouvelle_valeur`.

```
# let t = [| 2.3 ; 3.4 ; -2.1 |];;
val t : float array = [|2.3; 3.4; -2.1|]
# t.(0);;
- : float = 2.3
# t.(1) <- 5.72;;
- : unit = ()
# t;;
- : float array = [|2.3; 5.72; -2.1|]
```

Les fonctions de manipulation des tableaux se trouvent dans le module `Array` de la bibliothèque standard, on y accède par `Array.nom_de_la_fonction`. Certaines de ces fonctions sont données ici, vous êtes invités à consulter le manuel pour découvrir les autres.

Fonction	Type	Utilisation
`Array.make`	`int -> 'a -> 'a array`	`Array.make n x = [|x;..;x|]` (taille n)
`Array.length`	`'a array -> int`	Donne le nombre d'éléments d'un tableau.
`Array.of_list`	`'a list -> 'a array`	Convertit une liste en un tableau.
`Array.to_list`	`'a array -> 'a list`	Convertit un tableau en une liste.
`Array.init`	`int -> (int -> 'a) -> 'a array`	`Array.init n f = [|f 0;..;f (n-1)|]`
`Array.map`	Voir plus bas	Comme pour les listes
`Array.iter`	Voir plus bas	Comme pour les listes
`Array.fold_left`	Voir plus bas	Comme pour les listes
`Array.fold_right`	Voir plus bas	Comme pour les listes

Remarques

⇨ La fonction `Array.length` s'exécute en temps constant, contrairement à la fonction `List.length`.

⇨ Les fonctions `Array.map`, `Array.iter`, `Array.fold_left` et `Array.fold_right` sont des équivalents exacts des fonctions correspondantes du module `List`.
- `Array.map : ('a -> 'b) -> 'a array -> 'b array`, telle que `Array.map f t` renvoie `[| f t.(0); ... ; f t.(n - 1) |]` ;
- `Array.iter : ('a -> unit) -> 'a array -> unit`, telle que `Array.iter f t` équivaut à `begin f t.(0); ... ; f t.(n - 1) end` ;
- `Array.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a`, telle que `Array.fold_left f init t` renvoie `f (... (f (f init t.(0)) t.(1)) ...) t.(n-1)` ;
- `Array.fold_right : ('b -> 'a -> 'a) -> 'b array -> 'a -> 'a`, telle que `Array.fold_right f t init` renvoie `f t.(0) (f t.(1) ( ... (f t.(n-1) init) ...))`.

3.5.2 Aliasing

Attention, quand `t` est un tableau, `let u = t` fait pointer `u` vers la même zone mémoire que `t`. Ainsi, toute modification de l'un entraîne une modification de l'autre.

```
# let t = [|3; 7|];;
val t : int array = [|3; 7|]
# let u = t;;
val u : int array = [|3; 7|]
(* u et t sont *physiquement* égaux *)
# u.(0) <- 5;;
- : unit = ()
(* On modifie u. *)
# t;;
- : int array = [|5; 7|]
(* t a également été modifié. *)
```

Le problème se pose souvent quand on souhaite créer un tableau de tableaux.

```
# let t = [|0 ; 0 |];;
val t : int array = [|0; 0|]
```

```
# let u = Array.create 4 t;;
val u : int array array = [|[|0; 0|]; [|0; 0|]; [|0; 0|]; [|0; 0|]|]
(* u est maintenant constitué de 4 pointeurs vers le même tableau t. *)
# t.(1) <- 5;;
- : unit = ()
(* On modifie t. *)
# u;;
- : int array array = [|[|0; 5|]; [|0; 5|]; [|0; 5|]; [|0; 5|]|]
(* Tous les éléments de u ont été modifiés *)
# u.(2).(0) <- -1;;
- : unit = ()
(* On modifie l'un des éléments de u en place. *)
# u;;
- : int array array = [|[|-1; 5|]; [|-1; 5|]; [|-1; 5|]; [|-1; 5|]|]
(* Les autres ont aussi été modifiés. *)
```

Pour initialiser correctement une matrice, il que chaque ligne pointe vers un tableau différent :

```
(* Renvoie une matrice n * p dont tous les elements valent init *)
let make_matrix n p init =
  let t = Array.create n [| |] in
  for k = 0 to n - 1 do
    t.(k) <- Array.create p init
  done;
  t
```

Cette fonction existe déjà dans la bibliothèque standard : c'est

`Array.make_matrix : int -> int -> 'a -> 'a array array`.

3.5.3 Parcours de tableaux

Même si ce n'est pas une règle absolue, on privilégie souvent les boucles pour parcourir les tableaux de la même manière que l'on privilégie la récursion pour parcourir les listes Attention, beaucoup d'algorithmes sur les tableaux sont essentiellement récursifs. Pour un simple parcours, en revanche, une boucle est plus naturelle. Deux exemples :

- La fonction `somme : int array -> int` calcule tout simplement la somme des éléments d'un `int array`.
- La fonction `mini : 'a array -> 'a * int list` renvoie un couple `x, l` où `x` est le minimum du tableau fourni en entrée et `l` la liste des occurrences de `x` dans `t`.

```
let somme t =
  let s = ref 0 in
  let n = Array.length t in
  for i = 0 to n - 1 do
    s := !s + t.(i)
  done;
  !s
```

```
let mini t =
  let n = Array.length t in
  let x = ref t.(0) in
  let l = ref [0] in
  for i = 1 to n - 1 do
    if t.(i) < !x then begin
       x := t.(i);
       l := [i]
    end
    else if t.(i) = x then l := i :: !l
  done;
  (!x, !l)
```

```
# mini [|0; 3; 0; -2; 3; 1; -2|];;
- : int * int list = (-2, [6; 3])
# mini [|2; 4; 3; 6; 1; 4; 1; 3; 1; 5|];;
- : int * int list = (1, [8; 6; 4])
```

Remarque

⇨ Les cases d'un tableau sont numérotées de 0 à $n-1$ comme en Python, mais dans `for i = debut to fin`, les deux bornes sont incluses (contrairement à un `range` Python) : on écrira donc très souvent des boucles `for i = 0 to n - 1`.

3.6 Caractères et chaînes de caractères en OCaml

Le type `char`

— Le type `char` est codé sur 8 bits.
— Les littéraux de type `char` sont de la forme `'a'` pour les caractères ASCII imprimables :

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

— Ce n'est pas un type entier (on ne peut pas écrire `'b' + 3`), mais on dispose de deux fonctions de conversion.
  — `int_of_char : char -> int` qui associe à chaque caractère un entier entre 0 et 255.
  — `char_of_int : int -> char` qui prend un entier *entre 0 et 255* et renvoie le caractère correspondant.

```
# char_of_int 117;;
- : char = 'u'
# char_of_int 180;;
- : char = '\180'
# char_of_int 500;;
Exception: Invalid_argument "char_of_int".
# int_of_char 'x';;
- : int = 120
# int_of_char ' ';;
- : int = 32
# char_of_int (int_of_char 'a' + 4);;
- : char = 'e'
```

Le type `string`

Une chaîne de caractères est stockée de manière contiguë en mémoire, et elle connaît sa taille. Elle se comporte essentiellement comme un tableau de caractères, sauf que
— La syntaxe pour accéder à un élément est légèrement différente : `s.[i]` (des crochets au lieu de parenthèses) ;
— Elle n'est pas mutable (on ne peut pas faire `s.[i] <- ...`).

Les opérations les plus courantes sur les `string` sont
— *Accès à un caractère :* `s.[i]`, les éléments sont indexés à partir de zéro, et l'accès se fait en temps $\mathcal{O}(1)$.
— *Longueur :* `String.length : string -> int`, en temps $\mathcal{O}(1)$.
— *Concaténation :* `s ^ t` renvoie une nouvelle chaîne formée des caractères de `s` suivis de ceux de `t`. Complexité $\mathcal{O}(|s|+|t|)$.

Dépassement de capacité en OCaml : Comme dit plus haut, les calculs sur les `int` de OCaml sont effectués modulo 2^{63} puis le résultat est ramené dans l'intervalle $[\![-2^{62}, 2^{62}-1]\!]$.