Pascal Lienhardt
Samuel Peltier
Agnès Arnould
Laurent Fuchs

L1

# Bases en algorithmique et en programmation

Cours et 120 exercices corrigés



ellipses

Références sciences

# Bases en algorithmique et en programmation

Cours et 120 exercices corrigés

Pascal Lienhardt
Samuel Peltier
Agnès Arnould
Laurent Fuchs

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

*Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr*

ISBN 9782340-047761
©Ellipses Édition Marketing S.A., 2021
8/10 rue la Quintinie 75015 Paris

DANGER LE PHOTOCOPILLAGE TUE LE LIVRE

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5.2° et 3°a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective », et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit constituerait une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

# Introduction

## Cadre

Schématiquement, le domaine de l'**Algorithmique - Programmation** est celui du **calcul**. Il peut s'agir aussi bien de calculer la trajectoire d'un projectile, de retoucher une image, de produire une séquence d'images pour un film d'animation, de contrôler un personnage dans un jeu vidéo, de calculer un itinéraire routier, de simuler le déplacement d'une rame de métro automatique, etc. Dans tous les cas, les questions suivantes se posent :

1. ***que calcule-t-on, et à partir de quoi ?*** Par exemple, la trajectoire d'un projectile peut être définie comme une suite de points. Si l'on calcule une trajectoire, considère-t-on des points 2D, des points 3D ? Combien de points calcule-t-on ? Que prend-on en compte pour le calcul : la gravité, la vitesse initiale, la poussée du vent et / ou la résistance de l'air ? En fait, il s'agit ici de *décrire précisément* le problème et de le *formaliser*, c'est-à-dire d'identifier les informations qu'il faut manipuler, en particulier les données initiales et les résultats attendus, et de choisir comment les représenter : ceci permet d'envisager de résoudre le problème en calculant le *résultat* à partir du *paramètre*. Le résultat peut être composé de plusieurs valeurs : par exemple une trajectoire est composée de plusieurs points, un point est composé de plusieurs coordonnées. De même, le paramètre peut être composé de plusieurs valeurs : par exemple, une accélération, comme une vitesse, est décrite par plusieurs coordonnées ;
2. ***comment le calcule-t-on ?*** Il s'agit ici de *concevoir* et de *programmer* un *algorithme*, c'est-à-dire un mode de calcul permettant de calculer le résultat à partir du paramètre. Par exemple, la trajectoire d'un projectile peut être décrite par une équation mathématique, qui modélise le phénomène physique : la conception d'un algorithme de calcul de trajectoire peut donc se baser sur une telle équation. Plus généralement, suivant les problèmes, il peut exister, ou non, des bases, voire des stratégies, sur lesquelles appuyer la conception d'un algorithme. Programmer un

algorithme consiste à choisir un langage de programmation, et à décrire l'algorithme dans ce langage, en utilisant les mécanismes de construction de structures de données et de fonctions du langage ;

3. ***ce calcul est-il correct ?*** Comme il est très facile et très fréquent de se tromper, il est toujours nécessaire de *vérifier la validité* d'un algorithme. Les erreurs sont multiples : l'algorithme peut être intrinsèquement faux, suite à une erreur de conception. Il peut y avoir des erreurs de programmation, par exemple une erreur dans le choix d'une structure de données qui ne permet pas de représenter toutes les valeurs nécessaires, ou une simple erreur de syntaxe. Il existe des méthodes et des outils permettant de *prouver* des algorithmes. A minima, il est nécessaire de *tester* un algorithme que l'on vient d'écrire [1] ;
4. ***quel est le coût d'exécution*** [2] ***de ce calcul ?*** Effectuer un calcul nécessite des ressources. Un algorithme qui s'exécute en ordinateur nécessite des ressources en mémoire : c'est le *coût en espace* ; le *coût en temps* est lié au nombre d'opérations de base effectuées par l'ordinateur pour calculer le résultat.
5. Il ne faut pas non plus oublier le temps de conception, de développement et de maintenance, c'est-à-dire le temps passé par la ou les personnes qui concoivent et / ou qui programment l'algorithme, et le temps qui sera passé à maintenir les codes résultants. Cet aspect n'est pas approfondi dans ce livre.

## Objectifs

L'objectif de ce cours est d'étudier les notions de base de l'Algorithmique - Programmation, ainsi que leur mise en œuvre dans des exemples « simples ». Plus précisément, cela concerne :

1. la *« nature » des valeurs* manipulées dans un algorithme, et les *opérations* applicables à ces valeurs. Il s'agit ici d'étudier la notion de *type* ou de *structure de données*, qui permet de *représenter* des informations. Ceci comprend en particulier :
   - les types de base et les opérations associées, ainsi que les mécanismes permettant de construire de nouveaux types et de nouvelles opérations,

1. Il existe des méthodes et des outils permettant de réaliser des tests systématiquement, qui sortent du cadre de ce livre ; nous nous contentons ici d'évoquer certaines questions de base à se poser lors de l'écriture d'algorithmes.

2. Cette question n'est pas abordée dans ce livre.

   - la formalisation, c'est-à-dire le choix de types pour la représentation d'informations devant être manipulées dans un algorithme, en particulier le paramètre et le résultat ;

2. des *algorithmes de base* manipulant ces types ou structures de données. Il s'agit ici d'étudier :
   - les outils permettant de *formaliser un algorithme*, en particulier les notions de *fonction*, de *conditionnelle*, de *récursivité*, d'*itérativité*,
   - des algorithmes classiques de manipulation de certaines structures de données, par exemple les *listes* et les *tableaux*,
   - des exemples d'algorithmes pour la résolution de divers problèmes relativement « simples »,
   - des idées de base sous-jacentes à différents algorithmes.

## Langage

D'une part, *un algorithme doit être exprimé de manière non ambiguë* : il est donc nécessaire de disposer d'un langage adéquat. Ce *langage d'algorithmes* n'est pas nécessairement un langage de programmation pouvant être exécuté sur un ordinateur, car il peut être utile, pour décrire un algorithme, de s'abstraire de contraintes inhérentes à un langage de programmation.

D'autre part, en supposant que l'on ait choisi un langage de programmation, on peut souvent *implanter*[1] *un algorithme de différentes manières* aussi pertinentes les unes que les autres. En règle générale, il n'y a donc pas une implantation unique d'un algorithme dans un langage de programmation donné.

Enfin, un algorithme peut être programmé en utilisant différents langages de programmation. Or, chaque langage de programmation a ses propres spécificités, et peut imposer certaines constructions, suivant les fonctionnalités offertes par le langage. En conséquence, certains langages de programmation peuvent être plus adaptés que d'autres (selon certains critères) pour l'implantation d'un algorithme.

C'est pourquoi un algorithme est souvent écrit dans un langage d'algorithmes, appelé aussi langage *abstrait*, puis traduit dans un langage de programmation, appelé aussi langage *concret*[2].

Il est à noter que la phase de programmation peut « contraindre » l'algorithme, par exemple en apportant des restrictions à son domaine de validité.

1. On dit aussi *implémenter* ou *programmer*. Le *code* est le texte qui résulte de cette activité.

2. En fait, si les deux phases décrites ici correspondent bien au terme *Algorithmique - Programmation*, le processus qui va de la conception d'un algorithme à sa programmation n'est pas toujours structuré de cette manière, et peut prendre de nombreuses formes.

Nous avons choisi ici d'*utiliser un unique langage* :

1. ce langage est utilisé pour exprimer les algorithmes de manière suffisamment générale pour qu'ils puissent être traduits dans de nombreux langages de programmation ;
2. ce langage permet aussi de programmer les algorithmes, puisqu'il s'agit en fait d'une restriction du langage de programmation `OCaml`. Cette restriction est mise en oeuvre par le biais de *conventions d'écriture* ;
3. si le langage se base sur `OCaml`, ce cours n'est pas un cours de programmation en langage `OCaml`. De nombreux aspects de ce langage ne sont pas étudiés ; de même, certaines constructions présentées pourraient s'écrire différemment en `OCaml` [1], et certaines notions définies dans ce livre seraient sans doute définies différemment dans un cours de programmation `OCaml` (de la même manière qu'un même nom peut désigner des notions différentes dans des langages d'algorithmes ou de programmation différents).
4. En revanche, les aspects du langage que nous avons choisis permettent de mettre en évidence les notions de base importantes de l'Algorithmique - Programmation. Comme dit précédemment, il est ainsi possible de traduire assez directement la plupart des algorithmes présentés ici dans de nombreux autres langages de programmation ;
5. le choix d'utiliser un langage dérivé de `OCaml` est dû aux caractéristiques du langage `OCaml` que nous considérons comme essentielles pour un enseignement de bases de l'Algorithmique - Programmation (en particulier concernant le *typage*), ainsi qu'à des caractéristiques facilitant la mise à disposition d'outils (en particulier liées au système de types) et l'expérimentation (en particulier l'existence d'un *interpréteur*). Comme tout langage, celui-ci présente certains inconvénients, que nous jugeons suffisamment mineurs pour ne pas poser de problèmes significatifs compte-tenu de l'objectif suivi ici [2].

En résumé, ce cours porte sur l'étude de bases de l'Algorithmique - Programmation en utilisant un langage d'algorithmes dérivé du langage `OCaml`. Nous nous soucions donc essentiellement des aspects liés à l'algorithmique et à la programmation de base. La manière dont les choses sont effectivement implantées en langage `OCaml` ne sont pas étudiées ici, sauf mention contraire

1. Pour plus de précisions sur le langage `Ocaml`, ses caractéristiques, ses utilisations et ses utilisateurs, voir le site `https://ocaml.org`

2. Par exemple, comme nous n'utilisons qu'une restriction du langage `OCaml`, certaines constructions qui correspondent à des erreurs dans notre cadre sont autorisées par le langage `OCaml` en toute généralité. Ceci dit, les messages de l'interpréteur `ocaml` permettent de repérer ces erreurs.

explicite. Si l'on fait une analogie avec l'apprentissage de la conduite automobile, il s'agit d'apprendre les mécanismes de base de la conduite, aussi indépendamment que possible du véhicule utilisé.

## Mise en œuvre

Afin de simplifier les explications, nous considérerons que le lecteur dispose d'un accès à un interpréteur `ocaml`. Il existe par exemple des interpréteurs `ocaml` en ligne (cf. section 11.2.1 page 412) ; il est aussi possible d'installer un interpréteur [1] `ocaml` sur son ordinateur (*cf.* `https://ocaml.org/`, et section 11.3 page 419). Des solutions plus élaborées peuvent aussi être mises en place (*cf.* section 11.3.4 page 442). Dans tous les cas, il est bien sûr recommandé de sauvegarder le(s) code(s) écrit(s) dans un ou plusieurs fichiers.

Plus précisément, l'interpréteur `ocaml` :

1. permet de *saisir des expressions* écrites en langage `OCaml` ;
2. *vérifie la correction syntaxique* des expressions, puis traduit les expressions correctes de manière à ce qu'elles soient « compréhensibles » pour la machine : dit autrement, les expressions sont *compilées* ;
3. permet de *lancer l'exécution des calculs*. Pour chaque expression correcte, l'interpréteur répond par une « phrase » donnant les principales informations concernant l'expression.

**NB :** il est conseillé de ***tester systématiquement toutes les expressions*** décrites dans ce cours ***en utilisant l'interpréteur***, afin de bien comprendre le sens de ces expressions et le fonctionnement de l'interpréteur [2].

## Remerciements

Les auteurs remercient chaleureusement toutes les personnes dont les propos ou les écrits ont participé à la réflexion ayant abouti à ce livre, et en particulier Rita Zrour, Xavier Skapin, Hakim Ferrier-Belhaouari, Gaëlle Largeteau-Skapin, Yannick Degardin, Eric Andres, Stéphane Jean, Gilles Subrenat, Sylvie Alayrangues, Daniel Meneveaux, Wassim Rharbaoui, Patrice Naudin, Thierry Urruty, Simon Courtin, Lydie Richaume.

Les auteurs remercient tout particulièrement Annie Geniet, Michaël Lienhardt, Yves Bertrand et Catherine Dubois.

1. Il est recommandé d'installer aussi la librairie graphique `graphics`, qui, suivant les versions, figure ou non dans la distribution de base (*cf.* `https://opam.ocaml.org/packages/`).

2. La plupart des codes décrits dans ce livre, ainsi que les codes des fonctions utilitaires utilisées, sont disponibles sur le site des éditions Ellipses : http ://www.editions-ellipses.fr.

*à Anatole, qui a servi de cible,*
*à Chiara, qui a rectifié le tir,*
*à Dominique, qui a pesé sur la détente,*
*à Cécile, pour tout le reste !*

# Table des matières

Introduction ..... 3

I Notions élémentaires ..... 15

1 Types, variables, expressions ..... 17

1.1 Notions essentielles ..... 18

1.2 Conventions d'écriture ..... 34

1.3 Compléments ..... 37

1.4 Exercices ..... 44

1.5 Corrigés ..... 48

2 Fonctions ..... 55

2.1 Notions essentielles ..... 56

2.2 Conventions d'écriture ..... 64

2.3 Compléments ..... 66

2.4 Exercices ..... 70

2.5 Corrigés ..... 74

3 Conditionnelle ..... 85

3.1 Notions essentielles ..... 86

3.2 Conventions d'écriture ..... 94

3.3 Compléments ..... 95

3.4 Exercices ..... 96

3.5 Corrigés ..... 100

II Récursivité ..... 111

4 Composition de fonctions ..... 113

4.1 Notions essentielles ..... 114

4.2 Compléments ..... 121

4.3 Exercices ..... 129

4.4 Corrigés ..... 135

5 Fonctions récursives ..... 149

5.1 Notions essentielles ..... 150

5.2 Conventions d'écriture ..... 160

5.3 Compléments ..... 160

5.4 Exercices ..... 168

5.5 Corrigés ..... 173

6 Listes ..... 189

6.1 Notions essentielles ..... 191

6.2 Conventions d'écriture ..... 203

6.3 Compléments ..... 206
6.4 Exercices ..... 211
6.5 Corrigés ..... 217
III Itérativité ..... 239
7 Variables mutables, affectation ..... 241
7.1 Notions essentielles ..... 243
7.2 Conventions d'écriture ..... 253
7.3 Exercices ..... 256
7.4 Corrigés ..... 265
8 Instructions itératives ..... 281
8.1 Notions essentielles ..... 282
8.2 Conventions d'écriture ..... 295
8.3 Compléments ..... 297
8.4 Exercices ..... 302
8.5 Corrigés ..... 308
9 Tableaux ..... 325
9.1 Notions essentielles ..... 325
9.2 Compléments ..... 332
9.3 Exercices ..... 341
9.4 Corrigés ..... 346

IV Annexes ..... 361

10 Compléments ..... 363

10.1 Graphique ..... 363

10.2 Exercice : simulation du jeu de bataille ..... 372

10.3 Exercice : jeu de serpent ..... 384

11 Aspects « techniques » ..... 403

11.1 Fonctions utilitaires ..... 403

11.2 Interprétation ..... 412

11.3 Installation de OCaml ..... 419

Index ..... 445

# Première partie

# Notions élémentaires

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

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

caractères (de même que la valeur entière `123` n'est pas une chaîne de caractères) : par exemple, le type de la valeur `WEDNESDAY` est `t_day`, cette valeur ne peut donc pas être comparée à une chaîne de caractères.

**NB2.** Deux valeurs d'un type énuméré peuvent être comparées. Dans notre cadre, seuls les comparateurs d'égalité et de différence sont utilisés. Pour plus de précisions, voir section 1.3.1.2 page 39.

```
# WEDNESDAY ;;
- : t_day = WEDNESDAY
# WEDNESDAY = "WEDNESDAY" ;;
Error: This expression has type string but an expression was expected
       of type t_day
# WEDNESDAY = FRIDAY ;;
- : bool = false
```

#### 1.1.3.2 Types produits

Les types produits sont définis comme des **produits cartésiens de plusieurs types**[1] : par exemple `int * char` ou `int * int * unit * string` sont des types produits.

Les valeurs d'un type produit sont des **n-uplets** ou **tuples** : elles sont composées de valeurs des types intervenant dans le produit. Ces valeurs sont représentées entre parenthèses, et sont séparées par une virgule : par exemple, `(1,'A')` et `(-124,'?')` sont deux valeurs du type `int * char`. Les valeurs constituant un n-uplet sont appelées ses éléments, ou ses composants.

Il est possible de comparer deux n-uplets de même type : ils sont comparés suivant **l'ordre lexicographique**, c'est-à-dire que leurs premiers éléments sont comparés ; si les premiers composants sont égaux, les seconds éléments sont comparés ; si les premiers composants sont égaux et si les deuxièmes composants sont égaux, les troisièmes sont comparés, etc. Plus précisément, considérons deux n-uplets `a = (a_1, ⋯, a_n)` et `b = (b_1, ⋯, b_n)` de même type :

1. `a < b` si et seulement si il existe `k` compris entre `1` et `n`, tel que : `a_k < b_k`, avec `a_i = b_i` pour tout `i` compris entre `1` et `k-1` ;

1. Pour rappel, le produit cartésien $E_1 \times E_2$ de deux ensembles $E_1$ et $E_2$ est l'ensemble composé de tous les couples $(e_1, e_2)$, où $e_1 \in E_1$ et $e_2 \in E_2$. Plus généralement, le produit cartésien $E_1 \times \cdots \times E_n = \prod_{i=1}^{n} E_i$ de $n$ ensembles $E_1$, $\cdots$, $E_n$, est l'ensemble de tous les $n$-uplets $(e_1, \cdots, e_n)$, où, pour tout $i$ compris entre 1 et $n$, $e_i \in E_i$. Un $n$-uplet est appelé aussi un tuple.

2. `a = b` si tous les composants de `a` et de `b` sont deux à deux égaux, c'est-à-dire que, pour tout `i` compris entre `1` et `n`, `a_i = b_i` ;
3. `a > b` si et seulement si `b < a`

par exemple :

```
# (5,7,3) < (6,1,2) ;;
- : bool = true
# (5,7,3) < (5,7,4) ;;
- : bool = true
# (5,7,3) < (5,7,2) ;;
- : bool = false
# ("alentour", "autour") < ("alentours", "alphabet") ;;
- : bool = true
# ("alentour", "autour") < ("alentour", "autours") ;;
- : bool = true
```

Il est possible de donner un nom à un type produit, par exemple :

```
# type t_int_bool = int * bool ;;
type t_int_bool = int * bool
```

L'interpréteur répond qu'il a bien compris que l'on déclare un type, dont le nom est `t_int_bool`, et qui est égal au type produit `int * bool`.

Plus généralement, la déclaration d'un type produit se fait suivant la syntaxe suivante :

```
type type_name = type_name_1 * ··· * type_name_n ;;
```

où `type` est un mot réservé du langage `OCaml`, `type_name` est un identificateur et `type_name_1`, ..., `type_name_n` sont des noms de types déjà définis.

**NB1.** Un type produit peut être défini à partir de tous types, que ce soient des types de base ou non. Par exemple, la valeur ci-après est un couple, dont le premier composant est une valeur de type `float`, et le deuxième composant est un couple formé d'une valeur de type `int` et d'une valeur de type `bool`. Le type est donc `float * (int * bool)`, et les parenthèses sont importantes !

```
# (3.5, (4, true)) ;;
- : float * (int * bool) = (3.5, (4, true))
```

**NB2.** Attention à ne pas confondre des types produits. Par exemple, la valeur `(1, (2, 3))` est un couple, composé d'une valeur entière et d'un couple d'entiers, alors que la valeur `(1, 2, 3)` est un triplet d'entiers. L'interpréteur ne dit pas autre chose ci-après : en effet, un type quelconque est désigné en

OCaml par une apostrophe suivie d'une lettre : `'a * 'b * 'c` désigne donc un type composé de triplets de valeurs quelconques.

```
# (1, (2, 3)) = (1, 2, 3) ;;
Error: This expression has type 'a * 'b * 'c
       but an expression was expected of type int * (int * int)
```

#### 1.1.3.3 Types structurés

Une *structure* est **similaire à un n-uplet dont les différents composants sont nommés**, en particulier afin de pouvoir y accéder facilement.

La plupart des langages de programmation offrent la possibilité de manipuler des *types structurés*. En OCaml, comme dans beaucoup d'autres langages, les types structurés ne font pas partie des types de base, il est donc *nécessaire de les déclarer avant de les utiliser* :

```
type type_name =
    {name_1 : type_name_1 ; ··· ; name_n : type_name_n}
```

Le mot-clé **type** indique une déclaration de type, le reste est la définition d'un type structuré, dont les composants s'appellent respectivement `name_1`, ···, `name_n`, et ces composants sont de types respectifs `type_name_1`, ..., `type_name_n`.

Une valeur de ce type s'écrit :

```
{name_1 = value_1 ; ··· ; name_n = value_n}
```

où `value_1`, ···, `value_n` sont des valeurs de types respectifs `type_name_1`, ···, `type_name_n`.

Par exemple, on s'intéresse à une application graphique 2D dans laquelle il est nécessaire de manipuler des points et des vecteurs du plan. On choisit de définir les types suivants :

```
# type t_point = {x : float ; y : float} ;;
type t_point = { x : float; y : float;}
# type t_vector = {dx : float ; dy : float} ;;
type t_vector = { dx : float; dy : float; }
```

Voici des valeurs de ces types (bien sûr, les valeurs des composants doivent être compatibles avec leurs types ; de même, aucune valeur ne peut être définie si son type n'est pas défini : *cf.* les trois dernières expressions, qui contiennent des erreurs) :

```
# {x = 0.0 ; y = 0.0} ;;
- : t_point = {x = 0.; y = 0.}
```

```
# {x = 1.0 ; y = 1.0} ;;
- : t_point = {x = 1.; y = 1.}
# {dx = 1.0 ; dy = 0.0} ;;
- : t_vector = {dx = 1.; dy = 0.}
# {dx = 0.0 ; dy = 1.0} ;;
- : t_vector = {dx = 0.; dy = 1.}
# {x = 2.0 ; y = 0} ;;
Error: This expression has type int but an expression was expected of
type float
# {x = 0.0 ; dy = 1.0} ;;
Error: The record field dy belongs to the type t_vector
       but is mixed here with fields of type t_point
# {a = 0 ; b = 0} ;;
Error: Unbound record field a
```

**NB1.** Un composant d'un type structuré est aussi appelé **champ**[1]. Par exemple, une valeur de type `t_point` (resp. `t_vector`) possède deux **champs** `x` et `y` de type `float` (resp. `dx` et `dy` de type `float`).

**NB2.** Deux valeurs d'un même type structuré peuvent être comparées[2]. L'ordre est l'ordre lexicographique sur les champs[3]. Par exemple, l'expression `{x = 0.0 ; y = 3.0} < {x = 1.0 ; y = 2.0}` a pour valeur `true`.

### 1.1.4 Expressions

Une **expression est la description d'un calcul** dans laquelle interviennent des valeurs et des opérateurs. Bien sûr, l'expression doit être **valide**, c'est-à-dire que **tout opérateur doit être applicable à ses opérandes**. À l'extrême, une expression peut être réduite à une valeur.

```
# 7.2 ;;
- : float = 7.2
# 3 + 5 * 2;;
- : int = 13
# (3 + 5) * 2 ;;
- : int = 16
```

1. Un élément d'un type structuré est aussi appelé *enregistrement*, ou *record* en anglais. Par exemple, le message d'erreur `Unbound record field a` signifie que le nom de champ `a` n'est pas défini.

2. Attention : même si les types `t_point` et `t_vector` sont construits de manière similaire, ce sont deux types différents. Il n'est donc pas possible de comparer directement une valeur de type `t_point` et une valeur de type `t_vector`.

3. Informellement, si l'on considère une valeur d'un type structuré comme un n-uplet en oubliant les noms des champs, l'ordre est le même que celui défini sur les n-uplets.

```
# 3 * 5 / 2 ;;
- : int = 7
# 3 * (5 / 2) ;;
- : int = 6
# 4.3 +. 2.0 ** 3.0 ;;
- : float = 12.3
# not(3 + 7 > 9) || ((5 - 3 ) = 1) ;;
- : bool = false
# {x = 3.0 *. 7.2 ; y = 1.0 +. 2.0 *. 3.2} ;;
- : t_point = {x = 21.6; y = 7.4}
# not(3 > 9) || (5 - 2) + 7 ;;
Error: This expression has type int but an expression was expected of
type bool
```

Dans la plupart des langages de programmation, il existe une **priorité entre les opérateurs**. Par exemple en `OCaml`, en ce qui concerne les entiers, les opérateurs de multiplication et de division ont la même priorité : les opérateurs d'addition et de soustraction ont la même priorité, qui est inférieure à celle des opérateurs de multiplication et de division. Quand des opérateurs ont même priorité, les évaluations se font de la gauche vers la droite. Si l'on souhaite prioriser certains calculs, il faut **parenthéser**. Ce sont les règles usuelles en mathématiques. Par exemple :

- l'expression `3 + 5 * 2` vaut `13`, car la multiplication est prioritaire sur l'addition : cela signifie que l'expression `5 * 2` est évaluée en premier, avec comme résultat la valeur `10`, puis l'addition `3 + 10` est évaluée ;
- l'expression `(3 + 5) * 2` vaut `16`, puisque, du fait du parenthésage, l'addition est d'abord évaluée, avec comme résultat la valeur `8`, puis la multiplication `8 * 2` est calculée ;
- l'expression `3 * 5 / 2` vaut `7`, car la multiplication et la division ont mêmes priorités : l'expression est donc évaluée de la gauche vers la droite : `3 * 5` vaut `15` qui, divisé par `2`, vaut `7` (il s'agit du quotient de la division entière) ;
- l'expression `3 * (5 / 2)` vaut `6`, puisque le quotient de la division entière de `5` par `2` vaut `2` qui, multiplié par `3`, donne `6`.

**NB1.** La plupart des langages appliquent la règle de priorité des opérateurs de multiplication et de division par rapport aux opérateurs d'addition et de soustraction. Des règles diverses sont appliquées sur les ordres d'évaluation entre opérateurs de même priorité, entre opérateurs de types différents : il est donc très fortement ***recommandé de parenthéser*** les expressions, afin que l'écriture ne dépende pas d'un langage particulier, c'est-à-dire pour la

***généralité***, mais aussi pour la ***lisibilité***. Dans les trois exemples ci-après, si l'on ne connaît pas les priorités respectives des opérateurs, la structure de l'expression n'apparaît clairement, bien que de manière peu lisible, que dans le dernier exemple [1].

```
# 3 + 5 <> 4 + 3 && 7 < 8 = (9 - 8 < 3) ;;
- : bool = true
# 3 + 5 <> 4 + 3 && 7 < 8 = 9 - 8 < 3 ;;
Error: This expression has type int but an expression was expected of
type bool
# ((3 + 5) <> (4 + 3)) && ((7 < 8) = ((9 - 8) < 3)) ;;
- : bool = true
```

**NB2. En `OCaml`, toute expression a un type et une valeur**. L'interpréteur ne nous indique pas autre chose dans tous les exemples précédents !

### 1.1.5 Variables non mutables

#### 1.1.5.1 Notion de variable

Intuitivement, une variable non mutable [2] (parfois appelée aussi *constante*), est une variable au sens mathématique : c'est un **nom associé à une valeur** (et donc à un type, puisque toute valeur appartient à une type unique). Quand on écrit dans un énoncé de mathématiques : `soit x = 3`, on définit une variable appelée `x` dont la valeur est `3`. Comme la valeur `3` est entière, on sait que la variable `x` est entière.
NB. On pourrait tout aussi bien écrire : `soit` $x \in \mathbb{N}, x = 3$.

La plupart des langages de programmation offrent des mécanismes qui permettent d'associer des noms à des valeurs. En `OCaml`, on écrira par exemple :
`let x : int = 3 ;;`

Le mot-clé `let` indique que l'on déclare ce qui le suit, c'est-à-dire ici une variable non mutable `x` de type `int`, et dont la valeur est `3`. Si l'on utilise l'interpréteur :

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

1. Il s'agit d'un `et logique` appliqué à deux expressions booléennes : l'expression de gauche est une comparaison de deux valeurs entières définies par deux expressions, l'expression de droite est une comparaison de deux valeurs booléennes, chacune d'entre elles étant définie par une comparaison de valeurs entières.

2. Par la suite, s'il n'y a pas d'ambiguïté, nous utilisons parfois simplement le terme « variable ». En particulier, il n'y a aucune ambiguïté dans les parties I et II, dans lesquelles seules des variables non mutables sont utilisées.

```
# x * 2 ;;
- : int = 6
# let y : int = x + 4 ;;
val y : int = 7
# x + y * 2 ;;
- : int = 17
```

Lors de la déclaration de la variable [1] `x`, l'interpréteur répond qu'il a bien « compris » la déclaration : `val x : int = 3` signifie une déclaration d'une variable de nom `x`, de type `int` et de valeur `3`. Cette variable peut être utilisée ensuite dans n'importe quel calcul, par exemple `x * 2` ou `x + 4`. En fait, `x` et `3` sont *synonymes* ; plus généralement, l'écriture du nom d'une variable et l'écriture de sa valeur sont parfaitement équivalentes.

Une variable peut être déclarée en utilisant une expression dans laquelle interviennent d'autres variables préalablement définies : par exemple, la déclaration de `y` utilise la variable `x`.

Par la suite, toute déclaration de variable comprend son nom, son type et sa valeur, et a donc la forme suivante [2] :

```
let var_name : type_name = expression ;;
```

Voici quelques exemples de déclarations :

```
# let b : bool = not(3 + 7 > 9) || ((5 - 3) = 1) ;;
val b : bool = false
# let c : t_int_bool = (3 mod 2, 4 + 5 > 7) ;;
val c : t_int_bool = (1, true)
# let d : t_day = MONDAY ;;
val d : t_day = MONDAY
# let origin : t_point = {x = 0.0 ; y = 0.0} ;;
val origin : t_point = {x = 0.; y = 0.}
# let vect_i : t_vector = {dx = 1.0 ; dy = 0.0} ;;
val vect_i : t_vector = {dx = 1.; dy = 0.}
# let vect_j : t_vector = {dx = 0.0 ; dy = 1} ;;
Error: This expression has type int but an expression was expected of
type float
# let d : t_day = MERCREDI ;;
Error: This variant expression is expected to have type t_day
       The constructor MERCREDI does not belong to type t_day
```

1. Pour simplifier, nous désignons la variable par son nom ; mais il faut bien avoir à l'esprit que cette variable a un nom, un type et une valeur.

2. Il s'agit d'une convention d'écriture (*cf.* section 1.2 page 34), expliquée en section 1.3.1.1 page 37.

Les deux dernières déclarations sont erronées :

— le champ `dy` du type `t_vector` est de type `float`, alors que la déclaration indique une valeur de type `int` ;

— la valeur `MERCREDI` n'appartient pas au type `t_day` : l'interpréteur ne dit pas autre chose.

**NB.**

— pour les *types produits* : si l'on veut **accéder à chaque composant d'une valeur de type produit**, il est possible de déclarer des variables pour ce faire, de la manière suivante :

```
# let (v_int, v_bool) : t_int_bool = (3 mod 2, 4 + 5 > 7) ;;
val v_int : int = 1
val v_bool : bool = true
```

Il n'y a donc pas ici la déclaration d'une variable de type `t_int_bool`, mais la déclaration de deux variables de types respectifs `int` et `bool`. Ce type de déclaration est possible pour tout type produit, et sera souvent utilisé dans les chapitres suivants.

— pour les *types structurés* : il est possible **d'accéder à chaque champ d'une variable de type structuré via son nom**, de la manière suivante :

```
# origin.x ;;
- : float = 0.
# origin.y ;;
- : float = 0.
# let p1 : t_point =
      {x = origin.x +. vect_i.dx ; y = origin.y +. vect_i.dy} ;;
val p1 : t_point = {x = 1.; y = 0.}
```

Plus généralement, l'accès au champ `name_i` d'une variable `var_name` de type `{name_1 : type_name_1 ; ...; name_n : type_name_n}` s'écrit `var_name.name_i`

#### 1.1.5.2 Variables globales *vs* variables locales

Une variable, telle que déclarée précédemment, est une **variable globale** : cela signifie qu'on peut **l'utiliser dans toutes les expressions et déclarations qui suivent sa déclaration**.

Une **variable locale** est une variable qui **ne peut être utilisée que dans l'expression qui suit sa déclaration**. Une telle variable est déclarée ainsi :

```
let var_name : type_name = expression_1 in
      expression_2
```

La construction `let ··· in` est caractéristique d'une déclaration locale. La variable `var_name`, de type `type_name` et dont la valeur est définie par `expression_1`, **n'est connue et utilisable que** dans `expression_2`, comme le montre l'exemple ci-après :

```
# let my_x : int = 7 mod 4 in
    my_x * 2
;;
- : int = 6
# my_x ;;
Error: Unbound value my_x
```

La variable `my_x` a pour valeur `7 mod 4 = 3`, la valeur de l'expression définie est donc `3 * 2 = 6`, c'est ce que répond l'interpréteur. Mais quand on demande la valeur de la variable `my_x`, la réponse de l'interpréteur indique que celle-ci n'est pas définie. En effet, la réponse `Unbound value my_x` signifie que le nom `my_x` est inconnu. Effectivement, la variable n'est définie que dans l'expression précédente : en fait, la réponse de l'interpréteur `- : int = 6` indique qu'aucune variable n'est définie, seule la valeur `6` est calculée.

Il est possible de définir plusieurs variables locales simultanément. La syntaxe est la suivante :

```
let var_name_1 : type_name_1 = expression_1
       and var_name_2 : type_name_2 = expression_2
       and ...
in
    expression
```

Les variables locales `var_name_1`, `var_name_2`, etc, ne peuvent pas être utilisées dans les expressions `expression_1`, `expression_2`, etc, définissant les autres variables locales. En revanche, ces variables peuvent bien sûr être utilisées dans l'expression `expression` :

```
# let x : int = 7 mod 4 and y : int = 9 / 4 in
   x < y
;;
- : bool = false
```

Cette expression peut d'ailleurs elle-même commencer par la déclaration d'autres variables locales, ce qui permet d'utiliser des variables locales pour la

définition d'autres variables locales. Par exemple :

```
# let x : int = 2 and y : int = 3 in
    let z : int = x + 1 in
       y * z
;;
- : int = 9
```

Les variables locales `x` et `y` sont connues lors la définition de la variable locale `z` ; le résultat de l'expression `y * z` est donc égal à `3 * (x + 1)`, c'est-à-dire à `3 * (2 + 1)`, donc `9`. Il est donc possible de définir des variables locales à la suite les unes des autres [1]. Dans ce cas (et comme d'habitude), toute valeur définie antérieurement (comme `x`, dans l'exemple) peut être utilisée par la suite (pour la définition de la variable `y`, dans l'exemple).

**NB.** Par la suite, ***seules les variables locales sont utilisées*** dans les exemples de code. Sauf exception, les ***variables globales ne sont pas utilisées*** pour le code ; elles ne le sont que pour des tests avec l'interpréteur.

## 1.2 Conventions d'écriture

Comme dit précédemment, seule une partie des possibilités offertes par le langage `OCaml` est utilisée ici. Pour cette raison, et aussi pour des raisons de lisibilité [2], des conventions d'écriture sont strictement respectées ici.

Plus généralement, l'utilisation de conventions d'écriture est courante dans la plupart des entreprises et des communautés de développeurs, afin de simplifier la compréhension du code, voire d'éviter certaines erreurs. En effet, la conception, le développement et la maintenance de codes ont un coût : les conventions d'écriture font partie des outils permettant de réduire ce coût.

---

1. On peut remarquer que si deux variables sont indépendantes, il est équivalent en pratique de les déclarer simultanément ou de les déclarer à la suite : par exemple, les trois déclarations ci-après sont équivalentes en pratique :

```
let x : int = 7 mod 4
  and y : int = 9 / 4
in
  x < y
;;
```

```
let x : int = 7 mod 4 in
let y : int = 9 / 4 in
   x < y
;;
```

```
let y : int = 9 / 4 in
let x : int = 7 mod 4 in
   x < y
;;
```

2. Il est plus facile de comprendre des codes, quand ceux-ci sont toujours écrits de la même manière.

### 1.2.1 Espaces

1. Les opérateurs intervenant dans une expression sont isolés par des espaces (un espace avant et un espace après l'opérateur) ; il en est de même pour les deux points séparant l'identificateur du type dans une déclaration de variable.
2. Le signe - intervenant dans l'écriture d'une valeur numérique négative (de type `int` ou `float`) est suivi directement par les chiffres (le signe n'est pas séparé des chiffres par un espace).
3. De même, les deux points-virgules terminant une phrase `OCaml` sont précédés d'un espace.
4. Un séparateur, par exemple une virgule dans une valeur de type produit, est suivi d'un espace.
5. Bien sûr, deux noms consécutifs sont séparés par un espace.

La phrase `OCaml` ci-après illustre ces règles :

```
# let quad_value : int * string * float * bool =
        (3, "svp", -5.3, -2 < 4) ;;
val quad_value : int * string * float * bool = (3, "svp", -5.3, true)
```

### 1.2.2 Identificateurs

1. Un mot est une suite composée de lettres (minuscules ou majuscules), chiffres, _ (caractère *souligné*, appelé *underscore* en anglais).
2. Attention : `OCaml` est sensible à la casse : par exemple, `width` et `widtH` sont des mots différents [1].
3. Dans la mesure du possible, les mots sont choisis de manière à expliciter ce qu'ils désignent ; ils sont écrits en anglais. S'ils se composent de plusieurs mots, ceux-ci sont généralement séparés par le caractère *underscore*.
4. Les noms de types et de variables (appelés aussi des *identificateurs*) sont des mots écrits en minuscules ; on peut éventuellement utiliser aussi des chiffres et le caractère *underscore*. Un nom de variable commence forcément par une lettre minuscule. Un nom de type commence par `t_`, par exemple `t_point`, `t_int_bool`, etc.

1. On peut noter que si un nom n'est pas défini, l'interpréteur propose des noms définis proches, s'il en existe :

```
# let width = 3 ;;
val width : int = 3
# widtH ;;
Error: Unbound value widtH
Hint: Did you mean width?
```

Par exemple, pour une application graphique 2D dans laquelle on manipule des rectangles dont les côtés sont parallèles aux axes du repère 2D, on choisit de représenter un rectangle par son point en bas à gauche, sa longueur et sa hauteur. Voici quelques déclarations possibles :

```
# type t_point = {x : int ; y : int} ;;
type t_point = { x : int; y : int; }
# type t_rectangle = {base : t_point ; length : int ; height : int} ;;
type t_rectangle = { base : t_point; length : int; height : int; }
# let my_rectangle_1 : t_rectangle =
      {base = {x = 0 ; y = 0} ; length = 5 ; height = 10} ;;
val my_rectangle_1 : t_rectangle =
       {base = {x = 0; y = 0}; length = 5; height = 10}
# let my_rectangle_2 : t_rectangle =
      {base = {x = 5 ; y = 10} ; length = 3 ; height = 2} ;;
val my_rectangle_2 : t_rectangle =
       {base = {x = 5; y = 10}; length = 3; height = 2}
```

**NB :** Les lettres `l` et `O` peuvent facilement se confondre avec les chiffres `1` et `0` : il faut donc être particulièrement attentif quand elles sont utilisées.

### 1.2.3 Valeurs

1. Une valeur de type `float` comprend toujours un chiffre avant le point, et un chiffre après le point.
2. Une valeur d'un type énuméré est écrite avec des majuscules ; on peut éventuellement utiliser des chiffres et le caractère *underscore*, mais le premier caractère de la valeur est forcément une lettre majuscule (comme par exemple `MONDAY` du type `t_day`).
3. Une valeur d'un type produit est toujours délimitée par une parenthèse ouvrante et une parenthèse fermante [1], comme par exemple `(1, ('x', "hello"), 2.3)` du type `int * (char * string) * float`.

### 1.2.4 Variables

1. Une variable est toujours déclarée avec son type et sa valeur.
2. Seules des variables locales sont utilisées dans un code. Les variables globales sont utilisées lors de tests effectués avec l'interpréteur.
3. Nous appliquons le principe **un nom - une notion**. En particulier, les noms sont choisis de manière à respecter les contraintes suivantes : deux

1. Il faut noter qu'en `OCaml`, `(1) = 1` et `(('x', "hello")) = ('x', "hello")`.

types ont des noms différents ; deux variables ont des noms différents ; un type et une variable ont des noms différents.

### 1.2.5 Indentations

Des exemples d'expressions structurées, comportant différents niveaux de déclarations, ont été présentés précédemment. Les indentations sont utilisées pour mettre en évidence cette structuration, de manière à ce que ces expressions puissent être rapidement analysées quand elles sont lues. Toujours pour des raisons de lisibilité, quand une expression est trop longue, elle est coupée sur plusieurs lignes, qui sont indentées. Dans le cas d'une déclaration utilisant `let` $\cdots$ `in`, si celle-ci est coupée sur plusieurs lignes, les mots `let` et `in` apparaissent en début de ligne afin de délimiter la déclaration.

Voici quelques exemples :

```
# let x : int = 3 * 2
    and y : int = 25 mod 6
    and z : int = (23 * 2) mod 7
  in
    x + y + z
;;
- : int = 11
# let x : int =
     let y : int = 3 * 4 + 2 and z : int = 3 in
     let c : int = y + z in
       c * 4
;;
val x : int = 68
```

## 1.3 Compléments

### 1.3.1 Remarques générales

#### 1.3.1.1 Conventions d'écriture

Les conventions d'écriture décrites dans la section précédente ne sont pas des règles liées à `OCaml`, ni à un autre langage de programmation. Par exemple, pour `OCaml`, les seules parties de conventions qui sont des règles du langage sont les suivantes : une valeur de type énuméré commence par une lettre majuscule, un identificateur (un nom de type ou de variable) commence par une lettre minuscule.

Un autre exemple est le suivant. Étant donnée une valeur, `OCaml` en *infère* le type ; il est donc inutile a priori de déclarer explicitement le type d'une variable, puisque `OCaml` est capable de l'identifier à partir de la valeur de la variable. Dans l'exemple ci-après, le type de la valeur `3` est reconnu, et donc

ainsi le type de la variable `x` :

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

Ceci dit, ce n'est pas le cas de nombreux langages de programmation. La convention d'écriture imposant de définir le type et la valeur d'une variable est donc définie afin de :

— apprendre à *contrôler* explicitement tous les aspects d'une déclaration ; en particulier, cela permet d'accroître la lisibilité du code, voire de détecter certaines erreurs,
— permettre de traduire relativement directement tout le code écrit ici en `OCaml` dans d'autres langages de programmation.

Ceci peut être utile lors de l'utilisation d'autres langages de programmation.

Il est tout à fait possible en `OCaml`, comme dans d'autres langages de programmation, d'utiliser des noms déjà définis par ailleurs, par exemple :

```
# let int : float = 3.0 ;;
val int : float = 3.
# let x : int = 2 ;;
val x : int = 2
# int ;;
- : float = 3.
```

Utiliser le même nom pour un type et une variable peut être source de confusion, et donc provoquer des erreurs ; de plus, cela ne facilite pas la lisibilité ! De même, `OCaml` permet de réutiliser des noms de variables pour la définition de nouvelles variables, par exemple :

```
# let x : int = 3 ;;
val x : int = 3
# let x : string = "strange" ;;
val x : string = "strange"
# x ;;
- : string = "strange"
```

Cela signifie que le nom `x`, après avoir été associé à la valeur `3`, désigne désormais une chaîne de caractères de valeur `"strange"` : la première variable

de nom `x` n'existe plus [1]. Bien sûr, cela peut aussi entraîner des confusions et des erreurs, voire a minima poser des problèmes de lisibilité [2] ; c'est pourquoi le principe **un nom - une valeur** est appliqué.

Chacune des conventions définies pourrait être expliquée, mais ceci ne présente pas beaucoup d'utilité pour l'instant. Il suffit de se souvenir que les objectifs de ces conventions sont la lisibilité et la généralité, l'objectif ici étant d'étudier les mécanismes communs des outils de base en Algorithmique - Programmation.

#### 1.3.1.2 Types

Un type est défini ici comme un ensemble de valeurs. C'est amplement suffisant pour les objectifs suivis ici, mais en fait, le terme *type* est décliné en de nombreuses appellations, recouvrant des notions plus ou moins proches, et plus ou moins liées : type abstrait, type concret, structure de données, etc.

Les types de base présentés ici se retrouvent dans de nombreux langages de programmation, avec éventuellement différentes variantes, principalement en ce qui concerne les types numériques (entiers courts ou longs, flottants en virgule fixe ou non, etc). Du fait de ces variantes, il est important de bien

1. On peut imaginer que l'on dispose d'une unique étiquette sur laquelle figure le nom `x` : si cette étiquette est posée sur la valeur `3`, elle est synonyme de cette valeur. Mais si elle est posée sur la valeur `"strange"`, elle n'est bien sûr plus posée sur la valeur `3`, dont elle n'est plus synonyme.

2. L'exemple ci-après illustre un cas d'erreur lié à l'utilisation d'un même nom pour désigner deux types différents.

```
# type t_thing = {a : int ; b : float} ;;
type t_thing = { a : int; b : float; }
# let c : t_thing = {a = 1 ; b = 1.0} ;;
val c : t_thing = {a = 1; b = 1.}
# type t_thing = int * int ;;
type t_thing = int * int
# c ;;
- : t_thing = {a = 1; b = 1.}
# let d : t_thing = {a = 13 ; b = 4.0} ;;
Error: This expression has type t_thing/1292
       but an expression was expected of type t_thing/1297 = int * int
```

Dans cet exemple, le type `t_thing` est défini, ainsi que la variable `c`, puis le type `t_thing` est « redéfini ». La variable `c` continue d'exister, et son type est le « premier » type `t_thing`. Mais lorsqu'on essaie de définir la variable `d` de type `t_thing` avec une valeur du « premier » type `t_thing`, l'interpréteur signale une erreur, car le nom `t_thing` désigne maintenant le « deuxième » type `t_thing`, même si l'interpréteur se souvient de la définition du « premier » type. C'est pourquoi, dans le message, les deux types sont désignés par le même nom `t_thing`, complété par deux entiers différents afin de les distinguer.

analyser les caractéristiques des types de base, car un même algorithme traduit sans précautions d'un langage dans un autre peut donner des résultats différents (*cf.* section 1.3.2 suivante). Ceci dit, ces problèmes sortent du cadre de travail fixé ici.

Quelques remarques :

1. quel que soit le type, il est possible en `OCaml` de comparer deux valeurs quelconques de ce type. Par exemple, la valeur booléenne `false` est strictement inférieure à la valeur `true` ; de même, il existe un ordre sur les valeurs de types énumérés, qui correspond à l'ordre de leur déclaration : par exemple, l'ordre sur les valeurs du type `t_day` est : `MONDAY` < `TUESDAY` < $\cdots$ < `SUNDAY`. En dehors de tests d'égalité ou de différence de valeurs, nous évitons d'utiliser ces possibilités, qui ne se retrouvent pas forcément sous cette forme dans certains langages de programmation ;

2. les types énumérés existent en tant que tels dans de nombreux langages. En `OCaml`, il s'agit d'une utilisation particulière de *types sommes* ;

3. le type *unit* se retrouve sous une forme ou une autre dans différents langages (par exemple *void* en langage `C`) : *cf.* partie III pour plus de précisions ;

4. les types produits existent dans de nombreux langages plus ou moins récents. Nous les utilisons en particulier pour faciliter l'écriture de certains algorithmes, par exemple lorsque le résultat d'une fonction est composé de plusieurs valeurs ;

5. En `OCaml`, la déclaration d'un type produit consiste simplement à associer un nom avec le type produit. Par exemple, la déclaration `type t_int_bool = int * bool` associe le nom `t_int_bool` au type `int * bool`, les deux écritures devenant alors équivalentes. L'exemple ci-après illustre cela :

```
# let v : t_int_bool = (3, true) ;;
val v : t_int_bool = (3, true)
# (3, true) ;;
- : int * bool = (3, true)
# v = (3, true) ;;
- : bool = true
```

   La variable `v`, de type `t_int_bool`, est comparable à la valeur `(3, true)`, de type `int * bool`, car `t_int_bool` est synonyme de `int * bool`. Il ne faut donc pas être surpris si l'interpréteur indique soit l'une, soit l'autre de ces deux écritures.

### 1.3.2 Types numériques

Si un type est un ensemble de valeurs, chacune d'entre elles doit pouvoir être représentée explicitement afin de la manipuler : ***cela implique qu'en pratique, un type est fini***.

Il existe des types qui sont intrinsèquement finis, comme par exemple en `OCaml` le type `char`, qui contient 256 valeurs, le type `bool` (2 valeurs), ou le type `unit` (une valeur). Ce n'est pas si évident lorsqu'on souhaite représenter des chaînes de caractères de longueur quelconque, voire des nombres entiers ou des nombres réels. En fait, si l'on souhaite par exemple manipuler des chaînes de caractères de longueurs « aussi grandes que l'on veut », il faut aussi disposer d'une machine disposant d'une « place mémoire aussi grande que l'on veut », voire « d'autant de temps que l'on veut » (par exemple pour afficher tous les caractères d'une chaîne à l'écran). C'est pourquoi, en pratique, le nombre d'éléments d'un type est fini.

Par la suite, nous ne tenons pas compte en général du fait qu'un type est fini : nous faisons souvent *comme si* les nombres de valeurs du type `int` et du type `float` de `OCaml` sont aussi grands que l'on veut. Le plus souvent, cela ne pose pas de problèmes, car les valeurs manipulées appartiennent à l'ensemble des valeurs effectivement représentables en machine. Mais ce n'est pas toujours le cas : il est donc indispensable de savoir comment ces problèmes sont gérés et d'en tenir compte si nécessaire, sinon des erreurs très graves peuvent se produire. Dans cette section, l'objectif n'est pas de faire une étude approfondie : en effet, différents mécanismes existent, qui varient suivant les langages. Nous souhaitons juste illustrer quelques aspects, et quelques *dangers*, en prenant comme exemples les types `int` et `float` de `OCaml`.

#### 1.3.2.1 Représentation des nombres entiers

Dans la vie courante, un nombre est écrit comme une suite de chiffres, et les chiffres utilisés sont les dix chiffres usuels $0, \cdots, 9$. L'interprétation de l'écriture d'un nombre correspond à la manière dont on le nomme : par exemple, 3267 est l'entier « trois mille deux cents soixante sept », c'est-à-dire la valeur $3 * 1000 + 2 * 100 + 6 * 10 + 7 * 1$. Plus généralement, un entier qui s'écrit $a_n a_{n-1} \cdots a_1 a_0$ a pour valeur $\sum_{i=0}^{n}(a_i * 10^i)$, avec $0 \leq a_i \leq 9$. Comme tout nombre s'écrit ici avec 10 chiffres, on dit qu'on écrit en *base* 10.

Ce même mécanisme peut être appliqué avec un ensemble différent de chiffres. L'information élémentaire, en informatique, est le *bit*, qui peut prendre deux valeurs. Les entiers sont donc représentés en *base* 2. On considère deux chiffres, que l'on note 0 et 1. L'entier qui s'écrit 1101001 en base 2 a donc pour valeur : $1*2^6+1*2^5+0*2^4+1*2^3*0*2^2+0*2^1+1*2^0$, c'est-à-dire la valeur

de l'entier écrit 105 en base 10. De la même manière, la valeur de l'entier écrit 11 (resp. 1, 01, 110, 111) en base 2 est la valeur de l'entier écrit 3 (resp. 1, 1, 6, 7) en base 10.

Usuellement, les informations « atomiques » sont représentées avec un nombre fixe de bits : actuellement, ce nombre est souvent égal à 32 ou 64. Supposons qu'on travaille avec des entiers représentés sur 64 bits : le nombre de valeurs entières que l'on peut ainsi représenter est $2^{64}$. Même si ce nombre est « très grand », il existe une infinité d'entiers (au sens mathématique) que l'on ne peut pas représenter. Que se passe-t-il alors si l'on calcule une valeur entière supérieure à la plus grande valeur représentable, ou une valeur inférieure à la plus petite ?

En `OCaml`, sur 64 bits, seuls 63 bits sont utilisés pour représenter une valeur de type `int`. Il y a autant de valeurs strictement négatives que de valeurs positives, c'est-à-dire que l'on peut représenter tout entier [1] compris entre $-2^{62}$ et $2^{62} - 1$. Les constantes prédéfinies `min_int` et `max_int` correspondent à ces valeurs :

```
# max_int;;
- : int = 4611686018427387903
# min_int;;
- : int = -4611686018427387904
```

Ces valeurs sont *cycliques*, c'est-à-dire que si l'on additionne `1` à `max_int` (resp. on supprime `1` de `min_int`), on obtient `min_int` (resp. `max_int`) : dit autrement, la valeur qui suit la plus grande valeur possible (resp. la valeur qui précède la plus petite valeur possible) est la plus petite possible (resp. la plus grande possible).

```
# max_int + 1 ;;
- : int = -4611686018427387904
# min_int - 1;;
- : int = 4611686018427387903
```

Cette manière de traiter les entiers se retrouve dans de nombreux langages de programmation. Dans certains langages, une erreur est produite quand on calcule la valeur qui suit la plus grande valeur possible (ou la valeur qui précède la plus petite valeur possible).

Enfin, on peut remarquer que pour les valeurs du type `int`, une division par 0 produit une erreur.

```
# 2 / 0 ;;
Exception: Division_by_zero.
```

```
# 2 mod 0 ;;
Exception: Division_by_zero.
```

1. Avec 32 bits, comme seuls 31 bits sont utilisés pour représenter une valeur, les valeurs représentables sont comprises entre $-2^{30}$ et $2^{30} - 1$.

#### 1.3.2.2 Représentation des nombres réels

Comme dit précédemment, il n'est pas possible en pratique de représenter explicitement tous les nombres réels, ni même tous les nombres rationnels. Les valeurs qui sont représentées correspondent en général à des nombres rationnels écrits sur le même nombre de bits qu'une valeur représentant un entier : dans ce cas, il n'y a pas plus de valeurs représentables « réelles » que de valeurs représentables « entières ».

Quel que soit le nombre de bits utilisés pour la représentation de valeurs « réelles », certaines valeurs comprises entre des valeurs représentables ne le sont pas elles-mêmes. De même, certaines opérations appliquées à certains opérandes peuvent ne pas produire le résultat que l'on obtiendrait avec de « vrais » réels. Par exemple :

```
# 7.000000000000003 ;;
- : float = 7.00000000000000266
# 7.000000000000003 *. 2.0 ;;
- : float = 14.0000000000000053
# 4.323 ;;
- : float = 4.323
# 4.3 +. 0.023 ;;
- : float = 4.32299999999999951
# 7.000000000000003 /. 1000.0 ;;
- : float = 0.00700000000000000275
# (7.000000000000003 /. 10000.0) *. 10000.0 ;;
- : float = 7.00000000000000355
```

Il existe aussi deux constantes prédéfinies `min_float` et `max_float` : attention, elles ne se comportent pas comme `min_int` et `max_int` ; il en est de même pour une division par `0.0` (*cf.* valeurs `infinity` et `neg_infinity`) :

```
# max_float ;;
- : float = 1.79769313486231571e+308
# min_float;;
- : float = 2.22507385850720138e-308
# max_float +. 1.0 ;;
- : float = 1.79769313486231571e+308
# max_float *. 2.0 ;;
- : float = infinity
# 1.0e400;;
- : float = infinity
# -1.0e400;;
- : float = neg_infinity
```

```
# 5.25 /. 0.0 ;;
- : float = infinity
# -5.25 /. 0.0 ;;
- : float = neg_infinity
# 0.0 /. 0.0 ;;
- : float = nan
```

La valeur `nan`, qui résulte de la division de la valeur `0.0` par elle-même, peut être comprise comme `not a number`[1].

## 1.4 Exercices

Quelques recommandations :

— il est conseillé, pour chaque exercice traité sur machine, de saisir le code dans un fichier spécifique, dont le nom se termine par l'extension `.ml` ;

— pensez à sauvegarder très régulièrement le code que vous aurez saisi, pour éviter les pertes en cas de « plantage ».

— l'interpréteur mémorise tout ce que vous faites entre le moment où vous commencez à vous en servir et le moment où vous l'arrêtez. Vous pouvez avoir oublié certaines actions, mais pas l'interpréteur ! Ceci peut parfois susciter quelques erreurs qui vous paraissent incompréhensibles, mais qui ne le sont pas si l'on se souvient de tout ce qui a été fait. Dans ce cas, le plus simple est de quitter l'interpréteur et de le relancer, ce qui réinitialise la mémoire ;

— les commentaires, en `Ocaml`, sont encadrés par `(*` et `*)`, par exemple `(* ceci est un commentaire *)`. Il est conseillé d'ajouter des commentaires au code : l'idée est qu'une personne qui n'a pas écrit le code devrait pouvoir lire celui-ci sans difficultés particulières.

**NB.** Le premier exercice pratique, à faire systématiquement pour tous les chapitres, consiste à exécuter les phrases `OCaml` du cours, afin de bien comprendre ce qu'elles font et de vérifier que vous obtenez bien les mêmes résultats que ceux décrits dans le cours.

1. Cette valeur a un comportement particulier, qui se retrouve pour des valeurs similaires d'autres langages de programmation, par exemple :

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