Chapitre 1

Types, variables, expressions

Mots-clés :
- valeurs et types,
- opérations et expressions,
- variables non mutables.

Objectifs :
- comprendre les notions de valeur et de type,
- connaître les types de base,
- apprendre à construire des types,
- savoir manipuler des valeurs,
- savoir manipuler des variables non mutables,
- savoir formaliser un problème simple,
- prendre en main l'environnement de développement.

Un **algorithme** est la **description du calcul d'un résultat en fonction d'un paramètre**, par exemple le calcul de la trajectoire d'un projectile en fonction de sa position et de sa vitesse initiales. Lorsqu'on effectue le calcul ainsi décrit, le paramètre est une *valeur*, composée éventuellement de valeurs plus élémentaires : dans l'exemple, le paramètre du calcul de trajectoire est composé d'une position et d'une vitesse initiales ; la position, de même que la vitesse, est composée de coordonnées ; mathématiquement, une coordonnée est un nombre réel [1].

Afin de décrire des calculs, il est important de savoir quelles sont les valeurs qu'il est possible de manipuler, et ceci nous amène à la notion de *type*. Nous

1. Il est important de préciser « mathématiquement ». En effet, si une coordonnée peut prendre n'importe quelle valeur réelle en mathématiques, il n'en est pas forcément de même en informatique, comme nous le voyons plus loin.

considérons ici[1] qu'**un type est un ensemble de valeurs**. Tout langage d'algorithmes ou de programmation fournit *des types de base et des opérations* permettant de manipuler les valeurs de ces types, par exemple pour la manipulation de valeurs numériques ou textuelles. La plupart des langages fournissent aussi *des mécanismes permettant de construire des types plus élaborés* à partir des types préexistants, par exemple pour définir une vitesse à partir de coordonnées.

Il est alors possible de décrire des calculs de valeurs par des *expressions*, par exemple pour exprimer le calcul d'une coordonnée d'un point d'une trajectoire par une expression mathématique.

Pour des raisons de lisibilité, voire de généralité des calculs décrits, il est souvent utile d'utiliser des *variables* dans l'écriture d'algorithmes.

Toutes ces notions font l'objet du présent chapitre. Plus précisément, ces notions sont des outils, qu'il faut connaître et comprendre. Ces outils servent à résoudre des problèmes, il faut donc apprendre aussi à s'en servir, c'est-à-dire à *formaliser* les données du problème et le mode de résolution. Par exemple, pour le calcul de la trajectoire d'un projectile, quels types utiliser pour représenter les valeurs devant être manipulées, depuis une coordonnée jusqu'à une trajectoire ? Quelles opérations effectuer pour calculer la trajectoire en fonction des données initiales ?

Dit autrement, il est important de savoir à la fois :
— *traduire les données d'un problème en données informatiques*. Par exemple, si l'on souhaite réaliser un logiciel de retouche d'images, comment représente-t-on les valeurs qui caractérisent une image ?
— *manipuler des données informatiques*, c'est-à-dire *savoir décrire les calculs devant être effectués pour produire les résultats*, par exemple pour rehausser le contraste dans une image.

1.1 Notions essentielles

1.1.1 Types et comparaisons de valeurs

Les **types de base** fournis par le langage `OCaml` sont listés dans le tableau 1.1, et décrits dans la section 1.1.2 (page 19).

Il est possible de définir des **types construits** à partir de ces types de base, par exemple les *types produits* et les *types structurés* décrits dans la section 1.1.3 (page 24).

1. Il existe plusieurs approches de la notion de type.

NB1. Toute valeur appartient à un type unique.

type	description	exemples de valeurs
`int`	valeurs entières	`0`, `-5`, `222`
`float`	valeurs flottantes	`.2`, `-3.14`, `5.2e3`, `2.0`
`char`	caractères (256 valeurs)	`'a'`, `'?'`, `'1'`
`string`	chaînes de caractères	`"Hello World!"`
`bool`	valeurs booléennes	`true`, `false`
`unit`	valeur unique « vide »	`()`

TABLE 1.1 – Types de base.

NB2. Au sein d'un type [1], deux valeurs peuvent être comparées en utilisant les **opérateurs de comparaison** :

`=`	égal
`<>`	différent
`>`	strictement supérieur
`>=`	supérieur ou égal
`<`	strictement inférieur
`<=`	inférieur ou égal

— quel que soit l'opérateur de comparaison, *les deux opérandes doivent être du même type*. Par exemple, on ne peut pas comparer directement les valeurs `2` (type `int`) et `2.0` (type `float`) ;

— *la valeur d'une expression de comparaison est toujours un booléen*, c'est-à-dire une valeur de type `bool` (*cf.* section 1.1.2.5 page 23) : la valeur de `2 <> 3` est `true`, celle de `2 < 3` est `true`, celle de `2 >= 3`, qui est égale à celle de `not(2 < 3)`, est `false` ;

1.1.2 Types de base

Un type de base est un ensemble de valeurs, sur lesquelles des opérations prédéfinies peuvent être appliquées.

1. Il n'est pas possible de comparer des valeurs de types différents. Par exemple, comme le type `int` n'est pas le type `float`, il n'est *pas possible de comparer directement* une valeur de type `int` et une valeur de type `float`. Il est possible de *convertir* une valeur de type `int` en valeur de type `float`, et réciproquement, afin d'effectuer de telles comparaisons (*cf.* section 2.1.4 page 63). Si certains langages autorisent une comparaison entre valeurs de types différents, c'est parce qu'ils effectuent automatiquement des conversions : dans ce cas, il est nécessaire d'être très attentif à ces conversions implicites afin d'éviter des erreurs de conception.

1.1.2.1 Type `int`

Le **type** `int` est un ensemble des valeurs numériques correspondant à un sous-ensemble de l'ensemble des **entiers** $\mathbb{Z}$. Ces valeurs sont comprises [1] entre -2^{62} et $2^{62}-1$. Les opérations prédéfinies sont les opérations usuelles : addition, soustraction, multiplication, et les deux opérations de la division entière : quotient et reste.

Opérateurs sur les `int` :

`+`	addition
`-`	soustraction
`*`	multiplication
`/`	quotient de la division entière
`mod`	reste de la division entière (modulo)

Voici quelques exemples d'interactions avec l'interpréteur [2] :

```
# 5 ;;                          # (4 * 3 + 2) / 3 ;;
- : int = 5                    - : int = 4
# 5 * 2 + 3 ;;                 # (4 * 3 + 2) mod 3 ;;
- : int = 13                   - : int = 2
# 5 / 2 ;;                     # -7 / 3 ;;
- : int = 2                    - : int = -2
# 5 mod 2 ;;                   # -7 mod 3 ;;
- : int = 1                    - : int = -1
```

Chaque colonne correspond à la saisie de quatre expressions dans l'interpréteur `ocaml`, et aux quatre réponses de l'interpréteur :

— d'abord, l'expression est saisie. Le caractère `#` est affiché par l'interpréteur ; il signifie que l'interpréteur attend une « phrase » `OCaml`. Il faut noter que chaque phrase `OCaml` se termine par deux points-virgules ;

— puis la réponse de l'interpréteur est affichée. Par exemple, pour la phrase `5 ;;`, l'interpréteur répond `- : int = 5`. Le tiret signifie qu'il n'y a pas de nom associé à la valeur (*cf.* section 1.1.5 page 30) ; `int = 5` signifie que l'interpréteur a reconnu une expression de type `int` dont la valeur est `5`.

NB.

— Comme en mathématiques, une *division par* 0*, quotient ou reste, n'est pas définie* : *cf.* section 1.3.2.1 page 41 pour plus de précisions.

1. Nous supposons ici qu'il s'agit d'une version 64 bits. Dans une version 32 bits, les valeurs entières sont comprises entre -2^{30} et $2^{30}-1$: *cf.* section 1.3.2.1 page 41.

2. Pour plus de précisions sur l'interpréteur `ocaml`, *cf.* section 11.2 page 412.

— Les *comparaisons* de valeurs de type `int` ont les mêmes résultats que les comparaisons des entiers correspondants de $\mathbb{Z}$.

1.1.2.2 Type `float`

Le **type** `float` comprend un ensemble des valeurs numériques correspondant à un sous-ensemble de l'ensemble des **rationnels** $\mathbb{Q}$.

Opérateurs sur les `float` :

`+.`	addition
`-.`	soustraction
`*.`	multiplication
`/.`	division
`**`	exponentiation

```
# 5.2 ;;
- : float = 5.2
# 5.45e3 ;;
- : float = 5450.
# 5.45e-3;;
- : float = 0.00545
```

```
# 5.45e210 ;;
- : float = 5.45e+210
# 5.45e-210;;
- : float = 5.45e-210
# 5.0 ** 2.0 ;;
- : float = 25.
```

NB.

— Comme en mathématiques, nous considérons qu'une *division par* `0.0` *n'est pas définie* : *cf.* section 1.3.2.2 page 43 pour plus de précisions.

— Les *comparaisons* de valeurs de type `float` ont les mêmes résultats que les comparaisons des rationnels correspondants de $\mathbb{Q}$.

— Si, en mathématiques, les entiers sont des nombres rationnels particuliers, ce n'est pas le cas pour les valeurs de type `int` et les valeurs de type `float` : *une valeur de type* ***`int`*** *n'est pas de type* ***`float`****, et réciproquement.* Comme dit précédemment, une valeur appartient à un type unique. En particulier, il n'est pas possible d'appliquer des opérateurs définis pour un type sur des valeurs d'un autre type [1].

1. Cette remarque n'est qu'une généralisation de ce qui a déjà été dit à propos des opérateurs de comparaison. Par exemple, si l'on souhaite additionner une valeur de type `int` et une valeur de type `float`, il est nécessaire de convertir l'une des deux valeurs dans le type de l'autre valeur. Il y a donc deux possibilités : convertir la valeur de type `int` en `float`, et appliquer l'addition définie pour le type `float` ; ou convertir la valeur de type `float` en `int`, et appliquer l'addition définie pour le type `int`. Certains langages effectuent des conversions automatiques : dans ce cas, il est nécessaire d'être attentif à la conversion choisie par le langage, pour vérifier si elle correspond bien à ce que l'on souhaite faire.

```
# 2.5 + 3.5 ;;
Error: This expression has type float but an expression was expected
of type int
# 2 + 3.5 ;;
Error: This expression has type float but an expression was expected
of type int
```

On peut noter que l'interpréteur `ocaml` souligne la partie de l'expression qui pose problème. Dans les deux cas ci-dessus, l'opérateur correspond à l'addition entière, l'interpréteur attend donc que tous les paramètres soient de type `int`.

1.1.2.3 Type `char`

Le type `char` contient 256 valeurs permettant de représenter des **caractères**. Les valeurs de type `char` sont représentées entre apostrophes[1], par exemple : `'a'`, `'2'`, `'?'`, ou encore le caractère de retour à la ligne `'\n'`.

```
# 'a' ;;
- : char = 'a'
# 'R' ;;
- : char = 'R'
# '1' ;;
- : char = '1'
```

```
# '*' ;;
- : char = '*'
# '\n' ;;
- : char = '\n'
# '\255' ;;
- : char = '\255'
```

Les caractères sont ordonnés selon le standard ISO 8859-1. En particulier, les 128 premiers caractères correspondent au code ASCII (American Standard Code for Information Interchange).

Il existe un **ordre sur les caractères** : c'est l'ordre du code ASCII pour les 128 premiers caractères. En particulier, l'ordre sur les lettres minuscules est l'ordre alphabétique (il en est de même pour les lettres majuscules) ; de même, les caractères correspondant aux chiffres sont consécutifs. Par exemple, la lettre 'a' est inférieure à toutes les autres lettres minuscules.

NB. *Il ne faut pas confondre un caractère numérique avec l'entier correspondant* ; par exemple, le caractère `'2'` n'a rien à voir avec l'entier `2`.

1.1.2.4 Type `string`

Le type `string` permet de représenter des **chaînes de caractères**. Les valeurs de type `string` sont représentées entre guillemets.

1. Guillemet droit simple, ou simple quote.

Opérateur sur les `string` : l'opérateur de *concaténation* est noté `^`, il permet de *coller* deux chaînes l'une à la suite de l'autre.

```
# "Hello" ;;
- : string = "Hello"
# "World ! \n" ;;
- : string = "World ! \n"
```

```
# "Hello" ^ " " ^ "World ! \n" ;;
- : string = "Hello World ! \n"
```

Il existe un **ordre sur les chaînes de caractères** : c'est l'ordre lexicographique induit par l'ordre du type `char` : par exemple, la chaîne `"alentour"` est inférieure aux chaînes `"alentours"`, `"autour"`, `"bismuth"`, etc.

1.1.2.5 Type `bool`

Le type `bool` permet de représenter les valeurs de vérité *vrai* et *faux* de l'algèbre de Boole, bien connue en logique mathématique. Il contient donc exactement 2 **valeurs booléennes (ou booléens)** : `true` et `false`. En particulier, *le résultat d'une comparaison est une valeur booléenne*, quel que soit le type des opérandes.

```
# true ;;
- : bool = true
# false ;;
- : bool = false
# "alpha" < "alphabetique" ;;
- : bool = true
```

```
# 2 > 3 ;;
- : bool = false
# 'a' < 'b' ;;
- : bool = true
# "alpha" < "alentour" ;;
- : bool = false
```

NB. Comme dit précédemment, les deux opérandes d'une comparaison doivent être du même type :

```
# 3 < 4.5 ;;
Error: This expression has type float but an expression was expected of
type int
```

Opérateurs sur les `bool` : on peut construire des expressions booléennes plus élaborées en utilisant les opérateurs logiques suivants :

`&&`	et logique
`||`	ou logique
`not`	négation logique

Les opérateurs `&&` et `||` sont des opérateurs binaires, l'opérateur `not` est un opérateur unaire. Ces opérateurs sont définis selon les **tables de vérité** suivantes :

x	y	x && y
true	true	true
true	false	false
false	true	false
false	false	false

x	y	x || y
true	true	true
true	false	true
false	true	true
false	false	false

x	not(x)
true	false
false	true

Ci-après figurent quelques exemples d'expressions booléennes, ainsi que leurs valeurs :

```
# (2 > 3) && ('c' < 'k')  # ('c' > 'k') && ('c' < 'z')
- : bool = false          - : bool = false
# (2 > 3) || ('c' < 'k')  # not(2 > 3) || (('c' > 'k') && ('c' < 'z'))
- : bool = true           - : bool = true
```

1.1.2.6 Type `unit`

Le type `unit` ne contient qu'une seule valeur, notée `()`. Nous en parlons dans la partie III de ce livre (*cf.* page 241).

1.1.3 Types construits

1.1.3.1 Types énumérés

Un type énuméré est un **ensemble de valeurs déclarées explicitement** ; il permet de représenter directement un ensemble fini d'informations de même nature. Par exemple, le type suivant permet de représenter les noms des jours de la semaine :

```
# type t_day = MONDAY | TUESDAY | WEDNESDAY | THURSDAY | FRIDAY
| SATURDAY | SUNDAY ;;
type t_day = MONDAY | TUESDAY | WEDNESDAY | THURSDAY
| FRIDAY | SATURDAY | SUNDAY
```

L'interpréteur répond qu'il a bien compris que l'on déclare un type, dont le nom est `t_day`, et qui contient les valeurs listées à la suite.

La déclaration d'un type énuméré se fait suivant la syntaxe suivante :

```
type type_name = VALUE_1 | VALUE_2 | ··· | VALUE_n ;;
```

où `type` est un mot réservé du langage `OCaml` et `type_name` est un identificateur (*cf.* section 1.2 page 34). Chaque valeur est écrite en lettres majuscules. Les valeurs sont séparées par une barre verticale.

NB1. Même si une valeur d'un type énuméré est écrite comme une suite de caractères (comme toute valeur de tout type), ce n'est pas une chaîne de