Un livre de Wikilivres.

# OCaml



---

# Introduction

## Qu'est-ce qu'OCaml ?

OCaml est un langage de programmation, c'est-à-dire une manière de donner des instructions à l'ordinateur pour obtenir des résultats ou des effets.

Tout comme Java, C# ou Python, OCaml est un *langage de haut* niveau conçu pour écrire des applications et des bibliothèques de programmation évoluées, sans avoir à se préoccuper de questions de bas niveau, telles que la gestion de la mémoire, et conçu pour favoriser la réutilisation de code ou de composants. Tout comme ces langages, OCaml dispose de nombreuses bibliothèques spécialisées qui permettent de manipuler des interfaces graphiques, des images 3D, de services web, des sons et musiques, des objets mathématiques...

Par opposition à Java, C# ou Python, qui sont des *langages impératifs*, OCaml entre dans la catégorie des *langages fonctionnels*. Là où un programme impératif sera constitué d'une suite de modifications de la mémoire vive et de l'état de l'ordinateur, un programme fonctionnel sera constitué d'un ensemble de fonctions mathématiques, qui permettent de déduire un résultat à partir d'entrées. OCaml est aussi un langage multi-paradigme, c'est-à-dire que, lorsque la programmation fonctionnelle ne suffit pas, il reste possible d'utiliser la programmation impérative ou/et la programmation orientée-objets, c'est-à-dire à se ramener à une programmation qui ressemblera aux langages précédents.

Les concepteurs de Java ou C# affirment que ces langages sont *statiquement et fortement typé*, ce qui n'est

que partiellement vrai. Par opposition, OCaml est effectivement *statiquement fortement typé*, ce qui signifie que le langage de programmation procède automatiquement à de nombreuses vérifications et rejette les programmes considérés comme erronés ou programmés avec insuffisamment de rigueur. De plus, le typage d'OCaml se fait par inférence, ce qui signifie que l'essentiel des vérifications se fait de manière transparente, sans avoir à donner d'informations supplémentaire à OCaml, contrairement à ce qui se passe en Java ou en C# dans lequel il faut préciser le type de chaque variable, de chaque argument, chaque méthode... Là où les langages dynamiquement typés tels que Python ne contrôlent la cohérence des opérations que durant l'exécution du programme, OCaml analyse celle-ci intégralement lors de la phase de compilation, ce qui impose une rigueur supplémentaire lors de l'écriture du programme mais améliore la fiabilité et simplifie considérablement la phase de tests. De plus comme la vérification de type s'effectue à la compilation et non durant l'exécution, cette dernière n'est pas ralentie par des tests permanents de cohérence.

Par opposition à Java, C# ou Python, OCaml essaye le plus possible d'être un *langage déclaratif*, c'est-à-dire un langage dans lequel on va décrire la solution à un problème, plutôt que de construire cette solution étape par étape. Pour ce faire, OCaml est *extensible*, ce qui signifie que, lorsqu'une application a besoin d'une opération complexe et répétitive, il est possible d'ajouter cette opération sous la forme d'une nouvelle primitive du langage lui-même -- ou même d'ajouter à l'intérieur d'OCaml un langage de programmation spécialisé dans la résolution de votre problème. Ainsi, certaines bibliothèques, au lieu d'ajouter de nouvelles fonctionnalités au langage, modifient ce dernier.

Par opposition à Java, C# ou Python, OCaml essaye le plus possible d'être un *langage de haute performance*. Ainsi, un programme OCaml démarre beaucoup plus vite, s'exécutera fréquemment plus vite et consommera fréquemment 4 fois moins de mémoire vive qu'un programme Java ou C#. Un programme OCaml utilisera généralement un peu plus de mémoire vive qu'un programme Python mais s'exécutera souvent 10 fois plus vite. Dans certains cas, heureusement assez rares, ce gain de performances se fait au détriment du confort de programmation.

Ah, encore une chose : par opposition à Java ou C#, et tout comme Python, OCaml est et reste un *langage d'expérimentation*, ce qui signifie qu'il gagne régulièrement de nouvelles fonctionnalités inédites. Il existe ainsi des versions d'OCaml spécialisées dans la programmation distribuée (JoCaml, Acute), dans la manipulation d'arbres XML (OCamlDuce), dans l'écriture de compilateurs (MetaOCaml), dans l'écriture de scripts shell (Cash), etc.

Avec un peu d'expérience d'OCaml, vous constaterez qu'OCaml est un langage dans lequel les programmes écrits sont beaucoup plus concis qu'en Java ou C#, grâce à de puissantes primitives d'abstraction, grâce à l'extensibilité du langage, grâce à l'inférence de types et, tout simplement, grâce à une syntaxe plus concise. Cette concision est souvent un avantage, même si elle rend le code plus dense et donc parfois plus difficile à lire.

# Pourquoi apprendre OCaml ?

Cette questions a plusieurs réponses.

Commençons par l'aspect technique et pécuniaire : OCaml est gratuit, tous les outils dont vous pouvez avoir besoin pour programmer sont gratuits (sauf l'ordinateur), toutes les extensions d'OCaml qui vous serviront lors de vos développements sont gratuites et même ce livre est gratuit.

Et en tant que langage, que m'offre OCaml ?

Pour les mathématiciens, physiciens, statisticiens ou autres utilisateurs de programmes scientifiques, OCaml permet de développer des programmes rapides et fiables, d'une part parce que le code source du programme ressemblera à la description mathématique du problème et sera donc plus simple à vérifier et d'autre part parce que les analyses permises par OCaml éviteront de nombreuses erreurs. De plus, la programmation fonctionnelle de haut niveau simplifie la constitution des bibliothèques génériques et aisément réutilisables.

Les programmeurs Java, C#, C, C++ ou autres utilisateurs de langages ou librairies système ou orientés-objets découvriront avec OCaml une autre manière de voir le monde et d'écrire des programmes dépendant moins du mode d'exécution de la machine et plus des résultats voulus, de modifier le langage de programmation pour l'adapter au problème au fur et à mesure de la progression du projet, ou encore d'autres techniques de factorisation et de réutilisation de code. Même si vous ne deviez pas utiliser OCaml dans votre carrière, les concepts vous permettront d'analyser un grand nombre de problème sous des angles que vous n'aviez pas prévus jusqu'à présent.

Les programmeurs Python, Ruby, JavaScript découvriront un langage confortable, dans lesquels ils retrouveront un grand nombre d'idées connues, sous une forme plus rigoureuse et plus adaptée au développement de grands projets critiques. À l'inverse, les utilisateurs de langages industriels -- dont l'essentiel des concepts datent des années 70 -- découvriront des idées plus récentes en provenance du monde de la Recherche. Pour information un grand nombre de concepts considérés comme novateurs dans l'industrie sont tirés directement des langages fonctionnels : les exceptions, la vérification de débordement dans les tableaux, le ramasse-miettes, l'inférence de types, les types génériques, les mini-langages spécialisés, les énumérations sûres, les clôtures...

# Mise en bouche

Avant d'entrer dans l'enseignement, commençons par une mise en bouche.

*Note:* Tout ce livre utilise OCaml Batteries Included, une distribution de OCaml qui étend aussi bien les bibliothèques standard que le langage de programmation lui-même. Cette bibliothèque est gratuite et disponible au téléchargement, mais il est important de souligner que le langage présenté ici n'est *pas* (encore) du OCaml standard, seulement l'une de ses variantes.

## Bonjour, le monde

L'extrait suivant est un programme complet d'une ligne, qui a pour effet d'afficher le texte "Bonjour, le monde" :

```
1 print_endline ("Bonjour, le monde.")
```

On pourrait d'ailleurs simplifier légèrement le programme en supprimant les parenthèses, qui ne sont ici pas nécessaires, de manière à obtenir l'extrait suivant :

```
1 print_endline "Bonjour, le monde."
```

Dans ce qui précède, `print_endline` est une fonction, dont l'effet est d'afficher du texte et de passer à la ligne, et qui n'a pas de résultat. En Java, ce serait l'équivalent de

```
1 public class Main
2 {
3   public static final void main(String args[])
4   {
5     System.out.println("Bonjour, le monde");
6   }
7 }
```

## Pi

Pour définir une valeur, il faut utiliser le mot clé `let`, mot anglais que l'on peut traduire par le mot *soit* précédant les définitions en mathématiques.

Pour définir la valeur de pi, il suffit d'écrire : *soit pi = 3.1415926*, c'est à dire :

```
1 let pi = 3.1415926
```

Cette ligne donne le nom pi à la valeur 3.1415926. Cette valeur ne changera plus au cours de l'exécution du programme. En Java, ce serait l'équivalent d'une déclaration :

```
1 public static final float PI = 3.1415926;
```

## Identité

Les fonctions sont des valeurs comme les autres. Ainsi, pour définir la fonction identité, qui à tout x associe x lui-même, on peut écrire

```
1 let id = fun x -> x
```

ou encore

```
1 let id(x) = x
```

ou encore

```
1 let id x = x
```

Ces trois définitions sont équivalentes. La première précise explicitement que id est une fonction. La troisième pourra être comprise comme la définition d'une famille de valeurs $id(x)$ pour tout $x$ -- dans la dernière version, nous avons tout simplement supprimé les parenthèses qui étaient inutiles. Nous verrons plus tard que cette fonction va faire l'objet d'un certain nombre de vérifications à chaque fois qu'elle est appliquée. Comme précisé plus haut, ces vérifications ont entièrement lieu durant la compilation.

On utilise alors id de la manière suivante

```
1 let y = id 5;;
2 let z = id "Il fait beau";;
```

En Java, pour obtenir la même fonctionnalité, le plus simple est d'abandonner toute vérification et d'écrire une classe contenant la méthode suivante :

```
1 public class Fonctions
2 {
3   public static Object id(Object x)
4   {
5     return x;
6   }
7 }
```

On utilise alors id de la manière suivante :

```
1 int    y = (Integer)Fonctions.id(5);
2 String z = (String)Fonctions.id("Il fait beau");
```

Si l'on cherche à conserver une vérification de types, il devient nécessaire de définir une classe plus complexe :

```
1 public class Id<T>
2 {
3   public T appliquer(T x)
4   {
5     return x;
6   }
7 }
```

On utilise alors id de la manière suivante :

```
1 int    y = new Id<Integer>().appliquer(5);
2 String z = new Id<String>().appliquer("Il fait beau");
```

Profitons de cet exemple pour remarquer que, en OCaml, le mot-clé `return` n'existe pas et n'est pas nécessaire. En effet, le corps d'une fonction est une expression et toute expression est évaluée. En d'autres termes, le résultat d'une fonction est donné par la dernière expression de cette fonction.

## Delta

La fonction delta de Dirac est définie mathématiquement de l'ensemble des réels vers l'ensemble des réels par

- $delta(0) = 1$
- pour tout autre x, $delta(x) = 0$

Cette définition mathématique se traduit naturellement en OCaml par

```
1 let delta = function
2   0.0 -> 1.0
3   | _ -> 0.0
```

ou encore

```
1 let delta x = match x with
2   0.0 -> 1.0
3 | _   -> 0.0
```

ou encore

```
1 let delta x = if x=0.0 then 1.0 else 0.0
```

Ces trois extraits se lisent "delta est la fonction qui à 0,0 associe 1,0 et qui à toute autre valeur associe 0,0".

Retenez la structure des deux premiers extraits, qui reviendra fréquemment. Profitons-en pour préciser que le test d'égalité se note "=", qu'il fonctionne correctement, contrairement à celui de Java ou C# (dans lesquels il faut parfois utiliser "==" et parfois invoquer une méthode) ou Python (qui, par défaut, vérifie l'identité et non l'égalité).

## Opérateurs logiques

OCaml définit bien entendu les opérateurs logiques habituels. Nous pourrions, si nécessaire, les redéfinir à la main. Ainsi, le ou exclusif logique peut s'écrire

```
1 let ou_exclusif a b = match (a,b) with
2   (true, true) -> false
3 | (true, false) -> true
4 | (false, true) -> true
5 | (false, false) -> false
```

Ce qui peut se résumer

```
1 let ou_exclusif a b = match (a,b) with
2   (true, false) | (false, true) -> true
3 | _ -> false
```

Dans ce qui précède, `true` et `false` sont les noms donnés en OCaml aux booléens vrai et faux. L'utilisation de `match (a,b)` permet de prendre des décisions en fonction de la structure du couple (a,b).

Bien entendu, la même fonctionnalité pourrait s'écrire, de manière moins lisible, à l'aide des opérateurs logiques habituels :

```
1 let ou_exclusif a b = (a && not b) || (b && not a)
```

En Java, les premiers extraits se traduiraient

```
1 public static boolean ou_exclusif(boolean a, boolean b)
2 {
3   if ((a==true && b == false) || (a==false && b== true))
4     return true;
5   else
6     return false;
7 }
```

Le dernier extrait s'écrirait

```
1 public static boolean ou_exclusif(boolean a, boolean b)
2 {
3   return (a && !b ) || (!a && b );
4 }
```

On pourrait bien entendu, en OCaml comme en Java, écrire tout ceci à l'aide d'une comparaison entre `a` et `b`, comme suit :

En Java :

```
  1 public static boolean ou_exclusif(boolean a, boolean b)
2 {
3   return (a != b);
4 }
```

En OCaml :

```
1 let ou_exclusif a b = a <> b
```

voire, de manière plus concise,

```
1 let ou_exclusif = ( <> )
```

Remarquons que la fonction obtenue en OCaml est plus générique que la version Java, puisqu'elle permet de vérifier la différence entre n'importe quelle paire d'éléments du même type, pas uniquement des booléens.

## Factorielle

La fonction mathématique factorielle peut se définir de plusieurs manières. Directement, avec des points de suspension, on écrira $factorielle(n) = 1 * 2 * ... * n$. Ou encore sans points de suspension, par récurrence, à l'aide de

- pour tout n >= 1, $factorielle(n) = n * factorielle(n-1)$.
- pour tout autre n, $factorielle(n) = 1$.

Ces deux définitions peuvent être utilisées en OCaml.

### Définition récursive

On peut traduire directement la deuxième définition par

```
1 let rec factorielle = function
2 | n when n>=1 -> n * (factorielle (n - 1) )
3 | _           -> 1
```

Cet extrait se lit exactement comme la définition par récurrence.

Dans ce qui précède, `rec` signifie que la fonction est définie par récursivité, mécanisme qui correspond à la définition mathématique par récurrence ou par induction : les cas non-triviaux sont déterminés à partir des cas précédents, ou encore la fonction s'appelle elle-même. En OCaml, la récursivité est la forme de boucle la plus fréquente.

On pourrait aussi écrire le code suivant, toujours récursif :

```
1 let rec factorielle n = if n >= 1 then n * ( factorielle (n - 1) ) else 1
```

En Java, on recommande d'écrire :

```
public static int factorielle(int n)
```

```
1   2 {
3   int result = 1;
4   for(int i = 1; i < n; ++i)
5     result *= i;
6   return result;
7 }
```

Il est aussi possible d'écrire une version récursive en Java, même si, à cause des limites des implantations actuelles de Java, ce genre de pratique est généralement évité :

```
1 public static int factorielle(int n)
2 {
3   if(n<=0)
4     return 1;
5   else
6     return n * factorielle (n-1);
7 }
```

Comme on peut le constater, les versions OCaml sont plus concises et plus proches de la définition mathématique que la version Java recommandée. Même si cela n'est pas fondamental pour un exemple aussi simple, on peut déjà constater qu'il y a un peu moins d'occasions de commettre des erreurs simples en suivant la structure OCaml que la structure Java recommandée. Cette tendance ne fait que s'amplifier pour des exemples plus complexes.

**Définition directe**

On peut aussi définir le produit des nombres de 1 à n en opérant sur l'*énumération* des entiers de 1 à n, comme suit :

```
1 let factorielle n = reduce ( * ) ( 1 -- n )
```

Cette définition (légèrement fausse) se lit "factorielle n est le résultat qu'on obtient en appliquant la multiplication aux nombres consécutifs entre 1 et n". Pour obtenir cette définition, nous avons employé `1 -- n`, c'est-à-dire l'énumération des entiers de 1 à n, l'opération `( * )`, c'est-à-dire la multiplication, et la fonction `reduce`, qui applique l'opération choisie aux deux premiers éléments de l'énumération, puis au troisième, puis au quatrième, etc. Si l'énumération ne contient qu'un seul élément, le résultat de `reduce` est cet élément, quelle que soit l'opération appliquée.

La notion d'énumération se retrouve dans d'autres langages de programmation, y compris Java ou Python, généralement sous le nom d'*itérateur*. Il s'agit d'une manière d'accéder à tous les éléments d'un ensemble, dans un ordre donné, en ne consultant qu'une seule fois chaque élément. Cette notion est centrale en OCaml Batteries Included, puisqu'il s'agit de la manière principale de réaliser des boucles ou, plus généralement, des opérations répétitives sur une structure de données.

Pourquoi notre définition est-elle fausse ? Parce que nous n'avons pas correctement géré le cas où n<=0. Dans ces cas-là, l'énumération `1 -- n` est vide et la fonction que nous avons proposé n'est pas définie. Il convient donc de remplacer l'appel à `reduce` par un appel à une fonction légèrement plus complexe, `fold`, qui prend en plus une valeur initiale (ici 1) et est capable de traiter le cas de l'énumération vide :

```
1 let factorielle n = fold ( * ) 1 ( 2 -- n )
```

Cette définition de la factorielle renvoie 1 si `n` est inférieur ou égal à `2` et 1 * 2 * ... * n dans le cas contraire.

## Manipulation de fichiers

Un dernier exemple, qui fait cette fois appel à des bibliothèques de manipulation de fichiers. Le rôle de cet exemple est de lire et d'afficher le contenu d'un certain nombre de fichiers, en ajoutant avant chaque ligne le numéro de la ligne.

```
1 open Enum, File, IO;;
2
3 let add_line_number num line = (string_of_int num)^": "^line;;
4
5 iter f (args ())
6 where f arg = with_file_in arg  (fun input ->
7    write_lines stdout (mapi add_line_number (lines_of input))
8 );;
```

La première ligne permet de faire appel à des fonctions prédéfinies et rangées dans les *module* `Enum` (tout ce qui a trait aux énumération), `File` (tout ce qui a trait aux fichiers) et `IO` (des fonctions qui permettent de gérer la lecture ou l'écriture d'informations).

L'extrait définit ensuite une fonction `add_line_number` qui, à partir d'un nombre et d'une chaîne de caractères, construit une nouvelle chaîne de caractères qui commence par le nombre, puis continue par un caractère ":" et enfin le contenu de la chaîne initiale. Nous utiliserons cette fonction pour ajouter le numéro de la ligne avant le contenu de la ligne.

La ligne suivante fait appel à la fonction `iter`. Cette fonction, proche dans l'esprit de `reduce`, permet de répéter un traitement et de l'appliquer à chacun des éléments d'une énumération. Ici, le traitement est donné sous la forme de la fonction `f`, définie un peu plus loin, et l'énumération est `args ()`, c'est-à-dire les arguments passés au programme par la ligne de commande -- si vous ignorez ce qu'est la ligne de commande, ce n'est pas important, sachez juste ici qu'il va s'agir de plusieurs noms de fichiers donnés au lancement du programme.

L'instruction `where` introduit une définition après son utilisation, ici la définition de la fonction `f`. Cette fonction prend un argument `arg`, le nom du fichier à lire, puis invoque la fonction `with_file_in` avec cet argument `arg` et avec une autre fonction `fun input -> ...`. La fonction `with_file_in`, définie dans le module `File`, a pour rôle d'ouvrir un fichier (ici le fichier nommé `arg`), de passer le contenu de ce fichier à une autre fonction (ici, la fonction `fun input -> ...` ) et, une fois que cette dernière fonction a fini de s'exécuter, de fermer le fichier.

Que fait donc cette fameuse fonction `fun input -> ...` ? Partant d'un contenu de fichier `input`, elle décompose ce contenu en une énumération de lignes à l'aide de `lines_of`, applique à chaque ligne consécutive la fonction `add_line_number` à l'aide de la fonction `mapi`, puis écrit la nouvelle énumération ainsi obtenue dans le fichier `stdout`, à raison d'une ligne à la fois, à l'aide de la fonction `write_lines`. Il aurait tout aussi bien été possible de décomposer en caractères à l'aide de `chars_of`, ou en nombre, ou en constructions plus complexes -- et de réécrire de même à l'aide de `write_chars`, etc. Quant à `stdout`, il s'agit de la *sortie standard*, c'est-à-dire non pas un fichier réel mais l'écran. En Java, ce serait l'équivalent de `System.out`.

La fonction `with_file_in` constitue l'une des manières de gérer ce qui, en Java ou en Python, serait traité à l'aide d'un bloc de finalisation. En OCaml, la finalisation n'est pas (encore) une opération standard du langage mais peut être remplacée sans difficultés.

En Java, pour obtenir un résultat similaire, on aurait ajouté au programme quelque chose comme

```java
import java.io.*;
//...

LineNumberReader reader = new LineNumberReader(new FileReader("/tmp/in.txt"));
try
{
  try
  {
    PrintWriter writer = new PrintWriter(new FileWriter("/tmp/out.txt"));
    reader.setLineNumber(1);
    for(String line = reader.readLine(); line != null; line = reader.readLine())
      writer.print(reader.getLineNumber+": "+line)
  } finally {
    writer.close();
  }
} finally {
  reader.close();
}
//...
```

La complexité des deux programmes est comparable, même si l'on peut constater que, de nouveau, les bibliothèques de OCaml sont de plus haut niveau et donc plus simples à composer sans commettre d'erreurs. En particulier, dans la version Java, de nombreuses erreurs peuvent se glisser, qui ne seraient pas détectées par le compilateur, par exemple oublier de fermer les flux, oublier d'avancer progressivement dans un flux, tester si la ligne est vide au lieu de déterminer si elle vaut `null`, utiliser `equals` au lieu de `!=`, etc. À l'inverse, dans la version OCaml, le programmeur ne peut commettre aucune de ces erreurs.

Si nécessaire, OCaml fournit aussi les bibliothèques nécessaires pour gérer les fichiers à peu près de la même manière qu'en Java, ou pour traiter les expressions régulières.

## Et maintenant ?

Si cette introduction vous a donné envie d'en apprendre plus, il est temps de voir Les bases d'OCaml.

# Bases

### Objectifs du chapitre

Ce chapitre vous enseignera

- comment installer OCaml
- comment lancer et manipuler OCaml en ligne de commande
- comment lire, définir et manipuler des valeurs et des fonctions simples et d'ordre supérieur.

À l'issue de ce chapitre, vous pourrez utiliser OCaml comme une super-calculatrice programmable.

### Matériel nécessaire pour ce chapitre

Pour ce qui suit, vous allez avoir besoin d'une installation d'Objective Caml 3.11 ou ultérieur avec Batteries Included et peut-être de quelques outils. Objective Caml est disponible sur de nombreuses plate-formes.

*Note* Il est fort probable que tout ce qui suit fonctionne aussi avec une version plus ancienne de OCaml. Ceci

n'a pas été testé.

### Power Users (Linux, MacOS X, BSD, Cygwin)

Si la notion de dépendances et l'idée d'installer des logiciels en ligne de commande ne vous pose pas de problèmes, la méthode recommandée est d'utiliser la distribution GODI d'OCaml, disponible sous toutes les plate-formes Unix, Windows y compris (par le biais de la couche Unix Cygwin).

#### Windows uniquement

- Commencez par installer Cygwin (http://www.cygwin.com/) [archive]. Si vous avez des problèmes au cours de l'installation de GODI, vous devrez revenir à ce point et installer les paquets Cygwin manquants.

#### Toutes les plate-formes

Tout ce qui suit doit se dérouler sous votre compte utilisateur courant.

- Téléchargez la dernière version de l'installateur GODI (http://download.camlcity.org/download/godi-rocketboost-20110811.tar.gz) [archive].
- Décompressez l'archive que vous venez de récupérer.
- Prendre son éditeur favori, et éditer le fichier .bash_profile dans $HOME, par exemple en faisant

```
vim ~/.bash_profile
```

afin d'ajouter les lignes suivantes:

```
    if [ -d ${HOME}/godi/bin -a -d ${HOME}/godi/sbin ] ; then
        PATH=${HOME}/godi/bin:${HOME}/godi/sbin:${PATH}
    fi
    if [ -d ${HOME}/godi/man ] ; then
        MANPATH=${HOME}/godi/man:${MANPATH}
    fi
    # Spécifique Windows + Cygwin
    TTY=`tty`
```

(<PREFIX> est ici ${HOME}/godi)

N.B.: Étendre la variable $PATH en rajoutant les chemins des binaires produit par l'installation de GODI est une bonne idée, peu importe le shell et l'OS ;-)

- En ligne de commande, rendez-vous dans le répertoire créé par la décompression et lancez l'installateur en tapant

```
./bootstrap --prefix <PREFIX> --section=3.12
```

(remplacez `<PREFIX>` par le chemin complet vers le répertoire dans lequel vous souhaitez installer GODI).

- Modifiez votre chemin de recherche pour permettre de trouver GODI. Généralement, vous emploierez pour ce faire

les commandes suivantes :

```
PATH=$PATH:<PREFIX>/bin:<PREFIX>/sbin; export PATH
```

- Lancez l'installation en tapant, toujours dans le même répertoire

```
./bootstrap_stage2
```

Une fois l'installation de GODI terminée, vous pourrez ajouter des bibliothèques OCaml en tapant

```
godi_console perform -build godi-batteries
```

pour installer Batteries Included par exemple.

#### Spécifique à Cygwin

Sous Windows™, la commande permettant l'initialisation du bootstrap installe le portage pour Cygwin (vos applications auront besoin de cygwin.dll pour fonctionner). Il faudra néanmoins faire un

```
  export PATH=/usr/local/bin:/usr/bin
```

avant cette commande pour ne pas avoir d'espaces dans les chemins de la variable $PATH.

Pour avoir une installation qui fonctionne sans Cygwin, il faut spécifier un autre portage sur la ligne de commande.

#### Spécifique à MinGW32

Pour MinGW32 (à installer par Cygwin), installer d'abord gcc et ensuite gcc-mingw en version 3. Et ensuite remplacer la commande d'initialisation du bootstrap par:

```
  export PATH=/usr/local/bin:/usr/bin
./bootstrap --prefix <PREFIX> --section=3.12 --w32port=mingw
```

**Attention**: le portage pour MinGW32 permet générer des binaires natifs pour Windows™ plus rapide mais au prix d'avoir quelques appels systèmes en moins et d'avoir un certain nombre de soucis avec les chemins.

### Alternative, sous Unix

Des paquets pour OCaml sont disponibles pour :

- Ubuntu
- Debian
- Fedora
- Mageia
- Archlinux
- Gentoo
- ...

Vous pouvez les trouver à l'aide des mécanismes habituels de gestion de paquets de votre distribution.

En plus de ce qui précède, il est recommandé d'installer soit le logiciel `ledit` (disponible sous GODI), soit le logiciel `rlwrap` (disponible sous votre gestionnaire de paquets), qui vous simplifieront le travail lors de l'utilisation d'OCaml en ligne de commande.

**Alternative, sous Windows**

La distribution minimale pour Windows, avec un installateur graphique, est disponible sur le site officiel (http://caml.inria.fr/ocaml/release.fr.html) [archive]. Insistons sur le fait que cette distribution est minimale et donc peu utilisable.

# Le mode calculatrice d'OCaml

OCaml dispose de deux modes principaux d'utilisation :

- le compilateur, que nous verrons au prochain chapitre
- la ligne de commande, ou "mode calculatrice".

C'est ce dernier que nous allons utiliser le temps d'un chapitre. Ce mode calculatrice, bien que limité par rapport à un compilateur et beaucoup plus lent que le compilateur-optimisateur d'OCaml, est très pratique pour tester des extraits de code ou lancer des calculs.

Pour lancer ce mode calculatrice, la manière la plus simple minimale est d'ouvrir un terminal puis de taper

```
ocaml
```

mais l'édition et les fonctions de bases sont alors limitées, il est donc recommandé d'utiliser plutôt

```
rlwrap ocamlfind batteries/ocaml
```

ou, si vous avez installé ledit au lieu de rlwrap,

```
ledit ocamlfind batteries/ocaml
```

Cette instruction demande à votre système d'exploitation de trouver et d'exécuter la version de OCaml Batteries Included installée sur votre ordinateur. Sur cette ligne, `ocamlfind` est le gestionnaire de versions de OCaml et des bibliothèques OCaml, qui permet de manipuler diverses versions de OCaml, `batteries/ocaml` est le nom du mode calculatrice d'OCaml avec Batteries Included et `rlwrap`/`ledit` est un petit outil qui simplifie l'édition de texte.

# Premiers pas avec OCaml

À l'ouverture en mode calculatrice OCaml affiche le message suivant :

```
        Objective Caml version 3.12.0


------------------------------------------------
|                                              |
|     This is OCaml, Batteries Included.       |
```

```
|                                            |
|                                            |
|      If you need help, type '#help;;'      |
|                                            |
--------------------------------------------


#
```

La première ligne confirme que OCaml 3.12 a bien été chargé. La suite confirme que vous utilisez bien OCaml Batteries Included et non pas une distribution plus minimaliste de OCaml. Enfin, le symbole `#` est l'invite de commande, qui signifie que la ligne de commande est prête et attend vos instructions. Au cours de ce chapitre, nous ferons figurer ce symbole au début de chaque ligne tapée par l'utilisateur. Insistons sur le fait que c'est à OCaml d'afficher ce symbole et non pas à vous de le taper.

> **Note** Si jamais vous avez besoin d'interrompre un programme en cours d'exécution dans la calculatrice, utilisez la combinaison de touches `Ctrl+C`. Si vous avez besoin de quitter la calculatrice, écrivez `#quit;;`.

La ligne de commande accepte

- des calculs (ou 'évaluations')
- des définitions de valeurs
- des directives spéciales, qui permettent notamment de quitter ou d'ouvrir l'aide.

Pour commencer, intéressons-nous aux évaluations.

## Calculer avec OCaml

Pour évaluer 0 + 1, écrivons

```
# 0 + 1 ;;
```

Le symbole `;;` constitue la "fin de phrase". Dans la calculatrice, chaque évaluation doit être suivie d'une fin de phrase.

> **Convention** Les conventions syntaxiques de OCaml veulent qu'on insère une espace avant l'opérateur et une espace après l'opérateur. Ces conventions évitent certaines erreurs de relecture de programmes lors de l'utilisation d'opérateurs personnalisés. On insèrera de même une espace avant et après chaque parenthèse.

L'expression arithmétique précédente produira la réponse

```
- : int = 1
```

Cette réponse se lit "le résultat du dernier calcul est un entier, sa valeur est 1". Nous reviendrons plus tard sur la définition des entiers en OCaml.

OCaml dispose des opérations arithmétiques habituelles sur les entiers : addition (`+`), soustraction (`-`), multiplication (`*`), division entière (`/`), reste de la division entière (`mod`), négation (`~-`), avec les priorités mathématiques habituelles. Tous ces opérateurs sont infixes. Ainsi, on écrira

```
# 2 mod 3 ;;
- : int = 2
```

Notons que toutes ces opérations acceptent uniquement des entiers et produisent uniquement des entiers. Ainsi, on obtiendra

```
# 2 / 3 ;;
- : int = 0
```

De même, OCaml dispose des opérations arithmétiques habituelles sur les nombres à virgule : addition (`+.`), soustraction (`-.`), multiplication (`*.`), division (`/.`), puissance (`**`). Ces opérations, de nouveau, sont infixes et respectent les priorités mathématiques habituelles. On écrira donc

```
# 2. ** 3. ;;
- : float = 8
```

De nouveau, toutes ces opérations acceptent uniquement des nombres à virgule flottante et renvoient uniquement des nombres à virgule flottante. Si l'on tente de combiner entiers et nombres à virgule flottante, on obtiendra

```
# 2.0 / 3 ;;
  ^^^
This expression has type float but is here used with type int

# 2 /. 3.0 ;;
  ^
This expression has type int but is here used with type float
```

> **Note** En cas d'erreur de syntaxe ou la compilation, OCaml souligne la zone sur laquelle porte l'erreur à l'aide soit de symboles ^^^, soit de traits, selon l'environnement utilisé, voire la surligne en couleur. Il arrive malheureusement que les symboles soient décalé de quelques caractères.

Pour mélanger entiers et nombres à virgule, on emploie les fonctions de conversion `int_of_float` et `float_of_int`.

```
# float_of_int ( 2 ) /. 3.;;
- : float = 0.666666666667

# 2 / int_of_float( 1.5 );;
- : int = 2
```

Quelques autres fonctions complètent l'attirail :

- des fonctions sur les entiers telles que la valeur absolue (`abs`), successeur (`succ`), prédécesseur (`pred`), etc.
- des fonctions sur les nombres flottants, telles que les fonctions trigonométriques, la racine carrée (`sqrt`), l'exponentielle (`exp`), les fonctions hyperboliques (`tanh`, `cosh` et `sinh`), etc.

Pour consulter la documentation sur chacune de ces fonctions, utilisez la directive `#man`, comme suit :

```
#man "cosh";;
```

## Limitations et précautions

### Limites de calcul

Comme dans la majorité des langages de programmation, les entiers sont bornés par la capacité de calcul du microprocesseur. Avec OCaml, sur un ordinateur à processeur 32 bits, les limites sont de

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

c'est-à-dire $[-2^{30}, 2^{30}[$. Sur un ordinateur à processeur 64 bits, les limites sont de

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

c'est-à-dire $[-2^{62}, 2^{62}[$.

> **Note pour lecteurs avancés** : à la lecture, un programmeur habitué aux limites des systèmes s'interrogera sur le destin du dernier bit. Celui-ci sert à différencier les 'entiers' et les 'références' lors de la récupération de la mémoire vive. Cette astuce limite la gamme des entiers utilisables sur le système mais évite
>
> - d'une part l'imprécision et la lenteur des ramasse-miettes sans support du langage, tels que le très utilisé ramasse-miette de Boehm ou certaines versions de Java, qui sont contraints de considérer les nombres entiers comme des pointeurs ;
> - d'autre part le gaspillage de mémoire vive et la lenteur d'autres ramasse-miettes, dont certaines versions de Java, qui sont contraints d'ajouter un mot de mémoire après chaque valeur pour différencier entiers et références.

Pour les applications ayant besoin d'une gamme d'entiers plus grande que ce que OCaml propose par défaut, plusieurs alternatives existent. Notamment, OCaml est fourni avec une bibliothèque de calcul sur 64 bits et une bibliothèque de calculs sur entiers de taille illimitée. D'autres bibliothèques sont disponibles, pour faire face à d'autres circonstances.

### Dépassements de capacité

Par défaut, les opérations arithmétiques sur les entiers ne préviennent pas en cas de dépassement. Ainsi, on aura

```
# min_int - 1 ;;  (* Un nombre qui devrait être négatif *)
- : int = 1073741823
```

Il est possible de modifier la définition des opérations arithmétiques de manière à prévenir (lancer une exception) en cas de dépassement mais la solution est lente et reste imparfaite. Nous verrons ultérieurement

comment demander à OCaml Batteries Included d'utiliser cette version des opérations.

#### Conversion automatique

Un entier n'est pas un nombre flottant et un nombre flottant n'est pas un entier. Il a été plusieurs fois question de permettre en OCaml la conversion automatique d'un entier en un flottant ou d'un flottant en entier, comme dans certains autres langages de programmation, mais les développeurs d'OCaml ont rejeté l'idée, pour plusieurs raisons :

- la conversion automatique est une source invisible d'erreurs et de ralentissements pour les logiciels d'analyse numérique, qui constituent l'une des raisons principales d'être d'OCaml
- la philosophie des concepteurs d'OCaml est de ne pas résoudre de problèmes à moitié, or, à partir du moment où deux types peuvent être automatiquement convertis l'un en l'autre, le monde de la Recherche ne connaît pas encore de méthode automatique et exacte pour permettre systématiquement de déterminer durant la compilation le type d'une expression -- introduire la conversion automatique interdirait donc l'analyse automatique de types, qui est primordiale en OCaml.

### À retenir

- Le type des nombres entiers est `int`.
- Le type des nombres à virgule est `float`.
- Un entier n'est pas un nombre à virgule, un nombre à virgule n'est pas un entier. Pour convertir l'un en l'autre, il convient d'employer les fonctions `float_of_int` et `int_of_float`.
- Le message

```
This expression has type xxxxx but is here used with type yyyyy
```

signale une erreur de typage : l'analyse de type a déterminé que l'expression soulignée porte le type `xxxxx`, alors que le contexte dans lequel elle est utilisée est prévu uniquement pour des valeurs de type `yyyyy`.

## Booléens

Deux entiers ou deux nombres à virgule peuvent être comparés à l'aide des opérateurs =, <, <= et >, >=, <>, `!=`.

```
# 1 = 0;;
- : bool = false

# 1. = 0.;;
- : bool = false

# 1. = 0;;
       ^
This expression has type int but is here used with type float
```

> **Note** : l'opérateur = procède à une comparaison au sens mathématique du terme (comparaison structurelle) et non au sens de l'identité (égalité des références), contrairement à ce qui se produit dans l'essentiel des langages de programmation, et ne dépend pas de l'éventuelle `null`-ité d'un pointeur. En d'autres termes, si vous avez besoin de vérifier l'égalité de deux valeurs, et à moins d'avoir une très bonne raison de changer la définition de l'égalité, vous

pouvez utiliser sans crainte l'opérateur `=`.

Le résultat d'une comparaison est une valeur `true` ou `false`, de type `bool`.

Deux booléens peuvent eux-même être comparés à l'aide de l'opérateur `=` -- en fait, comme nous le reverrons plus loin, les opérateurs de comparaison peuvent être appliqués à n'importe quel couple de valeurs de même type. En plus de la comparaison, les booléens sont dotés des opérateurs et fonctions habituelles : le 'et logique' paresseux (`&&`), le 'ou logique' paresseux (`||`) et la négation (`not`).

Les booléens sont employés notamment dans les opérations de contrôle de flot (presque) habituelles d'OCaml :

```
# if 1 = 0 then 2 else 3 ;;
- : int = 3
```

Nous reviendrons un peu plus tard sur ces instructions de contrôle de flot.

### À retenir

- Pour comparer deux valeurs de même type, on emploie `=`.
- Le type des booléens est `bool`.
- Les deux booléens se notent `true` et `false`.

## Déclaration de valeurs

### Valeurs simples

Déclarer une valeur consiste à donner un nom au résultat d'une expression. Ainsi, l'extrait

```
# let pi = 3.1415927 ;;
val pi : float = 3.1415927
```

'lie', pour toute la suite de l'exécution du programme, le nom `pi` à la valeur '3.1415927'. La réponse d'OCaml se lit d'ailleurs "Dorénavant, le nom pi désigne un nombre flottant dont la valeur est 3.1415927."

Ce nom peut être réutilisé dans n'importe quelle expression :

```
# cos ( pi ) ;;
- : float = -0.999948994965
```

> **Note** : insistons sur le fait que la valeur de `pi` ne peut changer. Pour être plus précis, *il est possible de masquer `pi` par une nouvelle définition mais pas de réaffecter une nouvelle valeur à `pi`*. En d'autres termes, les valeurs OCaml fonctionnent comme les constantes dans de nombreux langages. On les appelle cependant des *variables*, car il s'agit bien de variables au sens mathématique du terme.
>
> **Note** : un nom lié est toujours *lié à une valeur*. En d'autres termes, en OCaml, il n'est jamais nécessaire de vérifier si une valeur est `null`, `undefined`, etc. Dans les cas où il est nécessaire de manipuler une valeur qui peut ou non être définie, on emploiera le mécanisme des `options`, que nous verrons plus tard.

Tenter d'utiliser un nom qui n'est pas lié provoque une erreur. Ainsi, on aura

```
# cos ( pi' ) ;;
        ^^^
Unbound value pi'
```

Nous verrons plus tard que, comme les erreurs de type ou de syntaxe, les erreurs de liaison sont détectées à la compilation, comme en Java ou C# et par opposition à Python, JavaScript ou Ruby.

Bien entendu, un nom peut être donné au résultat d'une expression plus complexe que la constante 3.1415. Ainsi, pour définir `pi`, on utilisera plutôt le calcul plus précis suivant :

```
# let pi = 2. *. acos ( 0. ) ;;
val pi : float = 3.14159265359
```

> **Note** : La liaison `let x = ... ;;` calcule immédiatement le résultat de l'expression. Si nécessaire, OCaml propose aussi des *expressions paresseuses*, qui ne sont calculées que lorsque leur résultat est effectivement consulté. Nous en rediscuterons plus tard.

Plusieurs noms peuvent être définis simultanément

```
# let  x = 1
  and  y = 2;;
val x : int = 1
val y : int = 2

# let ( a, b ) = ( 1, 2 );;
val a : int = 1
val b : int = 2
```

Notons que le nom n'est lié qu'à partir de la fin de sa déclaration. En d'autres termes, à l'intérieur de la définition de `x`, `x` n'a pas de sens :

```
# let x = x + 1 ;;
          ^
Unbound identifier x
```

Il est aussi possible de lier un nom localement. Ainsi, on écrira

```
# let x = 50 in
  3 * x * x + 4 * x + 5 = 0 ;;
- : bool = false
```

ou encore

```
# 3 * x * x + 4 * x + 5 = 0
  where x = 50;;
- : bool = false
```

Dans cet extrait, le nom `x` n'est lié que le temps de calculer `3 * x * x + 4 * x + 5` et de comparer ce résultat à `0`. On dit que *la portée de x* est `3 * x * x + 4 * x + 5 = 0`. La réponse de OCaml, qui

commence par un tiret, confirme qu'aucune nouvelle variable n'a été liée globalement par cette expression. Ainsi, essayer ensuite d'utiliser `x` hors de sa portée expression provoquera une erreur

```
# x ;;
  ^
Unbound value x

# let x = x in 3 * x * x + 4 * x + 5 = 0 ;;
          ^
Unbound value x
```

Comme le montre notre exemple précédent, `let...in...` est une expression et a donc un résultat. On peut donc combiner `let...in` et `let ...` de manière à donner un nom global au résultat d'un calcul qui utilise des noms locaux :

```
# let y =
   let x = 50 in
    3 * x * x + 4 * x + 5 = 0 ;;
val y : bool = False
```

On peut de même définir des couples ou partager des valeurs entre plusieurs résultats de manière à éviter de les recalculer :

```
# let (valeur_de_pi, incertitude_entre_valeurs) =
    let ( pi_1, pi_2 ) =
      ( 2. *. acos ( 0. ), 2. *. asin ( 1. ) )
    in
      ( pi_1, abs_float ( pi_1 -. pi_2 ) );;
val valeur_de_pi : float = 3.14159265359
val incertitude_entre_valeurs : float = 0
```

Sans surprise, si une liaison globale peut contenir des liaisons locales et si une liaison locale peut elle-même contenir d'autres liaisons locales, une liaison globale a toujours lieu au plus haut niveau, jamais à l'intérieur d'une expression. En d'autres termes, un calcul ne peut pas avoir comme effet secondaire de faire apparaître une nouvelle liaison globale. Ce comportement est similaire à celui de Java ou C#, par opposition à celui de JavaScript, Python ou Ruby.

Ce mécanisme de liaison locale est utilisé notamment :

- à l'intérieur de fonctions
- pour retenir des valeurs intermédiaires mais pas intéressantes lors d'un calcul
- pour masquer des informations et des fonctions avec plus de finesse que `protected` ou `private` dans les langages objets (la technique est en fait plus proche du masquage utilisé en JavaScript).

> **Note** Un nom de variable commence toujours par une minuscule ou le caractère _. Les identificateurs commençant par des majuscules sont réservées aux constructeurs et aux noms de modules, dont nous discuterons plus tard. Tenter d'utiliser un nom commençant par une majuscule en tant que variable provoque une erreur de syntaxe parfois difficilement lisibles.

```
# let Pi = 3.14;
      ^^
Error: Unbound constructor Pi
```

Nous verrons plus tard ce que sont les constructeurs en OCaml.

#### À retenir

- L'instruction `let` sert à lier un nom globalement.
- Les instructions `let ... in ...` et `... where ...` servent à lier un nom localement.
- Les variables ne changent pas de valeur durant le temps de leur définition.
- Le nom d'une variable commence toujours par une minuscule.
- Le message

```
Unbound value xxxxx
```

signifie que le nom xxxxx n'est pas lié (ou défini) dans ce contexte.

### Fonctions

#### Syntaxe

Dans ce qui précède, nous avons déjà manipulé quelques fonctions

```
# float_of_int ( 5 ) ;;
- : float = 5

# abs ( -5 );;
- : int = 5
```

En OCaml, les fonctions sont des valeurs comme les autres, qui peuvent être déclarées à l'aide de `let`, manipulées, évaluées, passées en tant qu'argument ou qui peuvent elles-mêmes être le résultat de l'évaluation d'expressions ou d'autres fonctions. Pour définir une fonction, OCaml propose plusieurs syntaxes, entre lesquelles nous choisirons selon des critères essentiellement esthétiques :

```
# let  ajouter_un          = fun x -> x + 1
  and  ajouter_deux ( x )  = x + 2
  and  ajouter_trois x     = x + 3
  and  (ajouter_quatre x)  = x + 4;;

val ajouter_un     : int -> int = <fun>
val ajouter_deux   : int -> int = <fun>
val ajouter_trois  : int -> int = <fun>
val ajouter_quatre : int -> int = <fun>
```

Avant de nous intéresser à la réponse d'OCaml, commençons par détailler ces quatre syntaxes. Notons aussi que nous aurions pu déclarer les quatre fonctions séparément ou sous la forme d'un quadruplet mais nous avons préféré utiliser `let...and...` pour habituer le lecteur à lire et à utiliser cette construction.

> **Vocabulaire** Dans ces fonctions, on appelle `x` *argument* ou *paramètre formel* de la fonction, `x+1` *corps de la fonction* et `fun x -> x + 1` *fonction anonyme*, *abstraction* ou *λ-expression*. Tout comme une valeur liée à l'aide de `let...in`, l'argument `x` a une *portée* limitée au corps de la fonction. En fait, `x` est *lié* à l'intérieur de la fonction.

La première syntaxe fait appel au mot-clé `fun` (pour "function") et s'énonce "Le nom *ajouter_un* est lié à la fonction qui à tout x associe x + 1", soit encore, en langage plus mathématique,

$ajouter\_un : x \longmapsto x + 1$. Cette formulation met l'accent sur le fait qu'une fonction est une valeur comme une autre.

La deuxième syntaxe s'énonce "Pour tout x, la valeur de `ajouter_deux ( x )` est x + 2", soit encore, en langage mathématique, $\forall x, ajouter\_deux(x) = x + 2$." Le résultat est absolument équivalent à celui qu'on obtient avec la première syntaxe et le vocabulaire est le même, mais cette formulation met l'accent sur la présentation d'une fonction comme une famille de valeurs.

La troisième et la quatrième syntaxe sont des variantes sur la deuxième, et s'énoncent de la même manière. Nous nous sommes contentés de jouer sur le placement des parenthèses afin d'illustrer la signification de celles-ci. En effet, contrairement à la majorité des langages, dans lesquels il est obligatoire d'encadrer les arguments d'une fonction entre des parenthèse, en OCaml, ces dernières ne sont obligatoires que pour forcer les priorités et résoudre d'éventuelles ambiguïtés, tout comme les parenthèses dans une expression arithmétique ne sont obligatoires que pour forcer la priorité des opérations.

> **Note** En OCaml, dans presque tous les cas, la syntaxe idiomatique est la plus courte. On définira donc généralement une fonction sous une forme proche de `ajouter_trois`.

Cette conception des parenthèses s'applique aussi lors de l'invocation de fonctions :

```
# ajouter_un (10) * 2 ;;
- : int = 22
# ajouter_un 10 * 2 ;;
- : int = 22
# ( ajouter_un 10 ) * 2 ;;
- : int = 22
# ajouter_un (10 * 2) ;;
- : int = 21
# 2 * ajouter_un (10) ;;
- : int = 22
# 2 * ajouter_un 10 ;;
- : int = 22
# 2 * ( ajouter_un 10 );;
- : int = 22
# ( 2 * ajouter_un ) 10 ;;
      ^
This function is applied to too many arguments, maybe you forgot a `;'
```

Nous nous intéresserons à ce message d'erreur un peu plus tard. Pour le moment, contentons-nous d'en déduire qu'une fonction -- qui n'est, par définition, pas un nombre -- ne peut pas être multipliée par 2 à l'aide de l'opérateur `*`. Du reste de l'échantillon, nous pouvons déduire que *l'invocation de fonction est plus prioritaire que la multiplication* et que le parenthésage n'est nécessaire que pour forcer la priorité.

> **Note** En cas de doute sur la priorité des opérateurs et des opérations, ajoutez des parenthèses. C'est ce que font tous les programmeurs OCaml et ça évite bien des erreurs, surtout à partir du moment où vous définirez vos propres opérateurs.

Insistons sur le fait qu'une fonction est une valeur comme une autre. Ainsi, on peut parfaitement écrire une fonction dont le résultat est une autre fonction

```
# let multiplication = fun x ->
    fun y ->
      x * y ;;
let multiplication : int -> int -> int = <fun>
```

Ce résultat se lit "Le nom `multiplication` est lié à une fonction dont l'argument est un entier et dont le résultat est une fonction de l'ensemble des entiers vers l'ensemble des entiers." Cette fonction `multiplication` se manipule naturellement :

```
# let multiplication_par_5 = multiplication 5;;
val multiplication_par_5 : int -> int = <fun>

# multiplication_par_5 10 ;;
- : int = 50
```

En une seule étape, nous écririons

```
# ( multiplication 5 ) 10 ;;
- : int = 50

# multiplication 5 10 ;;
- : int = 50
```

> **Vocabulaire** Dans ce qui précède, une fonction de la forme `fun y -> x * y` est appelée *"clôture"*. Le terme désigne une fonction définie à l'intérieur d'une autre fonction et qui utilise des variables locales ou des arguments de la fonction qui la contient. On emploie parfois le terme (improprement) pour désigner n'importe quelle fonction locale.

De même, nous pouvons définir une fonction qui prend en argument une fonction et produit comme résultat une fonction :

```
# let appliquer_operation_postfixe =
   fun x ->
   fun y ->
   fun o ->
    o x y ;;
val appliquer_operation_postfixe : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c = <fun>

# appliquer_operation_postfixe 5 10 multiplication;;
- : int = 50

# appliquer_operation_postfixe 5 10 max;;
- : int = 10
```

Nous reviendrons sous peu sur la manipulation de fonctions par des fonctions et sur le sens de ce type `'a -> 'b -> ('a -> 'b -> 'c) -> 'c`.

Il est à noter que, tout comme une fonction peut être définie globalement à l'aide de `let`, elle peut être définie localement avec `let...in`.

#### Exercices

1. Définissez une fonction `f` qui accepte en argument un nombre à virgule `x` et calcule $x^2 - 2 \times x + 1$.
2. Définissez une fonction `presque_zero` qui accepte en argument un nombre flottant `epsilon` et produit comme résultat une fonction qui, pour tout nombre flottant `x`, vérifie si `-epsilon < x` et `x <`

`epsilon`.

3. À l'aide de `f` et de `presque_zero`, définissez une fonction `presque_racine` qui accepte en argument un nombre à virgule `x` et produit une deuxième fonction qui accepte en argument un nombre à virgule `epsilon` et vérifie si `-epsilon < f(x) < epsilon` .

**Types**

Maintenant que nous avons examiné la syntaxe, revenons à la réponse de OCaml lors de la définition de chacune des fonctions :

```
val ajouter_un     : int -> int = <fun>
val ajouter_deux   : int -> int = <fun>
val ajouter_trois  : int -> int = <fun>
val ajouter_quatre : int -> int = <fun>
```

La première ligne s'énonce "Nous venons de définir la valeur globale `ajouter_un`. Cette valeur est de type `int -> int` et il s'agit d'une fonction, que nous n'afficherons pas."

> **Note** OCaml n'affichera jamais le contenu d'une fonction.

Plus en détail, le type `int -> int` est le type des fonctions à un seul argument, qui est un entier, et dont le résultat est un entier. Mathématiquement, en considérant `int` comme l'ensemble des entiers OCaml, ce type signifie que la fonction est prise dans l'ensemble $int \longrightarrow int$, ou encore $int^{int}$.

En Java, on écrirait :

```
public static int ...(int ...)
```

À l'inverse, en OCaml, c'est l'ordinateur qui détermine le type de la fonction en analysant ses arguments et son corps. Ici, l'analyseur de types a remarqué la présence d'une addition sur les entiers, qui détermine que `x`, le seul argument de la fonction, est un entier et que le résultat de la fonction est un entier. La fonction est donc de type `int -> int`.

> **Vocabulaire** Lorsqu'un analyseur de types déduit automatiquement le type des valeurs au lieu de recourir à des annotations, on appelle ce processus l'*inférence de types*.

On peut s'interroger sur le comportement de l'analyseur lorsqu'il est confronté à des contraintes différentes, par exemple des contraintes qui imposent que x soit à la fois un entier et un nombre flottant. Observons :

```
# let f x = ( x + 1, x +. 1.0 ) ;;
                     ^
This expression has type int but is here used with type float
```

L'analyseur a conclu que le programme était erroné, puisqu'il partait à la fois du principe que `x` était un entier (pour additionner 1) et du principe que `x` était un nombre flottant (pour additionner 1.0). La réponse d'OCaml est donc que la fonction contient une erreur de types et que cette définition doit être refusée. En effet, on constate que `f` n'est pas définie :

```
# f 5 ;;
  ^
```

```
Unbound value f
```

On peut aussi s'interroger sur le comportement de l'analyseur lorsque `x` n'est pas contraint. Observons :

```
# let id x = x;;
val id : 'a -> 'a = <fun>
```

La réponse d'OCaml s'énonce "Nous venons de définir la valeur globale `id`. Cette valeur est de type *pour tout type α, α -> α*. Il s'agit d'une fonction."

Le type de la fonction `id` est dit *polymorphe sur le paramètre α*. Ainsi, on pourra utiliser `id` sur des entiers, des booléens ou n'importe quel autre type de données, fonctions y compris.

```
# id 5;;
- : int = 5
# id true;;
- : bool = true
# id 3.2;;
- : float = 3.2
# id ajouter_deux ;;
- : int -> int = <fun>
# id ( fun x -> x + 1 ) ;;
- : int -> int = <fun>
```

En Java ou C#, le polymorphisme paramétrique est connu sous le nom de "généricité" et est à la fois moins puissant, plus compliqué, plus lent, plus long à écrire et moins robuste que le polymorphisme paramétrique d'OCaml. Nous revisiterons fréquemment le polymorphisme, qui est présent à tous les niveaux d'OCaml, et qui constitue un outil puissant de réutilisation de code.

> **Note** Dans le dernier exemple, nous avons passé comme argument à la fonction `id` la fonction `ajouter_deux` puis, plus tard, une λ-expression `fun x -> x + 1`. Passer en argument des fonctions nommées ou anonymes est une pratique courante en OCaml. On s'en sert notamment pour les parcours de structures de données, les boucles, etc.

#### Exercices

1. Que signifie le type de la fonction `max` de la bibliothèque standard d'OCaml ?
2. Que signifie le type de la fonction `appliquer_operation` définie précédemment ?
3. Sans écrire la fonction suivante, déterminez son type et déterminez ce qu'elle fait

```
let multiplie n f =
  fun x -> n * ( f x ) ;;
```

1. Quel est le type d'une fonction à deux arguments qui ignore son premier argument et produit comme résultat son deuxième argument ?

#### Arguments supplémentaires

Jusqu'à présent, nous n'avons défini et manipulé que des fonctions acceptant un seul argument. Bien entendu, comme tous les autres langages de programmation, OCaml permet de manipuler des fonctions à plusieurs arguments.

En fait, ce n'est pas tout à fait vrai. Nous avons déjà manipulé une fonction à plusieurs arguments : la multiplication

```
# multiplication ;;
- : int -> int -> int
```

Comme nous pouvons le remarquer, le type de `multiplication` est `int -> int -> int`. Nous l'avons lu "La fonction `multiplication` accepte un argument de type `int` et produit un résultat qui est une fonction qui accepte à son tour un argument de type `int` et produit un résultat de type `int`." De fait, ce type peut aussi se lire "La fonction `multiplication` accepte deux arguments de type `int` et produit un résultat de type `int`."

Si nous souhaitons définir une fonction dont l'effet sera identique à `multiplication`, nous pouvons de nouveau employer plusieurs syntaxes équivalentes :

```
# let multiplication_2 = fun x y -> x * y ;;
val multiplication_2 : int -> int -> int = <fun>

# let mutiplication_2 = fun x -> fun y -> x * y ;;
val mutiplication_2 : int -> int -> int = <fun>

# let multiplication_2 x = fun y -> x * y ;;
val multiplication_2 : int -> int -> int = <fun>

# let multiplication_2 x y = x * y ;;
val multiplication_2 : int -> int -> int = <fun>

# let multiplication_2 = multiplication ;;
val multiplication : int -> int -> int = <fun>
```

La première syntaxe met en valeur le fait que `multiplication_2` est une fonction à deux arguments dont le résultat est de multiplier ses arguments. La deuxième syntaxe met en avant le fait que `multiplication_2` est une fonction à un argument dont le résultat est une fonction qui à son tour attend un argument avant de multiplier ces deux arguments. La troisième syntaxe insiste sur le fait que, pour tout `x`, `multiplication_2 x` est de nouveau une fonction qui à son tour attend un argument avant de multiplier cet arguments par `x`. La quatrième syntaxe présente la famille des valeurs `multiplication_2 x y` pour tout entier `x` et tout entier `y`, qui vaut alors `x * y`. Enfin, la dernière formulation présente le fait que `multiplication_2` est la même opération que `multiplication`.

Ces quatre syntaxes sont presque équivalentes et tout aussi valables. Comme précédemment, on préfèrera la plus courte, c'est-à-dire ici la dernière. Lorsqu'un tel raccourci n'est pas possible, on se rabattra sur la syntaxe précédente.

Insistons donc sur un fait : en OCaml, *une fonction à plusieurs arguments est une fonction à un argument et dont le résultat est encore une fonction.*

L'une des conséquences de cette conception des fonctions et qu'il est possible, avec une fonction à plusieurs arguments, de pré-appliquer la fonction certaines des valeurs de manière à obtenir une fonction spécialisée. Ainsi, on pourra écrire

```
# multiplication 2;;
- : int -> int = <fun>
```

```
# let f = multiplication 2 in f 4 ;;
- : int = 8
```

> **Vocabulaire** La transformation d'une fonction à *n* arguments en une fonction à *n-1* arguments s'appelle la *currification*. C'est une technique fréquemment employée en OCaml. Elle permet de composer aisément des fonctions et sert notamment pour construire aisément des fonctions à passer en argument à d'autres fonctions. Nous y reviendrons sous peu.

Une autre conséquence de cette conception des fonctions est qu'il n'existe pas, en OCaml, de fonction sans arguments. Nous reviendrons sur ce point un peu plus tard.

#### Exercices

1. Réécrivez les fonctions `appliquer_operation`, `presque_zero` et `presque_racine` sous une forme plus concise.
2. Définissez une fonction `presque_racine_generique` qui acceptera en argument une fonction `f` des nombres flottants vers les nombres flottants, une valeur flottante `epsilon` et une valeur flottante `x` et déterminera si `-epsilon < f x` et `f x < epsilon`.
3. Réécrivez la fonction `presque_racine` à partir de la fonction `presque_racine_generique`.

#### Fonctions et opérateurs

Avant d'approfondir notre étude des fonctions et d'introduire la notion de récursivité, arrêtons-nous un instant sur la définition des opérateurs. Tout comme en C++ ou en Python, en OCaml, les opérateurs sont en fait des fonctions. Ainsi, chaque utilisation de l'opérateur `*` est en fait un appel à la fonction `( * )` (les espaces et les parenthèses font partie du nom de la fonction), chaque utilisation de `+` est un appel à la fonction `( + )`...

Nous pouvons donc écrire

```
# ( * ) ;;
- : int -> int -> int = <fun>

# ( * ) 5 10 ;;
- : int = 50

# ( +. ) 1.5 2.10 ;;
- : float = 3.6

# ( = ) 5 3 ;;
- : bool = False

# let multiplication = ( * ) ;;
val multiplication : int -> int -> int = <fun>
```

> **Vocabulaire** L'opérateur `*` est la *forme infixe* de la multiplication alors que la fonction `( * )` est la *forme préfixe* de la multiplication. De base, en OCaml, il n'existe pas de *formes postfixes*.

En fait, lorsque l'analyseur syntaxique d'OCaml rencontre une expression de la forme `x * y`, celle-ci est comprise comme l'appel de fonction `( * ) x y`. Ceci est important à savoir pour plusieurs raisons :

- les opérateurs d'OCaml peuvent être masqués tout comme les autres valeurs, ce qui permet de redéfinir localement la syntaxe des expressions arithmétiques, par exemple pour écrire naturellement des opérations sur des nombres complexes, des vecteurs, des fonctions ...

- de nouveaux opérateurs peuvent être ajoutés à OCaml, localement ou globalement, avec le même genre d'applications
- cette définition des opérateurs est la cause de messages d'erreur parfois surprenants.

Ainsi, revenons à l'erreur que nous avons rencontrée plus tôt :

```
# ( 2 * ajouter_un ) 10 ;;
      ^
This function is applied to too many arguments, maybe you forgot a `;'
```

On pourrait s'attendre à ce que `2 * ajouter_un` provoque une erreur d'analyse de type -- rappelons que l'opérateur `*` ne concerne que des entiers et que `ajouter_un` n'est visiblement pas un entier. On peut aisément confirmer que la multiplication d'une fonction par deux n'est pas légale :

```
# 2 * ajouter_un ;;
      ^^^^^^^^^^
This expression has type int -> int but is here used with type int
```

Cependant, ce n'est pas l'erreur que nous rapporte OCaml. En fait, pour comprendre ce message, il faut se souvenir que l'extrait

```
( 2 * ajouter_un ) 10
```

est compris comme

```
( ( * ) 2 ajouter_un ) 10
```

ou encore, en supprimant les parenthèses inutiles,

```
( * ) 2 ajouter_un 10
```

En d'autres termes, la fonction `( * )` est bien appliquée à trois arguments, `2`, `ajouter_un` et `10`. Comme la fonction `( * )` n'accepte que deux arguments, l'erreur rencontrée par OCaml est effectivement une erreur sur le nombre d'arguments.

#### Récursivité

De nombreuses fonctions mathématiques et informatiques nécessitent la répétition d'une opération un grand nombre de fois ou jusqu'à ce qu'une condition soit remplie. Ainsi, pour calculer le *n*ème terme d'une suite définie par récurrence, on aura souvent besoin de commencer par calculer tous les termes précédents. De même, pour chercher numériquement la limite d'une fonction ou d'une suite, on aura souvent besoin de répéter des approximations jusqu'à obtenir deux termes successifs suffisamment proches, ou pour calculer le contour de l'ensemble de Mandelbrot, on répétera des calculs complexes un nombre limité de fois ou jusqu'à avoir dépassé un module donné. Du côté informatique du terme, pour traiter un grand nombre d'informations en provenance d'un périphérique tel que le clavier ou un capteur, on appliquera une fonction à chaque information reçue jusqu'à l'arrivée d'un contre-ordre ou d'un changement de configuration. Enfin, pour de nombreux problèmes mathématiques et informatiques, chercher la solution nécessite de se ramener constamment à des sous-problèmes de plus en plus simples jusqu'à être confronté à des questions élémentaires. C'est ainsi que l'on trouve la solution à des problèmes de recherche de chemins dans des cartes

géographiques (méthode A* hiérarchique), qu'on calcule la valeur d'une expression arithmétique ou logique complexe, qu'on localise une valeur dans un arbre ou qu'on analyse les interactions entre corps célestes dans une approximation numérique du problème des n corps par quad-trees.

Tous ces comportements se traitent de la même manière. Dans la majorité des langages de programmation, on emploie les boucles impératives `for` et `while` ou `repeat` -- nous laisserons provisoirement de côté les boucles `for-each` et autres *compréhensions* présentes dans certains langages récents dont OCaml Batteries Included. En Objective Caml, si tous ces mécanismes sont disponibles, nous allons commencer par étudier le mécanisme plus général de *récursivité*, qui reprend en informatique les notions mathématiques de *récurrence* et d'*induction structurelle*.

Le principe de la récursivité est simple :

- si vous savez ce qu'est la récursivité, vous avez votre réponse
- si vous ne savez pas ce qu'est la récursivité, trouvez quelqu'un qui est plus proche que vous de Douglas Hofstadter et posez-lui la question. (Citation en substance, attribuée à Andrew Plotkin).

En pratique, on considérera plutôt l'approche suivante :

- si le problème est suffisamment simple pour pouvoir déterminer la solution par une manipulation simple, procéder à cette manipulation
- si le problème est plus complexe, commencer par le simplifier en un ou plusieurs problèmes plus simples de même structure, résoudre chacun de ces sous-problèmes, puis combiner les sous solutions pour les transformer en une solution au problème complet.

Illustrons ceci par le calcul de la somme des nombres de m à n :

- si n est inférieur à m, cette somme est nulle
- sinon, il suffit de calculer la somme des nombres de m + 1 à n et de lui ajouter m.

En OCaml, cette définition se traduira par

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

# somme_des_nombres 1 5 ;;
- : int = 15
```

Le mot-clé `rec` introduit une fonction récursive, c'est-à-dire donc une fonction qui peut s'appeler elle-même. Sans ce `rec`, `somme_des_nombres` ne serait pas lié à l'intérieur du corps de la fonction. On obtiendrait donc :

```
# let somme_faux m n =
    if n < m then
      0
    else
      m + ( somme_faux ( m + 1 ) n ) ) ;;
            ^^^^^^^^^^
Unbound value somme_faux
```

Attardons-nous un moment sur l'évaluation de `somme`. Pour obtenir des détails sur l'utilisation d'une fonction,

la ligne de commande OCaml dispose de l'instruction `#trace` :

```
# #trace somme ;;
somme_des_nombres is now traced.

# somme 1 3;;
somme <-- 1
somme --> <fun>
somme* <-- 3
somme <-- 2
somme --> <fun>
somme* <-- 3
somme <-- 3
somme --> <fun>
somme* <-- 3
somme <-- 4
somme --> <fun>
somme* <-- 3
somme* --> 0
somme* --> 3
somme* --> 5
somme* --> 6
- : int = 6
```

Cette réponse sera plus compréhensible avec quelques annotations :

```
# somme 1 3 ;;    (* Évaluons somme 1 3                                  *)

(* Début de l'évaluation de somme 1 3                                   *)
somme <-- 1       (* pour ce faire, évaluons  somme 1                   *)
somme --> <fun>   (* le résultat est une fonction                       *)
somme* <-- 3      (* évaluons ce résultat appliqué à 3                  *)
                             (* l'évaluation de somme 1 3 commence                *)
                             (* cette évaluation fait appel à la valeur somme 2 3 *)

 (* Début de l'évaluation de somme 2 3                                  *)
 somme <-- 2      (* Évaluons donc somme 2                              *)
 somme --> <fun>  (* le résultat est une fonction                       *)
 somme* <-- 3     (* évaluons ce résultat appliqué à 3                  *)
                             (* l'évaluation de somme 2 3 commence                *)
                             (* cette évaluation fait appel à la valeur somme 3 3 *)

  (* Début de l'évaluation de somme 3 3                                 *)
  somme <-- 3     (* Évaluons donc somme 3                              *)
  somme --> <fun> (* le résultat est une fonction                       *)
  somme* <-- 3    (* évaluons ce résultat appliqué à 3                  *)
                             (* l'évaluation de somme 3 3 commence                *)
                             (* cette évaluation fait appel à la valeur somme 4 3 *)

   (* Début de l'évaluation de somme 4 3                                *)
   somme <-- 4    (* Évaluons donc somme 3                              *)
   somme --> <fun>(* le résultat est une fonction                       *)
   somme* <-- 3   (* évaluons ce résultat appliqué à 3                  *)
                             (* l'évaluation de somme 3 3 commence    *)
          (* Nous avons toutes les informations nécessaires pour évaluer somme 4 3 *)
   somme* --> 0   (* Le résultat de cette évaluation est 0              *)
   (* Fin de l'évaluation de somme 4 3                                  *)

          (* Nous avons toutes les informations nécessaires pour évaluer somme 3 3 *)
  somme* --> 3    (* Le résultat de cette évaluation est 3 + 0, soit 3 *)
  (* Fin de l'évaluation de somme 3 3                                  *)
```

```
          (* Nous avons toutes les informations nécessaires pour évaluer somme 2 3 *)
 somme* --> 5     (* Le résultat de cette évaluation est 2 + 3, soit 5 *)
 (* Fin de l'évaluation de somme 2 3                                    *)

          (* Nous avons toutes les informations nécessaires pour évaluer somme 1 3 *)
somme* --> 6      (* Le résultat de cette évaluation est 1 + 5, soit 6 *)
(* Fin de l'évaluation de somme 1 3                                    *)
- : int = 6
```

Ainsi, pour évaluer `somme 1 3`, OCaml aura commencé par se ramener au cas `somme 2 3` puis au cas `somme_des_nombres 3 3` puis encore au cas `somme 3 4`, qui lui se calcule directement. À partir du résultat de `somme 3 4`, il est aisé de reconstruire le résultat de `somme 3 3`, qui lui-même permet de reconstruire aisément le résultat de `somme 3 2`, qui lui-même permet de reconstruire aisément le résultat de `somme 3 1`.

> **Note** En termes d'expressivité, le mécanisme de récursivité est exactement aussi puissant que les boucles impératives présentes dans la majorité des langages de programmation. En pratique, il est plus simple de traduire un programme écrit à l'aide d'une boucle impérative en un programme écrit à l'aide d'une boucle récursive que de faire la manipulation inverse. En effet, dans la majeure partie des cas, implanter un algorithme récursif à l'aide d'une boucle impérative nécessite de manipuler des structures de piles contenant elles-même des structures de données peu lisibles.

Notre boucle récursive n'a pas à être déclarée au plus haut niveau. On peut aussi la cacher à l'intérieur de la fonction, à l'aide de `let...in`. Ainsi, en appelant `aux` la boucle récursive, on obtiendra :

```
# let somme m n =
    let rec aux m' n' =
      if n' < m' then
        0
      else
        m' + ( aux ( m' + 1 ) n' ) )
    in
      aux m n;;

val somme : int -> int -> int = <fun>
```

Comme nous pouvons l'observer dans ce qui précède, à chaque appel récursif de `aux`, le nom `n'` reste lié à la valeur de `n`. Ainsi, au lieu d'utiliser `n'`, nous pouvons donc utiliser directement `m`. Nous obtenons alors

```
# val somme m n =
    let rec aux m' =
      if n < m' then
        0
      else
        m' + ( aux ( m' + 1 ) n ) )
    in
      aux m;;

val somme : int -> int -> int = <fun>
```

Notons que le nom `m'` tient ici un rôle similaire à celui d'une variable compteur `i` dans une boucle impérative `for(int i = 0; i <= ... ; ++i)`. Notons aussi que, dans cet exemple, l'intérêt principal d'avoir transformé la définition pour faire intervenir une boucle `aux` est de se débarrasser des arguments inutiles (ici `n'`). Dans des programmes complexes, il sera fréquemment utile de faire intervenir des

fonctions/boucles de cette forme.

> **Note** La somme des nombres de 1 à n peut très bien s'écrire en une seule ligne à l'aide de `1 -- n`. Cette deuxième méthode est plus succinte et plus lisible mais fait appel à des concepts plus avancés et que nous ne rencontrerons que dans un chapitre ultérieur.

La structure employée sera la même lorsque nous chercherons à définir la fonction factorielle. Rappelons que, mathématiquement, la factorielle peut se définir par récurrence sur l'ensemble des entiers par

- si *n < 1* alors *factorielle(n) = 1*
- si *n >=1* alors *factorielle(n) = n * factorielle ( n - 1 )*.

On écrira donc de même

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

# factorielle 5 ;;
- : int = 120
```

Ce que nous pourrons aussi réécrire en cachant la boucle

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

# factorielle 5 ;;
- : int = 120
```

Dans les exemples qui précèdent, nous nous sommes toujours ramenés à un cas plus simple en incrémentant et en décrémentant un entier de 1, à l'image d'une boucle `for`. Bien entendu, la récursivité ne se limite pas à ce type de cas. Ainsi, l'algorithme d'exponentiation rapide, qui permet de calculer $a^n$ sans procéder à `n` multiplications, emploie une structure légèrement plus compliquée.

La définition mathématique de cet algorithme est la suivante :

- $a^0 = 1$
- $\forall k, a^{2 \cdot k} = (a^k)^2$
- $\forall k, a^{2 \cdot k + 1} = a \cdot (a^k)^2.$

Rappelons que la notation "2k" signifie "un nombre pair *n* pour lequel nous noterons *k = n/2*. De même, "2k+1" signifie "un nombre impair *n* pour lequel nous noterons *k = (n-1)/2*." Cette définition se traduit par

- si *n=0*, le résultat est 1
- si *n>0* et si *n* est pair, commençons par calculer $x = a^{n/2}$ ; le résultat est alors $x^2$
- si *n>0* et si *n* est impair, commençons par calculer $x = a^{\lfloor n/2 \rfloor}$ ; le résultat est alors $x^2 \cdot a$

En OCaml, on écrira donc :

```
# let rec puissance a n =
   if n = 0 then 1
   else
     if n mod 2 = 0 then let x = puissance a ( n / 2 ) in x * x
     else                let x = puissance a ( n / 2 ) in x * x * a ;;
val puissance : int -> int -> int = <fun>
```

En fait, on préfèrera modifier légèrement cette définition pour calculer `x` et `x * x` une seule fois :

```
# let rec puissance a n =
   if n = 0 then 1
   else
     let x = puissance a ( n / 2 ) in
     let y = x * x                 in
       if n mod 2 = 0 then y
       else                y * a;;
val puissance : int -> int -> int = <fun>
```

Notons que la récursivité ne se limite pas à *une* fonction s'appelant elle-même. Ainsi, lors de l'analyse de langages de programmation, voire de langages humains, il est courant de se retrouver confronté à des dizaines ou des centaines de fonctions *mutuellement récursives*. Nous rencontrerons cec genre d'exemple à la fin du prochain chapitre. Pour le moment, nous nous contenterons d'une méthode extraordinairement complexe pour déterminer la parité d'un nombre positif :

```
# let rec pair   n = n <> 1 && ( n = 0 || impair ( n - 1 ) )
  and     impair n = n <> 0 && ( n = 1 || pair   ( n - 1 ) ) ;;
val pair   : int -> bool = <fun>
val impair : int -> bool = <fun>

# pair 5;;
- : bool = false
# pair 6;;
- : bool = true
```

Le mécanisme de récursivité reviendra à tous les niveaux dans OCaml.

#### Exercice

1. Définissez une fonction `somme_des_carres m n` qui calcule la somme des carrés des entiers entre `m` et `n`.
2. Réécrivez la fonction `factorielle` de manière à changer l'ordre d'intervention des nombres. Dans la version présentée plus haut, le cas de base renvoie `1`, qu'on multiplie successivement par `2` ... `n`. Inversez cet ordre.
3. Écrivez une fonction `pgcd` en partant de la définition suivante :

- *pgcd (2m, 2n) = 2 pgcd (m, n)*
- *pgcd (2m, 2n+1) = pgcd (m, 2n+1)*
- *pgcd (2m+1, 2n+1) = pgcd(n-m, 2m+1)* lorsque *m<n*
- *pgcd (m,m) = m.*

Rappelons que la notation mathématique *2m* signifie "un nombre pair p pour lequel nous noterons m = p/2", et *2m+1* signifie "un nombre impair i pour lequel nous noterons m = (i-1)/2".

1. De la même manière que `pair` et `impair`, écrivez trois fonctions mutuellement récursives pour déterminer si un nombre est divisible par 3 ou si le reste de sa division par 3 est 1 ou si le reste de sa division par 3 est 2.

#### Réutilisation de fonctions

De la même manière que nous avons défini la somme des nombres entre `m` et `n`, nous pouvons définir le produit des nombres entre ces bornes :

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

En tout et pour tout, nous avons changé

- le nom de la fonction
- le 0, remplacé par 1
- l'addition, remplacée par une multiplication.

Face à ces deux fonctions extrêmement proches, le réflexe du programmeur OCaml sera de chercher comment écrire une seule fonction qui pourra être spécialisée pour devenir l'une ou l'autre. C'est le processus d'*abstraction*, qui à la source de la réutilisation de code et du développement de bibliothèques puissantes de programmation.

> **Note** Le terme de "patrons de conception" (design patterns) n'existe pas en programmation fonctionnelle. Cependant, s'il existait, le processus d'abstraction figurerait en deuxième place dans la liste des patrons de conception, juste après le développement de fonctions polymorphes paramétriques.

Pour écrire une telle fonction,

- choisissons un nom pour notre fonction abstraite -- mettons `segment_fold_right`
- faisons du 0 (devenu un 1 dans le produit) une valeur `v`, qui sera un argument de notre fonction
- faisons de l'addition (devenue une multiplication dans le produit) une valeur `f`, qui sera aussi un argument de notre fonction
- n'oublions pas de passer `v` et `f` à `operation_sur_intervalle` lors de l'invocation récursive.

La transformation produit donc :

```
# let rec operation_sur_intervalle f v m n =
    if n <= m then v
    else           f m ( operation_sur_intervalle f v ( m + 1 ) n ) ;;
val operation_sur_intervalle : (int -> 'a -> 'a) -> 'a -> int -> int -> 'a =
  <fun>
```

ou encore

```
# let operation_sur_intervalle f v m n =
    let rec aux m' =
      if n <= m' then v
      else            f m' ( aux ( m' + 1 ) )
    in aux m;;
```

À partir de là, nous pouvons obtenir de nouveau la somme et le produit des nombres en appliquant directement notre nouvelle fonction (rappelons que `( + )` et `( * )` sont les fonctions correspondant aux opérateurs + et *) :

```
# let somme_des_nombres_2   = operation_sur_intervalle ( + ) 0 ;;
val somme_des_nombres_2 : int -> int -> int = <fun>

# let produit_des_nombres_2 = operation_sur_intervalle ( * ) 1 ;;
val produit_des_nombres_2 : int -> int -> int = <fun>
```

À ce prix-là, nous pouvons définir en une ligne la fonction factorielle, la somme des carrés des entiers...

```
# let factorielle_2         = produit_des_nombres_2 1 ;;
val factorielle_2 : int -> int = <fun>

# let somme_des_carres_2    = operation_sur_intervalle ( fun x acc -> x * x + acc ) 0 ;;
val somme_des_carres_2 : int -> int -> int = <fun>
```

On peut aussi construire à partir de `operation_sur_intervalle` des fonctions similaires qui permettront de travailler sur les carrés des éléments d'un intervalle ou sur les nombres `x` de l'intervalle tels que `p x` est vrai ou sur les nombres impairs de l'intervalle...

```
# let operation_sur_carres_intervalle f v = operation_sur_intervalle ( fun x acc -> f ( x
val carres_fold_right : (int -> 'a -> 'a) -> 'a -> int -> int -> 'a = <fun>

# let operation_sur_intervalle_filtre f p v = operation_sur_intervalle (fun x acc -> if p
val filtre_fold_right :
  (int -> 'a -> 'a) -> (int -> bool) -> 'b -> 'a -> int -> int -> 'a = <fun>

# let operation_sur_nombres_pairs_intervalle f = operation_sur_intervalle_filtre f ( fun
val operation_sur_nombres_pairs_intervalle : (int -> 'a -> 'a) -> 'b -> 'a -> int -> int
  <fun>

# let operation_sur_nombres_impairs_intervalle f = operation_sur_intervalle_filtre f ( fu
val operation_sur_nombres_impairs_intervalle : (int -> 'a -> 'a) -> 'b -> 'a -> int -> in
  <fun>
```

Cette possibilité d'abstraire des fonctions en des fonctions plus génériques qui seront plus tard spécialisées à l'aide d'arguments fonctionnels constitue l'un des avantages majeurs de la programmation fonctionnelle. On l'utilise notamment pour définir de nouveaux types de boucles sur des structures de données personnalisées -- tout comme ici sur les intervalles, les ensembles de nombres impairs, les ensembles de nombres impairs, etc. -- pour la gestion sûre des ressources externes, là où dans d'autres langages on emploierait `try...finally`, etc. Combiné avec la vérification de types d'OCaml, ceci permet de concevoir des bibliothèques sûres de manipulation de ressources et de structures de données.

Pour comparaison, en Java, cette notion d'intervalle s'écrirait de la manière suivante

```
//Définition de l'intervalle
public class Interval implements Iterable<Integer>
{
  //...
  public final int begin;
  public final int end;
  public Iterator<Integer> iterator ()
  {
```

```
    return new IntervalIterator(this.begin, this.end);
  }
}

//Définition de l'itération sur tous les entiers d'un intervalle d'entiers
public class IntervalIterator implements Iterator<Integer>
{
  private int current ;
  private final int end ;
  public IntervalIterator(int begin, int end)
  {
    this.current = begin;
    this.end     = end;
  }
  public final Integer next ()
  {
    if(this.current < this.end)
      return this.current++;
    else
      throw new NoSuchElementException(this.current);
  }
  public void remove()
  {
     this.current++;
  }
  public bool hasNext()
  {
    return this.current < this.next ;
  }
}

//Calcul de la somme des entiers
public int sommeDesEntiers(int debut, int fin)
{
  int total = 0;
  for(Integer i : new Interval(debut, fin))
    total += i;
  return total;
}
```

Ajouter les filtres à cette version Java est possible mais compliquerait considérablement l'extrait.

D'autres langages, comme Python, emploient une syntaxe plus concise mais moins générique, spécialisée dans la définition de boucles sur les structures de données : les *générateurs*. Tout ceci est aussi disponible en OCaml, sous le nom d'*énumérations*, et sera détaillé plus tard.

#### Exercices

1. De la même manière que `operation_sur_intervalle`, définissez une fonction `operation_sur_intervalle_inverse` qui parcourra l'intervalle dans l'autre sens. Vous testerez qu'elle fonctionne en calculant à l'aide de cette fonction la somme des entiers dans divers intervalles.
2. Définissez une fonction similaire à `operation_sur_intervalle_filtre` mais qui n'applique son argument `f` qu'aux entiers `x` de l'intervalle tels que

- `x` soit divisible par 3 ou par 7
- **et** `x` vérifie le prédicat `p` (c'est-à-dire `p x` soit vrai).

Pour ce faire, vous réutiliserez `operation_sur_intervalle_filtre`.

#### Limitations et précautions

#### Affichage de fonctions

Comme nous l'avons noté, OCaml n'affiche jamais une fonction. En effet, les fonctions sont compilées immédiatement et leur représentation lisible oubliée. Après cela, il reste uniquement possible de consulter le type de la fonction.

#### Comparaison de fonctions

En OCaml, deux fonctions ne peuvent jamais être comparées à l'aide des opérateurs =, <, >, >=, <=, <> :

```
# id < id ;
Exception: Invalid_argument "equal: functional value".
```

La raison à cette limitation est à l'intersection du monde des mathématiques et de celui de l'informatique. En effet, rappelons que, mathématiquement, si nous avons $\begin{cases} f : A \longrightarrow B \\ g : A \longrightarrow B \end{cases}$ alors, par définition, $\begin{cases} f = g \iff \forall x \in A, f(x) = g(x) \\ f < g \iff \forall x \in A, f(x) < g(x) \\ f > g \iff \forall x \in A, f(x) > g(x) \end{cases}$. En d'autres termes, pour comparer deux fonctions, il ne suffit pas de vérifier si elles ont été écrites de la même manière, mais il faut tester chacune des deux fonctions avec toutes les valeurs possibles. Ceci n'est pas faisable sur les machines actuelles -- et ne sera probablement jamais faisable, à moins que les machines quantiques fassent un jour leur apparition. Plutôt que de fournir une demi-solution, les programmeurs OCaml ont préféré jeter l'éponge et abandonner les comparaisons entre fonctions.

Malheureusement, l'analyseur de types d'OCaml n'est pas assez puissant pour déterminer lors de la compilation si l'opérateur = va être appliqué à deux fonctions. Cette vérification ne peut donc être effectuée que durant l'exécution, à l'instar des vérifications dynamiques de Java, C# ou de celles, omniprésentes, de Python, Ruby ou JavaScript.

Il reste possible de vérifier l'égalité physique entre deux fonctions à l'aide de l'opérateur ==. Cet opérateur, dont la sémantique est similaire à celle du même opérateur en Java, permet de vérifier si deux valeurs sont exactement la même (et non pas si elles ont le même résultat), comme suit :

```
# id == id ;;
- : bool = true
# id == ( fun x -> x );;
- : bool = false
# let x = id in id == x ;;
- : bool = true
```

#### Modulo

L'opérateur `mod` n'est pas associé à une fonction `( mod )`. Il s'agit probablement du seul opérateur OCaml à ne pas être traduit sous la forme d'une fonction portant le même nom. Sans entrer dans les détails pour le moment, tous les autres opérateurs OCaml sont gérés par le même mécanisme d'analyse syntaxique, alors que `mod` dispose d'un mécanisme pour lui seul.

Il s'agit d'une irrégularité mineure dans le langage OCaml. Il semble probable que cette irrégularité sera corrigée dans la prochaine version majeure, dans laquelle la gestion des opérateurs est en cours de révision.

#### Couples d'arguments

Le premier réflexe d'un programmeur habitué à un autre langage de programmation est généralement de définir une fonction à plusieurs arguments avec la syntaxe suivante :

```
# let multiplication_2 ( x, y ) = x * y;;
val multiplication_2 : (int * int) -> int = <fun>
```

Cette syntaxe est tout à fait légitime mais, comme on peut le constater en examinant le type du résultat, elle ne produit pas la même fonction que `multiplication`. En effet, le type de `multiplication` place cette fonction dans l'ensemble $int \longrightarrow (int \longrightarrow int)$, c'est-à-dire $int^{int^{int}}$. À l'inverse, le type de `multiplication_2` place cette fonction dans l'ensemble $(int \times int) \longrightarrow int$ ou encore $int^{int \times int}$.

Ceci est du au fait que `( x, y)` représente, comme en mathématiques, le *couple* $(x, y)$, pris dans l'ensemble $int \times int$, c'est-à-dire le type `int * int`. En OCaml, on préférera généralement l'approche présentée avec `multiplication`, qui favorise la réutilisation et la composition de fonctions.

#### Mémoire des résultats

Précisons que, par défaut, OCaml ne garde pas en mémoire les résultats d'un calcul. Ainsi, si l'on invoque

```
# factorielle 5 ;;
- : int = 120

# factorielle 6 ;;
- : int = 720
```

Le calcul de `factorielle 6` ne réutilisera aucun des résultats de `factorielle 5`. Ce comportement par défaut, commun à presque tous les langages de programmation (sauf Maple et quelques autres langages spécialisés dans les mathématiques), vise à éviter de gaspiller la mémoire vive à retenir des résultats qui seront a priori probablement inutiles. Il est cependant possible de demander à OCaml de conserver ces résultats en mémoire, c'est le processus de *mémoification*. Nous verrons comment ajouter la mémoification dans le chapitre consacré à l'optimisation.

#### Profondeur de récursivité

Généralement, une fonction récursive écrite de manière naïve consommera plus de mémoire vive qu'une boucle impérative, en raison de la nécessité de retenir des informations pour rassembler les résultats des sous-problèmes. En particulier, une telle boucle récursive, si elle est appelée sur un échantillon trop long, provoquera un *débordement de pile*, c'est-à-dire une erreur fatale qui interrompra le calcul. Si la situation est moins grave que dans beaucoup de langages de programmation, grâce à la faible consommation de mémoire de OCaml, on cherchera à tirer le plus possible parti de la notion de *boucles récursives terminales*, que le compilateur OCaml est capable d'optimiser pour supprimer ce problème.

Nous discuterons plus avant cette récursivité terminale dans le chapitre consacré à l'optimisation. Pour le moment, nous nous contenterons de la définition informelle suivante : une fonction est récursive terminale si le résultat est entièrement construit lors de la division en sous-problèmes et non pas dans une phase ultérieure de reconstitution.

Les boucles récursives terminales ont exactement le même pouvoir expressif que les boucles impératives.

**À retenir**

- Les fonctions sont des valeurs comme les autres.
- Les appels de fonctions sont associatifs à gauche.
- Les parenthèses servent uniquement à forcer la priorité des opérations.
- Le message

```
This function is applied to too many arguments, maybe you forgot a `;'
```

désigne une erreur de typage : la fonction soulignée est appliquée à plus d'arguments qu'elle n'en accepte.

- Les valeurs sont typées automatiquement, y compris les fonctions polymorphes.
- Une fonction à plusieurs arguments est une fonction à un argument dont le résultat est encore une fonction.
- Un type qui fait intervenir `'a` est dit polymorphe et la valeur de `'a` peut être remplacé par n'importe quel type.
- Pour répéter un traitement, on fait appel à des boucles récursives, introduites par `let rec...`.

# À propos de la programmation fonctionnelle

Plus haut dans ce chapitre, nous avons défini la somme des nombres

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

Comparons ceci à un équivalent écrit dans un langage impératif, ici Java :

```
public static int sommeDesNombres(int m, int n)
{
  int total = 0;
  for(int i = m; i <= n; ++i)
    total += i;
  return total;
}
```

Les deux extraits sont de taille sensiblement équivalente et beaucoup de programmeurs auront plus de facilité à lire le code Java, peut-être parce qu'ils ont généralement été formés sur des langages impératifs.

La philosophe impérative de Java recommande d'utiliser deux variables intermédiaires `total` et `i`, dont la valeur va changer plusieurs fois au cours de l'exécution de la fonction. Au contraire, la philosophie fonctionnelle d'OCaml recommande de ne jamais réaffecter une valeur à une variable existante mais plutôt de se *ramener au cas* où la valeur de la variable est différente, de la même manière qu'en mathématiques -- ici en appelant récursivement la fonction `somme_des_nombres`.

Dans l'absolu, aucune des deux philosophies n'est meilleure. L'une d'entre elles provient du monde de la Physique, dans lequel il est normal que l'état de l'univers change en permanence. L'autre provient du monde des Mathématiques, dans lesquelles le modèle est immuable mais on peut se placer dans des cas différents. Peut-être l'histoire de l'informatique aurait-elle été différente si les premiers programmeurs avaient été plus

mathématiciens que physiciens, mais depuis la machine de Von Neumann le modèle sous-jacent à la partie électronique des ordinateurs est un modèle dans lequel l'univers change effectivement en permanence, sous la forme de modifications de la mémoire vive ou du disque dur.

Employer l'approche physique/impérative permet de représenter naturellement des interactions avec le monde extérieur au programme (afficher quelque chose à l'écran ou sauvegarder un fichier, par exemple), qui n'ont pas de contrepartie simple en mathématiques. Cependant, comme c'est fréquemment le cas dès qu'il s'agit de l'interaction entre Physique et Mathématiques, si l'approche Physique permet d'avancer plus rapidement vers un résultat qui semble correct à première vue, l'approche Mathématique est nécessaire dès qu'il s'agit de s'assurer que les résultats sont effectivement corrects et réutilisables. Dans le monde de l'informatique, cela signifie qu'un développeur lambda produira souvent plus vite un programme impératif qu'un programme fonctionnel, mais qu'il sera plus difficile d'être certain que le programme fonctionne effectivement et que le programme sera fréquemment moins réutilisable. En effet, l'approche plus rigoureuse de la programmation fonctionnelle demandera au développeur plus de réflexion avant d'écrire son programme -- ou en tout cas avant d'arriver à le compiler -- mais vérifiera automatiquement plus de propriétés de sûreté et garantira par construction des propriétés telles que la réutilisabilité du code (un code qui ne modifie pas l'état de l'univers peut être utilisé dans n'importe quelles circonstances) ou l'optimisation du code (un code qui ne modifie pas l'état de l'univers peut être transformé en code parallèle, exécuté sur un autre microprocesseur ou un autre ordinateur, sans avoir à changer le reste du programme).

Encore une fois, cela ne signifie pas qu'une philosophie est meilleure que l'autre. Les deux états d'esprit sont nécessaires pour faire progresser un projet ou l'informatique dans son ensemble.

## Bilan du chapitre

Au cours de ce chapitre, nous avons couvert

- l'installation d'OCaml
- le lancement d'OCaml en ligne de commande
- l'utilisation d'OCaml en tant que calculatrice, les entiers, les nombres à virgule flottante et les booléens
- la définition et l'utilisation de nouvelles valeurs et la notion de liaison
- la définition et l'utilisation de fonctions
- la currification de fonctions
- l'abstraction de fonctions pour permettre leur réutilisabilité
- la notion de types, de types de fonctions, de types polymorphes.
- les messages d'erreur qui vont avec tout cela.

Dans le prochain chapitre, nous allons couvrir plus en profondeur la notion de types et de structures de données.

# Structures

Dans presque tous les langages de programmation, l'une des phases les plus importantes lors de la conception d'un programme est de décider comment représenter les données du programme, qu'il s'agisse de ressources internes, telles que des valeurs numériques ou des fonctions ou des ressources extérieures au programme, telles que des imprimantes ou des fichiers. Dans tous les cas, le choix de bonnes représentations de données -- ou *structures de données* -- sera nécessaire pour écrire des programmes rapides, concis, corrects et dont le code source ressemblera aux spécifications.

Ainsi, considérons le cas des nombres complexes. Manipuler numériquement les nombres complexes

nécessite de retenir la valeur de ces nombres, soit la partie réelle et la partie imaginaire, soit l'argument et le module, soit les deux à la fois. Dans un monde idéal, il n'y aurait aucune différence entre ces trois représentations. En pratique, sur un ordinateur, cependant, passer de la représentation polaire (argument et module) à la représentation cartésienne (partie réelle et partie imaginaire) ou réciproquement implique des calculs trigonométriques relativement lents et des erreurs d'arrondis. Une représentation mixte dans laquelle les coordonnées polaires et cartésiennes sont connues à tout instant, à son tour, gaspillerait beaucoup de temps et de précision à synchroniser systématiquement ces informations. Il convient donc de choisir une structure de données adaptée au problème et qui minimisera ces impondérables. Une autre contrainte importante sera d'adopter une représentation qui évitera le plus possibles d'erreurs de programmation, d'une part en évitant de confondre accidentellement les deux notations, d'autre part pour éviter de créer des valeurs sans aucun sens, telles que celles qu'on obtiendraient avec un argument strictement négatif.

Dans le cadre de logiciels non mathématiques, citons de même la représentation des fichiers. Dans certains langages ou certaines bibliothèques, dès qu'il s'agira de manipuler un fichier, par exemple pour consulter son contenu ou pour le modifier, il sera nécessaire de préciser le nom du fichier. Dans d'autres langages, toutes les opérations sur un fichier doivent être précédées d'une *ouverture*, qui associe au fichier un numéro unique et c'est ce numéro qui est manipulé par chacune des actions ultérieures de consultation ou de modification. Enfin, une troisième possibilité fait aussi appel à une phase d'ouverture et associe au fichier directement l'ensemble des actions qui pourront être entreprises sur ce fichier, sans passer par un entier ou une autre valeur concrète. De ces trois techniques, quelle est la meilleure ? La première est certainement la plus simple à comprendre mais c'est aussi la plus lente (à chaque opération, il est nécessaire de retrouver sur le disque ou en mémoire le fichier correspondant au nom qui vient d'être donné) et elle est aussi fort fragile, puisqu'aucun garde-fou ne permet de faire la différence entre un nom de fichier prévu pour être consulté, un nom de fichier prévu pour être modifié ou encore un nom qui n'a rien à voir avec un fichier. La deuxième technique est plus rapide et surtout plus simple à ajouter à un langage qui ne disposerait pas encore d'une bibliothèque de gestion de fichiers mais aussi plus complexe à manipuler (puisqu'il faut garder trace du numéro associé au fichier qui nous intéresse) et tout aussi fragile, puisqu'aucun garde-fou ne permet d'éviter de confondre le numéro d'un fichier ou le résultat d'une opération arithmétique. La troisième méthode est celle qui sera favorisée en OCaml. Si elle est plus complexe à ajouter à un langage (quelques langages ne seraient d'ailleurs pas assez puissant pour permettre cette méthode) et si elle est tout aussi abstraite que la méthode précédente, il s'agit de la plus robuste des trois, puisqu'il est impossible d'essayer de modifier un fichier qui a été ouvert pour être uniquement consulté et puisqu'il est impossible de confondre la représentation du fichier (ici, les opérations de manipulation) avec la représentation de quelque chose qui n'aurait rien à voir avec un fichier. Enfin, en théorie, cette dernière technique pourrait être la plus rapide, car cette robustesse permet d'éviter un certain nombre de tests qui sont fondamentaux dans des approches plus fragiles pour éviter les désastres.

En OCaml, une *structure de données* est composée d'un ou plusieurs *types* et les valeurs nécessaires pour manipuler les valeurs portant ces types. Ainsi, la structure des données des entiers peut être définie par le type `int`, les constantes `0`, `1`... et les quelques opérations de base. Au cours de ce chapitre, nous allons étudier les différentes techniques pour définir de nouvelles structures de données. À partir du chapitre suivant, nous commencerons à employer les structures de données déjà disponibles sous OCaml Batteries Included.

# Objectifs du chapitre

Ce chapitre vous enseignera

- le rôle des types
- les types et structures de données les plus utilisées en OCaml
- comment définir de nouveaux types et de nouvelles structures de données.

# Types de base

Jusqu'à présent, nous nous sommes intéressés essentiellement aux *valeurs*. Comme nous l'avons vu, en OCaml et comme dans la majorité des langages de programmation fortement typés, chaque valeur est associée à un *type de donnée* (ou juste "type").

Mathématiquement, les types peuvent être considérés comme une approximation de la notion d'*ensembles*. Ainsi, à partir de quelques ensembles simples et supposés connus, de nouveaux ensembles peuvent être construits par produit cartésien, par somme (union étiquetée), par génération de monoïdes libres, de fonctions, de quantificateurs existentiels ou universels, etc. Informatiquement, les types peuvent être considérés comme la définition de *comment une information doit être représentée en mémoire*. À partir de quelques types *primitifs* dont la spécification est supposée connue par OCaml, de nouvelles représentations peuvent être construites sous la forme de tuples de types, de types variants, de listes, de fonctions, de types paramétrés, etc.

Dans la quasi-totalité des langages de programmation, l'utilisation des types est contrainte par un *système de types*, dont le rôle est de garantir que les valeurs des différents types sont combinées correctement : un booléen ne peut pas être utilisé comme une fonction, une fonction n'est pas un texte, la fonction `int_of_float` s'applique à un nombre de type `float` et produit un nombre de type `int`... En OCaml, comme c'est presque systématiquement le cas en Java ou en C#, avant d'exécuter un programme, une phase d'*analyse de types* vérifie que l'utilisation des valeurs dans un programme est cohérente avec le système de types. Ce processus est dit *statique* car il a lieu avant l'exécution du programme, durant la compilation. D'autres langages fortement typés, comme Python, vérifient ces contraintes *dynamiquement*, c'est-à-dire durant l'exécution du programme. La vérification dynamique est plus souple mais plus lente et moins rigoureuse : là où l'analyse statique rejette un certain nombre de programmes corrects parce qu'ils risquent de produire des résultats incohérents -- c'est-à-dire parce que leur résultat ne peut être expliquée en termes d'ensembles -- l'analyse dynamique ne rejette a priori aucun programme, au risque de laisser passer quantité d'erreurs. Enfin, d'autres langages, comme C ou dans une moindre mesure C++, sont statiquement mais faiblement typés : des vérifications ont bien lieu durant la phase de compilation mais ces vérifications n'offrent aucune garantie sur la cohérence des ensembles.

Les types de donnés les plus simples offerts par OCaml sont :

- les entiers, `int`, approximation des entiers relatifs -- les éléments de `int` sont notés 1, 2, 3...
- les nombres flottants, `float`, approximation des réels -- les éléments de `float` sont notés 0.0, 0.1, -0.1, ...
- les booléens, `bool`, approximation de l'algèbre minimale de Boole -- les éléments de `bool` sont notés `true` et `false`
- les caractères, `char`, définis par le standard iso-8859-1 (alphabets européens) -- les éléments de `char` sont notés `'a'`, `'b'` ...
- les chaînes de caractères, `string`, approximation de l'ensemble des suites finies de caractères européens -- les éléments de `string` sont notés `"abc"`, `""`, `"du texte"`...
- les ficelles de caractères, `Rope.t`, approximation de l'ensemble des suites finies de caractères internationaux -- les éléments de `Rope.t` sont notés `r"abc"`, `r""`, `r"du texte"`...
- l'unité, `unit`, ensemble à un seul élément qui sert généralement d'approximation à l'ensemble vide -- pour des raisons de cohérence de notations, on considère que cet ensemble contient exactement une valeur, notée `()`.

Voyons tout de suite comment construire et manipuler des types plus évolués.

# Contraintes de types

Dans la suite de ce chapitre, nous serons quelquefois amenés à forcer OCaml à donner un type précis à une valeur, soit parce que c'est la seule manière de rendre un exemple convaincant, soit pour trouver la source d'un problème. Pour ce faire, OCaml permet de spécifier manuellement des *contraintes de types*.

Ainsi, les trois extraits suivants définisent chacun la fonction `id_int`, identique à la fonction `id` mais applicable uniquement à des entiers :

```
# let ( id_int : int -> int ) x = x ;;
val id_int : int -> int = <fun>

# let id_int ( x : int ) : int = x ;;
val id_int : int -> int = <fun>

# let id_int ( x : int ) = x ;;
val id_int : int -> int = <fun>

# let id_int = fun ( x : int ) -> x ;;
val id_int : int -> int = <fun>
```

De nouveau, les résultats de ces définitions sont rigoureusement équivalents. La première syntaxe insiste sur le fait que `id_int` est une fonction de type `int -> int`. La deuxième souligne sur le fait que `x` est lié à un entier (c'est le rôle de `( x : int )`) et que le résultat de `id_int` est encore un entier (c'est le rôle de `: int`). La troisième souligne de même le fait que `x` est lié à un entier et laisse OCaml déterminer que le résultat de la fonction `id_int` est donc lui-même un entier. La quatrième est une variante sur la troisième.

Cette syntaxe est générale : dès qu'un nom (ici aussi bien `id_int` que `x`) est lié à une valeur, que ce soit à l'aide de `let`, `let...in`, de `fun` ou du filtrage par motifs (que nous rencontrerons dans ce chapitre), il est possible de forcer le type associé à ce nom, en ajoutant une annotation `( le_nom : le_type )`. Comme d'habitude dans OCaml, les parenthèses ne sont pas systématiquement nécessaires.

Bien entendu, en introduisant des annotations de types, nous pouvons ajouter des erreurs de types dans une fonction qui, autrement, compilerait. Ainsi, si l'on demande à notre fonction de prendre un argument qui doit être simultanément flottant et entier, OCaml se plaindra d'une contradiction :

```
# let ( id_wrong : float -> float ) ( x : int ) = x ;;
                                       ^
This pattern matches values of type int
but is here used to match values of type float
```

Notez que le message d'erreur n'est pas exactement celui auquel OCaml nous a habitué en cas d'erreur de types. C'est que, ici, nous ne somme pas en train d'utiliser `x` mais de lier `x`. Du point de vue d'OCaml, il s'agit d'un *motif* ("pattern") et non pas d'une expression. À l'inverse, si l'erreur avait eu lieu dans le corps de la fonction, nous aurions retrouvé un message familier :

```
# let ( id_wrong_2 : int -> float ) ( x : int ) = x ;;
                                                  ^
This expression has type int but is here used with type float
```

Tout au long de ce chapitre, nous emploierons fréquemment les contraintes de types pour mettre en évidence des comportements du système de types d'OCaml. Au cours du développement de fonctions complexes, il arrive aussi régulièrement au programmeur d'insérer des contraintes de types le temps de comprendre pourquoi OCaml donne à une expression un type qui ne correspond pas à ce que le développeur avait prévu. Enfin, dans certains cas, heureusement rares, des annotations de types sont nécessaires pour demander à

OCaml de prendre en compte le *sous-typage*. Nous y reviendrons en temps et en heure.

# Liaison de types

De la même manière que nous avons utilisé `let` pour nommer globalement une valeur, nous pouvons employer `type` pour nommer globalement un type. Ainsi,

```
# type fonction_des_entiers_vers_les_entiers = int -> int ;;
type fonction_des_entiers_vers_les_entiers = int -> int
```

lie le nom de type `fonction_des_entiers_vers_les_entiers` à l'expression de type `int -> int`. Pour toute la suite du programme, nous pourrons employer `fonction_des_entiers_vers_les_entiers` comme synonyme exact de `int -> int`.

```
# let ( f : fonction_des_entiers_vers_les_entiers ) x = x ;;
val f : fonction_des_entiers_vers_les_entiers = <fun>

# f 5 ;;
- : int = 5
```

Avec quelques détours, nous pouvons vérifier que OCaml ne fait aucune distinction entre `fonction_des_entiers_vers_les_entiers` et `int > int` :

```
# let ( g : int -> int ) x = x ;;
val g : int -> int = <fun>

# let (teste_egalite_des_types : 'a -> 'a -> unit) x y = () ;;
val teste_egalite_des_types : 'a -> 'a -> unit = <fun>

# teste_egalite_des_types f g ;;
- : unit = ()
```

Dans ce qui précède, `teste_egalite_des_types` est une fonction dont les deux arguments doivent être de même type. Comme cette fonction accepte comme arguments `f` et `g`, nous pouvons en déduire que ces deux valeurs sont de même type `int > int`.

> **Note** Un type n'est pas une valeur. Il n'est donc pas possible d'écrire des fonctions qui prennent en argument un type ou produisent comme résultat un type. Si vous avez réellement besoin de ce genre de fonctionnalités, vous aurez besoin d'employer des *foncteurs*, une notion plus puissante que les fonctions mais aussi quelque peu plus difficile à utiliser, et que nous aborderons dans un chapitre ultérieur.

# Sommes/Variants

## Les sommes comme énumérations

Les *types sommes* ou *types variants* d'OCaml représentent une approximation de l'*union* mathématique ou, plus précisément, de l'*union disjointe*. Sous leur forme la plus simple, les types sommes peuvent être utilisés de la même manière que les énumérations de C ou de Pascal. Ainsi, on écrira :

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

Cette définition déclare quatre *constructeurs*, `Trefle`, `Pique`, `Coeur` et `Carreau`, qui à eux quatre constituent l'ensemble des éléments de type `couleur_de_carte`:

```
# Trefle ;;
- : couleur_de_carte = Trefle

# Trefle = Pique  ;;
- : bool = false

# let ( x : couleur_de_carte ) = true ;;
                                 ^^^^
This expression has type bool but is here used with type couleur_de_carte
```

Pour manipuler une valeur de type `couleur_de_carte`, nous pourrons la comparer à l'aide d'une chaîne de `if...then...else` ou, de manière plus générale, employer le *filtrage par motifs*. Ce dernier permet de définir une fonction ou une expression dont le résultat sera déterminé par la structure d'une valeur.

Ainsi, pour calculer le nom d'une couleur, on écrira au choix :

```
# let string_of_couleur_de_carte c =
    if c = Trefle       then "trefle"
    else if c = Pique   then "pique"
    else if c = Coeur   then "coeur"
    else                     "carreau" ;;
val string_of_couleur_de_carte : couleur_de_carte -> string = <fun>

# let string_of_couleur_de_carte c = match c with
    | Trefle -> "trefle"
    | Pique  -> "pique"
    | Coeur  -> "coeur"
    | Carreau-> "carreau" ;;
val string_of_couleur_de_carte : couleur_de_carte -> string = <fun>

# let string_of_couleur_de_carte = function
    | Trefle -> "trefle"
    | Pique  -> "pique"
    | Coeur  -> "coeur"
    | Carreau-> "carreau" ;;
val string_of_couleur_de_carte : couleur_de_carte -> string = <fun>
```

La première version de la fonction `string_of_couleur_de_carte` emploie le mécanisme de contrôle `if...then...else` que nous avons déjà vu et ne devrait surprendre personne. La deuxième version met l'accent sur un contrôle de flux, de la même manière que le `switch` de C ou Java, et s'énonce "la valeur de `string_of_couleur_de_carte c` dépend de la structure de `c`: si `c` est le constructeur `Trefle`, cette valeur est `"trefle"`, si `c` est le constructeur `Pique`, cette valeur est `"pique"`, si `c` est le constructeur `Coeur`, cette valeur est `"coeur"`, si `c` est le constructeur `Carreau`, cette valeur est `"carreau"`." La troisième syntaxe, plus mathématique, est une définition de fonction par cas, qui pourrait s'énoncer "`string_of_couleur` est la fonction qui à `Trefle` associe `"trefle"`, à `Pique` associe `"pique"`, à `Coeur` associe `"coeur"` et à `Carreau` associe `"carreau"`."

Les trois versions de cette fonction produiront exactement les mêmes résultats. La deuxième et la troisième sont strictement équivalentes et seront probablement plus rapides que la premières. De plus, le mécanisme de *filtrage par motifs* employé dans ces deux dernières variantes est plus lisible, plus sûr et, comme nous le verrons bientôt, bien plus puissant qu'une simple chaine de `if...then...else`.

Pourquoi plus sûr ? Parce que, comme nous pouvons le constater, OCaml est capable de vérifier que nous n'avons oublié aucun cas :

```
# val string_of_couleur_de_carte_oubli = function
                                         ^^^^^^^^
    | Trefle -> "trefle"
    ^^^^^^^^^^^^^^^^^^^
    | Pique  -> "pique"
    ^^^^^^^^^^^^^^^^^^^
    | Coeur  -> "coeur" ;;
    ^^^^^^^^^^^^^^^^^^^
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Carreau
val string_of_couleur_de_carte_oubli : couleur_de_carte -> string = <fun>
```

OCaml nous répond ici que nous avons oublié dans notre filtrage par motifs (en anglais, "pattern-matching"), de gérer une valeur telle que `Carreau`. Ceci est considéré comme un avertissement ("warning") et non pas une erreur, car nous pourrions l'avoir fait exprès en sachant pertinemment que la fonction ne sera jamais appliquée à une `couleur_de_carte` `Carreau`. (La lettre "P" désigne le nom de l'avertissement, de manière à pouvoir activer ou désactiver les avertissements lors de la compilation.)

Après une telle définition, nous pouvons manipuler `string_of_couleur_de_carte_oubli`, à nos risques et périls :

```
# string_of_couleur_de_carte_oubli Trefle ;;
- : string = "trefle"

# string_of_couleur_de_carte_oubli Carreau ;;
Exception: Match_failure ("", 30, -72).
```

Appliquer `string_of_couleur_de_carte_oubli` au constructeur `Trefle` a produit le résultat escompté alors que l'appliquer à `Carreau` a provoqué une erreur `Match_failure` (erreur au cours du filtrage). Cette erreur est peu lisible lorsque nous utilisons OCaml en mode ligne de commande mais pourra s'avérer bien plus utile pour déboguer du code lorsque nous manipulerons OCaml en tant que langage compilé.

> **Note** L'exception `Match_failure` est le plus proche équivalent OCaml au `NullPointerException` de Java. On la rencontre bien moins souvent car le compilateur OCaml vous avertira toujours lorsque vous définissez une fonction qui peut déclencher cette exception.

Du point de vue d'OCaml, les nombres `1`, `2`, `3` ... sont eux aussi autant de constructeurs du type `int`, qui aurait pu être défini sous la forme

```
# type int = 0 | 1 | 2 | (* .... *) | -1 | -2 | (* ... *) ;;
```

Une conséquence de cette conception des nombres est que nous pouvons procéder à des filtrages par motifs

sur les entiers :

```
# let string_of_nombre_francais = function
   | 1 -> "Un"
   | 2 -> "Deux"
   | 3 -> "Trois"
   | 4 -> "Quatre"
   | 5 -> "Cinq"
   | 6 -> "Six"
   | 7 -> "Sept"
   | 8 -> "Huit"
   | 9 -> "Neuf"
   | 0 -> "Zero"
   | n -> string_of_int n ;;
val string_of_nombre_francais : int -> string = <fun>
```

Les dix premiers cas sont comparables à ce que nous avons déjà rencontré. Le dernier est plus surprenant, car il introduit une variable `n`, qui est liée à la valeur que nous sommes en train de filtrer. En d'autres termes, cette fonction peut s'énoncer "la fonction qui à 1 associe `"Un"`, à 2 associe `"Deux"`..., à 0 associe `"Zero"` et *à toute autre valeur n* associe le résultat de `string_of_int n`." Cette dernière fonction convertit un nombre entier en une représentation sous la forme d'une chaîne de caractères.

> **Note** Tout comme une succession de "if...then...else", le filtrage par motifs essaye tous les cas dans l'ordre dans lequel ils sont écrits. En d'autres termes, le motif `n`, qui peut accepter n'importe quelle valeur, n'est essayé que si aucun des motifs 1, 2, ..., 9, 0, n'accepte l'argument de la fonction.

Ainsi, on aura

```
# string_of_nombre_francais 5 ;;
- : string = "Cinq"

# string_of_nombre_francais 11 ;;
- : string = "11"
```

De la même manière que les entiers sont des constructeurs prédéfinis, les booléens sont les éléments d'un type `bool` qui aurait pu être défini sous la forme

```
# type bool = false | true ;;
```

> **Note** Les noms de constructeurs commencent par une majuscule. Les seules exceptions sont quelques types prédéfinis par OCaml, notamment les booléens (dont les constructeurs sont `true` et `false`), les entiers, les flottants...

Ainsi, si nous tentons de définir un type à l'aide de constructeurs qui commmencent par une minuscule, nous obtiendrons

```
# type somme_erreur = a | b ;;
                      ^
Error: Parse error: [str_item] or ";;" expected (in [top_phrase])
```

C'est-à-dire "Je ne comprends pas ce que vous avez écrit. À cet endroit-là d'une [top_phrase] (une phrase de

plus haut niveau, c'est-à-dire une déclaration), j'attends soit une autre [str_item] (c'est-à-dire une déclaration ou un calcul), soit un ';;'."

Nous reviendrons fréquemment sur le filtrage par motifs -- pour le moment sans le définir précisément -- car il s'agit d'un des mécanismes les plus intéressants de la programmation fonctionnelle.

## Les sommes comme conteneurs

Comme mentionné plus haut, les types *somme* approximent la notion mathématique d'union. Jusqu'à présent, nous avons n'avons guère vu l'union se manifester. Ceci est du au fait que nos constructeurs `Trefle`, `Pique`, `Coeur` et `Carreau` ne prennent aucun argument -- et ne désignent donc chacun, en pratique, qu'un ensemble réduit à un singleton. Voyons tout de suite un type un peu plus complexe, qui tirera parti d'ensembles plus vastes :

```
# type carte_de_tarot =
  | Atout    of int
  | Roi      of couleur_de_carte
  | Dame     of couleur_de_carte
  | Cavalier of couleur_de_carte
  | Valet    of couleur_de_carte
  | As       of couleur_de_carte
  | Nombre   of int * couleur_de_carte ;;
type carte_de_tarot =
  | Atout of int
  | Roi of couleur_de_carte
  | Dame of couleur_de_carte
  | Cavalier of couleur_de_carte
  | Valet of couleur_de_carte
  | As of couleur_de_carte
  | Nombre of int * couleur_de_carte
```

La déclaration (et la réponse) s'énoncent "le type `carte_de_tarot` est constitué de l'ensemble des valeurs `Atout i` où `i` est un entier, de l'ensemble des valeurs `Roi c` où `c` est une `couleur_de_carte`, de l'ensemble des valeurs `Dame c` où `c` est une `couleur_de_carte`, de l'ensemble des valeurs `Cavalier c` où `c` est une `couleur_de_carte`, de l'ensemble `Valet c` où `c` est une `couleur_de_carte`, de l'ensemble `As c` où `c` est une `couleur_de_carte` et enfin de l'ensemble `Nombre i c` où `i` est un entier et `c` est une `couleur_de_carte`."

Après cette définition, nous pouvons employer des valeurs de type `carte_de_tarot`:

```
# Atout 5 ;;
- : carte_de_tarot = Atout 5

# Cavalier Trefle ;;
- : carte_de_tarot = Cavalier Trefle

# Nombre (5, Trefle) ;;
- : carte_de_tarot = Nombre 5 Trefle
```

> **Vocabulaire** Les *constructeurs* de `carte_de_tarot` sont `Atout _`, `Roi _`, `Dame _`, `Cavalier _`, `Valet _`, `As _` et `Nombre(_, _)`.

Notons que cette représentation laisse un élément de fragilité, puisque nous pouvons très bien écrire :

```
# Atout 23;;
- : carte_de_tarot = Atout 23

# Nombre (-1, Trefle);;
- : carte_de_tarot =  Nombre (-1, Trefle)
```

En d'autres termes, nous ne pouvons pas encore manipuler les cartes en toute sécurité. Avant cela, nous devrons nous assurer que `Nombre` ne peut être associés qu'à des entiers compris entre 1 et 9 et `Atout` à des entiers entre 0 et 21. Nous verrons comment garantir ce genre de propriétés lorsque nous parlerons de modules.

Nous pouvons maintenant manipuler les `carte_de_tarots` à l'aide du filtrage par motifs. Ainsi, pour afficher le nom complet d'une carte, nous pourrons définir la fonction

```
# let string_of_carte_de_tarot carte = match carte with
  | Atout    i    -> string_of_int i ^ " d'atout"
  | Roi      c    -> "Roi de "       ^ ( string_of_couleur_de_carte c )
  | Dame     c    -> "Dame de "      ^ ( string_of_couleur_de_carte c )
  | Cavalier c    -> "Cavalier de "  ^ ( string_of_couleur_de_carte c )
  | Valet    c    -> "Valet de "     ^ ( string_of_couleur_de_carte c )
  | As       c    -> "As de "        ^ ( string_of_couleur_de_carte c )
  | Nombre (i, c) -> string_of_chiffre_francais i ^ " de " ^ ( string_of_couleur_de_carte
val string_of_carte_de_tarot : carte_de_tarot -> string = <fun>

# let string_of_carte_de_tarot = function
  | Atout    i    -> string_of_int i ^ " d'atout"
  | Roi      c    -> "Roi de "       ^ ( string_of_couleur_de_carte c )
  | Dame     c    -> "Dame de "      ^ ( string_of_couleur_de_carte c )
  | Cavalier c    -> "Cavalier de "  ^ ( string_of_couleur_de_carte c )
  | Valet    c    -> "Valet de "     ^ ( string_of_couleur_de_carte c )
  | As       c    -> "As de "        ^ ( string_of_couleur_de_carte c )
  | Nombre (i, c) -> string_of_chiffre_francais i ^ " de " ^ ( string_of_couleur_de_carte
val string_of_carte_de_tarot : carte_de_tarot -> string = <fun>
```

Les deux écritures de `string_of_carte_de_tarot` sont strictement équivalentes. Dans ce qui précède, nous avons utilisé la fonction `string_of_chiffre_francais` définie plus haut, la fonction `string_of_int` vue plus haut et l'opérateur `^` de *concaténation* de chaînes de caractères. La première syntaxe, de nouveau, insiste sur le fait que le comportement de la fonction est différent selon la structure de `carte` : si `carte` s'écrit `Atout i`, la fonction se comporte comme `string_of_int i ^ " d'atout"` sinon, si `carte>` s'écrit `Roi c`, la fonction se comporte comme `"Roi de " ^ ( string_of_couleur_de_carte c )` ... La deuxième syntaxe, plus mathématique, se prononcerait "s'il existe un entier `i` tel que l'argument s'écrive `Atout i`, le résultat est donné par ..." ou encore, avec un point de vue ensembliste, "`string_of_carte_de_tarot` est la fonction qui à tout élément `i` de l'ensemble `Atout` associe `string_of_int i ^ " d'atout"`, à tout élément `c` de l'ensemble `Roi` associe..." ou encore, cette fois avec un point de vue plus informatique, "`string_of_carte_de_tarot` est la fonction qui à tout entier `i` étiqueté `Roi` associe ..."

De nouveau, lors du filtrage par motifs, nous avons *lié* des variables (ici `i` ou/et `c` selon les cas) à des valeurs contenues dans `carte`.

Une fois cette fonction définie, nous pouvons écrire

```
# string_of_carte_de_tarot ( Atout 5 ) ;;
- : string = "5 d'atout"
```

```
# string_of_carte_de_tarot ( Valet Pique ) ;;
- : string = "Valet de pique"

# string_of_carte_de_tarot ( Nombre (5, Trefle ));;
- : string = "Cinq de trefle"
```

Notons que nous n'avons pas à nous arrêter en si bon chemin : nous pouvons très bien utiliser le filtrage par motifs pour chercher des motifs de complexité arbitraire. Ainsi, considérons l'extrait suivant :

```
# let est_pique = function
   | As         Pique  -> true
   | Roi        Pique  -> true
   | Dame       Pique  -> true
   | Cavalier   Pique  -> true
   | Nombre (_, Pique) -> true
   | _                 -> false ;;
val est_pique : carte_de_tarot -> bool = <fun>
```

Tout comme `As c` était un motif, qui acceptait n'importe quel `As` et liait `c` à sa couleur, `As Pique` est un motif plus précis, qui accepte uniquement la valeur `As Pique`. Le motif `Nombre _ Pique`, quant à lui, est équivalent au motif `Nombre c Pique` -- le symbole _, qui se prononce "don't care", est un motif qui accepte n'importe quelle valeur mais ne procède à aucune liaison. En d'autres termes, on emploiera _ pour signifier que nous ne somme pas intéressés par un valeur correspondante. C'est ainsi, le dernier motif _ accepte toutes les valeurs. Comme dans l'exemple `string_of_nombre_francais`, nous tirons parti du fait que les motifs sont essayés dans l'ordre dans lequel ils sont écrits, si bien que _ tient ici un rôle similaire à `else` (dans le cadre d'un `if...then...else`) ou `default:` (dans le cadre d'un `switch(...) case: ... default: ...`.

Nous verrons plus loin comment fusionner plusieurs motifs et réduire la taille de la fonction `est_pique`

**Exercices**

1. Définissez un type de données `couleur_ou_erreur` qui pourra contenir soit une `couleur_de_carte` (constructeur `Couleur`), soit un message d'erreur (une chaîne de caractères quelconque, avec le constructeur `Erreur`).
2. Définissez une fonction `get_couleur : carte_de_tarot -> couleur_ou_erreur`, qui renverra la couleur de la carte qu'on lui passe en argument. Comme les Atouts n'ont pas de couleur, on utilisera `Erreur` pour répondre dans le cas où la carte serait un Atout.
3. La fonction suivante ne produit pas le résultat escompté. Pourquoi ? Le type de cette fonction pourra vous aider à trouver l'erreur.

```
# let est_couleur couleur = function
  | As       couleur    -> true
  | Roi      couleur    -> true
  | Dame     couleur    -> true
  | Cavalier couleur    -> true
  | Nombre (_, couleur) -> true
  | _                   -> false ;;
val est_couleur : 'a -> carte_de_tarot -> bool = <fun>
```

1. Définissez une fonction `est_couleur : couleur_de_carte -> carte_de_tarot -> bool`, qui détermine si une carte porte bien une couleur donnée. Vous pourrez utiliser = pour comparer les couleurs. Renvoyez `false` si la carte est un Atout.

## Filtrage par motifs

### Match

En OCaml, le filtrage par motifs est une manière de définir une expression en fonction de la structure d'une valeur, c'est-à-dire de la manière dont la valeur est *construite* -- et ce terme cache les constructeurs que nous avons rencontrés tout au long de cette section.

Sous sa forme la plus générale, le filtrage par motifs selon la structure de la valeur `v` s'écrit

```
match v with
| p1 -> e1
| p2 -> e2
(* ... *)
| pn -> en
```

Cette construction se lit "si `v` a la structure `p1`, évaluer l'expression `e1`, dans le cas contraire, si `v` a la structure `p2`, évaluer l'expression `e2` ... dans le cas contraire, si `v` a la structure `pn`, évaluer l'expression `en`." Si `v` n'a aucune de ces structures, il s'agit d'une erreur fatale d'exécution.

Notons que la valeur `v` utilisée ici peut être remplacée par n'importe quelle expression `expr`, auquel cas c'est le résultat de l'évaluation de `expr` qui sera utilisé à la place de `v`.

> **Vocabulaire** `p1`, `p2` ... `pn` sont des *motifs*.

Nous avons déjà vu de nombreux exemples de motifs. Ainsi, dans

```
# let string_of_couleur_de_carte c = match c with
    | Trefle -> "trefle"
    | Pique  -> "pique"
    | Coeur  -> "coeur"
    | Carreau-> "carreau" ;;
```

les constructeurs `Trefle`, `Pique`, `Coeur`, `Carreau` sont employés en tant que motifs. De même, dans

```
# let string_of_nombre_francais x = match x with
   | 1 -> "Un"
   | 2 -> "Deux"
   | 3 -> "Trois"
   | 4 -> "Quatre"
   | 5 -> "Cinq"
   | 6 -> "Six"
   | 7 -> "Sept"
   | 8 -> "Huit"
   | 9 -> "Neuf"
   | 0 -> "Zero"
   | n -> string_of_int n ;;
```

les constructeurs sans arguments `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9` et `0` et le nom de variable `n` sont utilisés en tant que motifs. De manière plus compliquée, dans

```
# let string_of_carte_de_tarot carte = match carte with
  | Atout    i    -> string_of_int i ^ " d'atout"
```

```
  | Roi      c    -> "Roi de "       ^ ( string_of_couleur_de_carte c )
  | Dame     c    -> "Dame de "      ^ ( string_of_couleur_de_carte c )
  | Cavalier c    -> "Cavalier de "  ^ ( string_of_couleur_de_carte c )
  | Valet    c    -> "Valet de "     ^ ( string_of_couleur_de_carte c )
  | As       c    -> "As de "        ^ ( string_of_couleur_de_carte c )
  | Nombre (i, c) -> string_of_chiffre_francais i ^ " de " ^ ( string_of_couleur_de_carte
```

le motif `Atout i` est composé de la variable `i` placée dans le constructeur à un argument `Atout`, le motif `Roi c` est composé de la variable `c` placée dans le constructeur à un argument `Roi`... et le motif `Nombre(i, c)` est composé de la variable `c` et de la variable `i` placés dans le constructeur à deux arguments `Nombre`.

Voyons tout de suite la définition exacte d'un motif :

- une variable `v` est un motif, qui *accepte* n'importe quelle valeur -- lorsque la valeur est acceptée, `v` est liée à cette valeur
- si `A` est un constructeur qui n'accepte aucun argument, alors `A` est un motif, qui accepte exactement la valeur `A`
- si `A` est un constructeur qui accepte *n* arguments, si `p1`, `p2` .. `pn` sont *n* motifs, alors `A(p1, p2, ..., pn)` est un motif, qui accepte tous les `A(e1, e2, ..., en)` tels que `p1` accepte `e1`, `p2` accepte `e2` ... `pn` accepte `en`
- si `p1` et `p2` sont des motifs, `p1 | p2` est un motif, prononcé "`p1` ou `p2`" et qui accepte toutes les expressions acceptées soit par `p1`, soit par `p2`.

Ainsi, dans la définition de `string_of_carte_de_tarot` :

- OCaml commence par vérifier si `carte` est défini à l'aide du constructeur `Atout`
  - Le cas échéant, OCaml vérifie si `carte` s'écrit bien `Atout i` pour une certaine valeur de `i` -- de fait, comme `x` est défini à l'aide du constructeur `Atout`, c'est effectivement toujours le cas
  - Le cas échéant, OCaml lie `i` à la valeur nécessaire pour que `carte` s'écrive `Atout i` et donc l'expression `string_of_int i ^ " d'atout"`
- si `x` ne s'écrivait pas à l'aide du constructeur `Atout`, OCaml vérifie si `x` peut s'écrire à l'aide du constructeur `Roi`
  - Le cas échéant, OCaml vérifie si `carte` s'écrit bien `Roi c` pour une certaine valeur de `c` - de fait, comme `x` est défini à l'aide du constructeur `Roi`, c'est effectivement toujours le cas
  - Le cas échéant, OCaml lie `c` à la valeur nécessaire pour que `carte` s'écrive `Roi c` et évalue l'expression

```
"Roi de " ^ ( string_of_couleur_de_carte c )
```

- ...
- Si `x` ne s'écrit d'aucune des manières précédentes, OCaml vérifie si `carte` peut s'écrire à l'aide du constructeur `Nombre`
  - Le cas échéant, OCaml vérifie si `carte` s'écrit bien `Nombre(i, c)` pour une certaine valeur de `i` et `c` - de fait, comme `carte` est défini à l'aide du constructeur `Nombre`, c'est effectivement toujours le cas
  - Le cas échéant, OCaml lie `c` et `i` aux valeurs nécessaires pour que `x` s'écrive `Nombre i c` et évalue l'expression `string_of_chiffre_francais i ^ " de " ^ ( string_of_couleur_de_carte c )`.

Insistons sur le fait qu'un motif peut contenir un motif. Ainsi, on aurait pu ajouter un premier cas plus spécifique

```
# let string_of_carte_de_tarot carte = match carte with
  | Atout    0 -> "Excuse"
  | Atout    i -> string_of_int i ^ " d'atout"
 ...
```

auquel cas, en plus de vérifier si `carte` peut s'écrire `Atout x` pour un certain `x`, OCaml aurait vérifié si la valeur de `x` était de plus acceptée par `0`.

#### Function

Dans certains des exemples que nous avons rencontrés, nous avons employé `function` au lieu de `match`. Ceci est permis par quelques raccourcis syntaxiques d'OCaml. Ainsi, considérons l'expression suivante :

```
fun x -> match x with
   | p1 -> e1
   | p2 -> e2
   ...
   | pn -> en
```

Pour peu que ce nom `x` ne soit pas utilisé dans `e1`...`en`, cette expression peut s'abréger de la manière suivante :

```
function p1 -> e1
       | p2 -> e2
       | ...
       | pn -> en
```

### Précautions et limitations

#### Variables dans les motifs

Lorsqu'un motif accepte une valeur, toutes les variables qui apparaissent dans ce motif sont liées par le filtrage. En particulier, nous aurons

```
# let x = 1 in match 5 with x -> x ;;
      ^
Warning Y: unused variable x.
- : int = 5
```

Comme le signale l'avertissement, la première définition de `x` est totalement ignorée. L'utilisation de `x` en tant que motif accepte n'importe quelle valeur -- ici `5` -- et lie cette valeur à `x`.

C'est pour cette raison qu'une même variable ne peut pas apparaître deux fois dans un motif sans `|` :

```
# match (1,2) with (x,x) -> print_int x ;;
                       ^
Variable x is bound several times in this matching
```

En effet, pour que cette expression ait un sens, `x` devrait valoir à la fois la valeur `1` et la valeur `2`.

À l'inverse, dans un motif de la forme `p1 | p2`, les variables qui apparaissent dans `p1` doivent être exactement les mêmes que celles qui apparaissent dans `p2`.

```
# function
    Atout i | Roi j  -> i  ;;
```

```
           ^^^^^^^^^^
Variable i must occur on both sides of this | pattern
```

### À titre de comparaison

Dans Java ou tout autre langage par objets statiquement typé, l'équivalent du type somme `carte_de_tarot` serait une hiérarchie de classes doté d'un visiteur (le visiteur est l'un des patrons de conception les plus importants pour la programmation orientée objets) :

```
 1 public interface CarteDeTarotVisitor<T>
 2 {
 3   public T visitAtout   (Atout c);
 4   public T visitRoi     (Roi   c);
 5   public T visitDame    (Dame   c);
 6   public T visitCavalier(Cavalier  c);
 7   public T visitValet   (Valet   c);
 8   public T visitAs      (As   c);
 9   public T visitNombre  (Nombre c);
10 }
11
12 public abstract class CarteDeTarot
13 {
14   protected CarteDeTarot()
15   {
16   }
17
18   public<T> T accept(CarteDeTarotVisitor<T> v);
19 }
20
21
22 public class Atout extends CarteDeTarot
23 {
24   public final int valeur;
25   public Atout(int valeur)
26   {
27     assert(0 <= valeur && valeur <= 21 );
28     this.valeur = valeur;
29   }
30
31   public<T> T accept(CarteDeTarotVisitor<T> v);
32   {
33      return v.visitAtout(this);
34   }
35 }
36
37 public class Roi extends CarteDeTarot
38 {
39   public final CouleurDeCarte couleur;
40   public Roi(CouleurDeCarte couleur)
41   {
42     assert(couleur != null );
43     this.couleur = couleur;
44   }
45
46   public<T> T accept(CarteDeTarotVisitor<T> v);
47   {
48      return v.visitRoi(this);
49   }
50 }
51
52 public class Dame extends CarteDeTarot
53 {
```

```
    54   public final CouleurDeCarte couleur;
 55   public Dame(CouleurDeCarte couleur)
 56   {
 57     assert(couleur != null );
 58     this.couleur = couleur;
 59   }
 60
 61   public<T> T accept(CarteDeTarotVisitor<T> v);
 62   {
 63      return v.visitDame(this);
 64   }
 65 }
 66
 67 public class Cavalier extends CarteDeTarot
 68 {
 69   public final CouleurDeCarte couleur;
 70   public Cavalier(CouleurDeCarte couleur)
 71   {
 72     assert(couleur != null );
 73     this.couleur = couleur;
 74   }
 75
 76   public<T> T accept(CarteDeTarotVisitor<T> v);
 77   {
 78      return v.visitCavalier(this);
 79   }
 80 }
 81
 82 public class Valet extends CarteDeTarot
 83 {
 84   public final CouleurDeCarte couleur;
 85   public Valet(CouleurDeCarte couleur)
 86   {
 87     assert(couleur != null );
 88     this.couleur = couleur;
 89   }
 90
 91   public<T> T accept(CarteDeTarotVisitor<T> v);
 92   {
 93      return v.visitValet(this);
 94   }
 95 }
 96
 97 public class As extends CarteDeTarot
 98 {
 99   public final CouleurDeCarte couleur;
100   public As(CouleurDeCarte couleur)
101   {
102     assert(couleur != null );
103     this.couleur = couleur;
104   }
105
106   public<T> T accept(CarteDeTarotVisitor<T> v);
107   {
108      return v.visitAs(this);
109   }
110 }
111
112 public class Nombre extends CarteDeTarot
113 {
114   public final CouleurDeCarte couleur;
115   public final int            valeur;
116   public Nombre(CouleurDeCarte couleur, int valeur)
117   {
118     assert(couleur != null );
119     assert(2 <= valeur && valeur <= 10 );
```

```
   120     this.couleur = couleur;
121     this.valeur  = valeur;
122   }
123
124   public<T> T accept(CarteDeTarotVisitor<T> v);
125   {
126      return v.visitNombre(this);
127   }
128 }
```

À ce stade, le seul avantage de la version Java sur notre version OCaml est que cette première vérifie la valeur de la carte, pour garantir qu'un Atout sera toujours entre 0 et 21 et une carte numérotée entre 2 et 10. Nous verrons bientôt qu'il est aussi simple de faire de même en OCaml, en introduisant l'équivalent des constructeurs Java -- en OCaml, ce sont des fonctions ordinaires. Dans les langages orientés objet, l'utilisation du patron de conception Visiteur est importante pour pouvoir implanter par la suite des fonctions telles que `string_of_carte_de_tarot` sans modifier la hiérarchie originelle. Notons que, sur ce dernier point, il ne s'agit pas d'un avantage décisif d'OCaml mais d'une manière différente de concevoir la programmation : dans les langages par objets, le comportement d'un programme se décrit essentiellement sous la forme d'un ensemble de classes, où chaque classe est caractérisée par des propriétés (les champs) et des instructions auxquelles elle sait répondre (les méthodes). À l'inverse, dans les langages fonctionnels, les comportement d'un programme se décrit essentiellement sous la forme d'un ensemble de conteneurs de données (les structures), sur lesquelles agissent des transformations (les fonctions). Un problème dans lequel une fonction devra agir sur plusieurs structures se traduira donc plus naturellement en programmation fonctionnelle, tandis qu'un problème mieux caractérisé par des interactions entre objets se traduira plus naturellement en programmation orientée objets. Nous verrons plus loin comment rassembler des propriétés et instructions en modules sous OCaml et ainsi retrouver une bonne partie des bénéfices de la programmation par objets.

Une fois cette hiérarchie implantée et en supposant que l'on s'interdit de modifier la susdite hiérarchie, par exemple parce qu'elle est fournie sous la forme d'une bibliothèque de programmation, l'équivalent de la fonction `string_of_carte_de_tarot`, en supposant que nous savons déjà transformer une couleur de carte en une chaîne de caractères, serait :

```
1 public class StringOfCarteDeTarot implements CarteDeTarotVisitor<String>
2 {
3   public String visitAtout   (Atout c)
4   {
5      return c.valeur + " d'atout ";
6   }
7   public String visitRoi     (Roi   c)
8   {
9      return "Roi de "+c.couleur;
10  }
11  public String visitDame    (Dame   c)
12  {
13     return "Dame de "+c.couleur;
14  }
15  public String visitCavalier(Cavalier  c)
16  {
17     return "Cavalier de "+c.couleur;
18  }
19  public String visitValet   (Valet   c)
20  {
21     return "Valet de "+c.couleur;
22  }
23  public String visitAs      (As   c)
24  {
25     return "As de "+c.couleur;
```

```
   26   }
27   public String visitNombre  (Nombre c)
28   {
29      return Utilitaires.getChiffreFrancais ( c.valeur ) + " de " + c.couleur ;
30   }
31 }
```

## Paramètres de types

Tout comme nous avons employé des types de fonctions contenant des paramètres de types `'a`, `'b`..., nous pouvons définir de nouveaux types *paramétriques* avec ces mêmes variables. Ainsi, une variante sur le type `couleur_ou_erreur` est le type `option`, défini dans la bibliothèque standard de OCaml comme

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

Nous pouvons donc utiliser :

```
# None ;;
- : option 'a = None

# Some 5 ;;
- : option int = Some 5

# Some "blue ";;
- : option string = Some "blue"
```

Nous pouvons de même réécrire `get_couleur`

```
# let get_couleur = function
    | As       c    -> Some c
    | Roi      c    -> Some c
    | Valet    c    -> Some c
    | Dame     c    -> Some c
    | Cavalier c    -> Some c
    | Nombre (_, c) -> Some c
    | _             -> None ;;
val get_couleur : carte_de_tarot -> couleur_de_carte option = <fun>
```

Comme d'habitude, c'est OCaml qui détermine quel type doit remplacer `'a`.

La fonction `est_couleur` se réécrit alors par exemple :

```
# let est_couleur couleur carte = match get_couleur carte with
      None   -> false
    | Some c -> c = couleur ;;
val est_couleur : couleur_de_carte -> carte_de_tarot -> bool = <fun>
```

De même, il est fréquent d'utiliser des types ressemblant à

```
# type ('a, 'b) succes_ou_erreur =
```

```
    | Succes  of 'a
    | Erreur  of 'b ;;
type ('a, 'b) succes_ou_erreur =  Succes  of 'a | Erreur  of 'b
```

Un tel type permet d'écrire des fonctions qui produiront comme résultat `Succes x`, où `x` est une valeur renvoyée lorsque la fonction s'exécute correctement ou `Erreur y`, où code `y` est une valeur renvoyée lorsque la fonction rencontre un problème :

```
# let division x y =
    if y = 0 then  Erreur "Division par zero"
    else           Succes ( x / y );;
val division : int -> int -> succes_ou_erreur int string = <fun>
```

On pourrait, de même, définir le type d'un singleton, d'un couple, d'un triplet... par

```
# type empty_set           = EmptySet ;;
# type 'a singleton        = Singleton of 'a ;;
# type ('a, 'b) pair       = Pair of 'a * 'b ;;
# type ('a, 'b, 'c) triple = Triple of 'a * 'b * 'c;;
```

En fait, nous verrons bientôt qu'OCaml propose une autre syntaxe, plus concise, pour les n-uplets.

### À titre de comparaison

En Java et dans l'essentiel des langages de programmation, le type `'a option` est inutile car (presque) tous les types peuvent prendre la valeur spéciale `null`. Cette possibilité est un héritage de l'ère C, dans laquelle les références en mémoire ne sont que des entiers comme les autres et la valeur `0` est utilisée par convention pour représenter un pointeur qui ne pointe sur rien d'intéressant. L'inconvénient de cette approche est double :

- quelques types précis, fixés lors de la conception du langage, ne peuvent, pour des raisons de performances, pas prendre la valeur `null`
- comme toute valeur à l'exception de ces quelques types peut valoir `null`, le compilateur ne peut pas vérifier si l'on passe `null` à une fonction qui ne peut pas accepter ce genre de valeur. C'est au programmeur d'ajouter manuellement à chaque fonction qui ne peut pas accepter une valeur `null` une vérification sur la `null`-ité des arguments, sous peine de devoir plus tard déboguer des `NullPointerExceptions` (en Java), voire des `SEGFAULT` (dans beaucoup d'autres langages).

Par comparaison, l'approche OCaml :

- garantit l'impossibilité d'erreurs comparables au `NullPointerException`
- permet de documenter clairement quelles valeurs peuvent prendre la valeur `None` et quelles valeurs sont toujours définies
- peut s'étendre, comme le montre l'exemple `succes_ou_erreur` pour renvoyer des détails sur une éventuelle erreur
- pourrait encore s'étendre pour gérer un nombre arbitraire de possibilités, contrairement à l'alternative `null` / pas `null`.

## Sommes récursives

En OCaml, les types peuvent être définis récursivement -- et c'est fréquemment le cas des types sommes. Pour ce faire, il n'est pas nécessaire d'utiliser un mot-clé particulier. Ainsi, une liste de cartes pourra être

définie comme :

- soit la liste vide
- soit une liste non-vide, c'est-à-dire une carte suivie d'une liste de cartes (que nous appellerons la *queue*).

En OCaml, nous traduirons ceci par

```
# type liste_de_cartes =
     | ListeVide
     | ListeNonVide of (carte_de_tarot * liste_de_cartes ) ;;
type liste_de_cartes =
     | ListeVide
     | ListeNonVide of (carte_de_tarot * liste_de_cartes )
```

De fait, à l'aide des paramètres de types, on pourrait même écrire

```
# type 'a liste =
     | ListeVide
     | ListeNonVide of ( 'a * 'a liste ) ;;
type 'a liste =
     | ListeVide
     | ListeNonVide of ( 'a * 'a liste )
```

> **Note** Il existe une syntaxe plus agréable et plus lisible pour les listes, ainsi que toute une bibliothèque de fonctions sur ces listes, que nous détaillerons plus tard. Pour le moment, contentons-nous d'oublier que tout ceci est déjà défini dans OCaml.

Nous venons de définir les constructeurs `ListeVide` et `ListeNonVide(_, _)`. Pour ajouter un élément en tête d'une liste, nous pourrons écrire par exemple :

```
# let cons tete queue = ListeNonVide tete queue ;;
val cons : 'a -> 'a liste -> 'a liste = <fun>
```

> **Note** Le nom "cons" (le "s" se prononce) est couramment utilisé en programmation fonctionnelle. Il désigne l'ajout d'un élément en tête de liste.

En appliquant plusieurs fois les constructeurs ou `cons`, nous pouvons écrire de la manière suivante la liste qui contient 1, 2, 3, 4 et 5 :

```
# let une_liste = ListeNonVide (1, ListeNonVide (2, ListeNonVide (3, ListeNonVide (4, Lis
val une_liste : int liste =
  ListeNonVide ( 1,
   ListeNonVide ( 2,
     ListeNonVide ( 3,  (ListeNonVide (4,  ListeNonVide (5, ListeVide))))

# let une_liste = cons 1 (cons 2 (cons 3 (cons 4 (cons 5 ListeVide)))) ;;
val une_liste : int liste =
  ListeNonVide ( 1,
   ListeNonVide ( 2,
     ListeNonVide ( 3,  (ListeNonVide (4,  ListeNonVide (5, ListeVide))))
```

> **Note** Ceci est une liste (chaînée) au sens de la programmation fonctionnelle. Contrairement aux listes impératives, on s'interdit de *modifier* la liste en supprimant ou en insérant de nouveaux éléments. Au lieu de cela, insérer ou supprimer un élément d'une liste seront des opérations de

transformation, qui construiront de nouvelles listes en lieu et place des anciennes. Bien entendu, en OCaml, il est aussi possible de définir des listes impératives. D'expérience, ces dernières sont utilisées bien moins fréquemment.

Nous pouvons aisément déterminer si une liste est vide :

```
# let est_vide = function
   | ListeVide -> true
   | _         -> false ;;
val est_vide : 'a liste -> bool = <fun>

# let est_vide l = l = ListeVide ;;
val est_vide : 'a liste -> bool = <fun>

# let est_vide = ( = ) ListeVide ;;
val est_vide : '_a liste -> bool = <fun>
```

La première formulation utilise le filtrage par motifs. La deuxième compare `l` avec `ListeVide` pour produire le même résultat. Pour le moment, laissons de côté la troisième formulation et son surprenant `'_a` -- sachez juste que si vous rencontrez un tel type, vous êtes tombés sur une limitation du système de types actuel d'OCaml et que pour vous en tirer, la bonne méthode est d'ajouter l'argument `l` à votre fonction et de revenir à la deuxième formulation.

Dans tous les cas, nous aurons

```
# est_vide une_liste ;;
- : bool = false
```

Nous pouvons de même aisément compter le nombre de cartes dans une liste de cartes. Pour ce faire, nous pourrons appliquer la définition par récurrence suivante :

- la liste vide contient 0 cartes
- dans une liste non vide, nous pouvons commencer par calculer le nombre de cartes de la queue et lui ajouter 1.

Soit, en OCaml, récursivement et à l'aide du filtrage par motifs

```
# let rec longueur l = match l with
    | ListeVide               -> 0
    | ListeNonVide (_, queue) -> longueur queue + 1 ;;
val longueur : 'a liste -> int = <fun>

# let rec longueur = function
    | ListeVide               -> 0
    | ListeNonVide (_, queue) -> longueur queue + 1 ] ;
val longueur : 'a liste -> int = <fun>
```

De nouveau, les deux formulations sont équivalentes. Notons que l'argument est de type `'a liste` : comme nous n'avons jamais utilisé le contenu de la liste, leur type n'est pas contraint -- et cette fonction peut traiter des listes de n'importe quel type.

Bien entendu, de la même manière que nous avons défini des fonctions abstraites sur les entiers, il est naturel de définir des fonctions abstraites sur les listes. Ainsi, en généralisant la valeur `0` (sous la forme d'un argument `init`) et l'opération + (sous la forme d'un argument `f`), nous obtenons, selon l'ordre des

opérations :

```
# let operation_sur_liste_droite f init =
   let rec aux = function
     | ListeVide               -> init
     | ListeNonVide (c, queue) -> ( f c ( aux queue ) )
   in aux;;
val operation_sur_liste_droite : ('a -> 'b -> 'b) -> 'b -> 'a liste -> 'b =
  <fun>
```

ou

```
# let operation_sur_liste_gauche f init =
   let rec aux acc = function
     | ListeVide               -> acc
     | ListeNonVide (c, queue) -> aux ( f c acc ) queue
   in aux init;;
val operation_sur_liste_gauche : ('a -> 'b -> 'b) -> 'b -> 'a liste -> 'b =
  <fun>
```

Une fois ceci fait, nous pouvons demander à OCaml d'appliquer une transformation à la liste `une_liste`, en passant chaque élément par `cons` et en initialisant le résultat à `ListeVide` :

```
# operation_sur_liste_droite cons ListeVide une_liste ;
- : int liste =
  ListeNonVide ( 1,
   ListeNonVide ( 2,
     ListeNonVide ( 3,  (ListeNonVide (4,  ListeNonVide (5, ListeVide))))

# operation_sur_liste_gauche cons ListeVide une_liste ;
- : liste int =
  ListeNonVide ( 5,
   ListeNonVide ( 4,
     ListeNonVide ( 3,  (ListeNonVide (2,  ListeNonVide (1, ListeVide))))
```

On pourrait définir de même des fonctions de filtre, de transformations de listes...

### ... et listes

Comme mentionné plus haut, il n'est en fait pas nécessaire de définir nous-même une notion de liste, puisque OCaml dispose de cette notion et d'une syntaxe simplifiée pour les manipuler.

Conceptuellement, l'ensemble des listes d'éléments de type `'a`, noté `'a list` est un type somme doté des constructeurs

- `[]`, la liste vide
- `_ :: _` , la liste non vide -- on notera `h :: t` la liste dont le premier élément est `h` et le reste des éléments est une liste `t`.

> **Vocabulaire** Le constructeur `[]` est appelé "nil". Le constructeur `_ :: _` est appelé "cons".

Ainsi, nous aurons

```
# [] ;;
```

```
- : 'a list = []

# 1 :: 2 :: 3 :: 4 :: 5 :: [] ;;
- : int list = [1; 2; 3; 4; 5]
```

Comme le montre la réponse d'OCaml, il existe en fait une notation plus simple :

```
# [1; 2; 3; 4; 5];;
- : int list = [1; 2; 3; 4; 5]
```

Ces notations peuvent être utilisées aussi bien pour créer de nouvelles listes que pour le filtrage par motifs. Ainsi, nous pouvons définir les fonctions `est_vide`, `tete`, `queue` comme

```
# let est_vide = function
   |  []  -> true
   |  _   -> false ;;
val est_vide : 'a list -> bool = <fun>


# let tete = function  h :: _  -> h ;;
             ^^^^^^^^^^^^^^^^^^^^
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val tete : 'a list -> 'a = <fun>

# let queue = function _ :: t -> t ;;
              ^^^^^^^^^^^^^^^^^^^
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val queue : 'a list -> 'a = <fun>
```

Ces deux dernières définitions sont accompagnées d'un avertissement -- effectivement, elle n'ont aucun sens lorsque la liste est vide.

> **Note** La fonction `tete` est déjà définie dans la bibliothèque standard sous le nom `List.hd` et la fonction queue sous le nom `List.tl`. De nombreuses autres fonctions sur les listes sont définies dans la bibliothèque standard. Systématiquement,

pour définir une fonction qui compte le nombre d'éléments dans une liste, nous pourrons écrire

```
# let rec longueur = function
   |  [] -> 0
   |  _ :: t -> longueur t + 1 ;;
```

Dans l'extrait qui précède, le motif `[]` désigne la liste vide. De même, le motif `_ :: t` consiste en le constructeur cons appliqué aux deux motifs `_` (en position de la tête de liste) et `t` (en position de la queue de liste). En d'autres termes, ce motif désigne une liste contenant au moins un élément (qui ne nous intéresse pas) et dont la queue de liste est liée au nom `t`.

De même, pour écrire une fonction qui filtrera une liste et ne gardera qu'un élément sur deux, nous pourrons écrire

```
# let rec un_sur_deux = function
   | []           -> []
   | _ :: []      -> []
   | h :: _ :: t  -> h :: (un_sur_deux t) ;;
val un_sur_deux : 'a list -> 'a list = <fun>
```

Cette définition s'énonce "la fonction `un_sur_deux` se définit récursivement par

- `un_sur_deux` appliqué à la liste vide produit la liste vide
- `un_sur_deux` appliqué à une liste à un seul élément produit encore la liste vide
- `un_sur_deux` appliqué à une liste contenant au moins deux éléments se calcule de la manière suivante
  - appelons `h` le premier élément
  - appelons `t` la suite de la liste
  - calculons `un_sur_deux t`
  - utilisons ce dernier résultat pour donner la suite de notre nouvelle liste et `h` comme premier élément
  - ceci est notre résultat."

Formulé différemment, nous avons parcouru la liste en éliminant un élément sur deux.

De fait, avec la notation raccourcie, nous pourrions réécrire la fonction sous la forme suivante

```
# let rec un_sur_deux = function
   | []                -> []
   | [ _ ]             -> [] (* une liste composée d'un seul élément *)
   | h :: _ :: t       -> h :: (un_sur_deux t) ;;
val un_sur_deux : 'a list -> 'a list = <fun>
```

#### Exercices

1. Écrivez une fonction qui détermine le dernier élément d'une liste.
2. Écrivez une fonction qui détermine l'avant-dernier élément d'une liste.
3. Écrivez une fonction qui inverse une liste.

## Arbres

De la même manière que nous avons défini des listes à l'aide d'un type somme récursif, ces mêmes types peuvent servir à définir des arbres. Ainsi, on peut représenter les formules de logique booléenne sous la forme suivante :

```
# type formule =
    | Booleen  of bool
    | Variable of string
    | Et       of formule * formule
    | Ou       of formule * formule
    | Non      of formule
    | Implique of formule * formule ;;
type formule =
  | Booleen of bool
  | Variable of string
  | Et of formule * formule
  | Ou of formule * formule
  | Non of formule
```

```
  | Implique of formule * formule

# Booleen true ;
- : formule = Booleen true
```

Partant de là, nous pouvons définir des transformations sur les arbres :

```
# let formule_ou_exclusif a b = Et ( Ou ( a, b ), Ou ( Non a, (Non b ) ) );;
val formule_ou_exclusif : formule -> formule -> formule = <fun>

# formule_ou_exclusif ( Booleen true ) ( Booleen false ) ;;
- : formule =
Et (Ou (Booleen true, Booleen false),
 Ou (Non (Booleen true), Non (Booleen false)))
```

De même, nous pouvons programmer l'évaluation d'une expression booléenne, en supposant qu'aucune variable n'apparaît dans l'expression :

```
# let rec evalue arbre = match arbre with
  | Booleen _ -> arbre
  | Et (f, g) -> (match (evalue f, evalue g) with
     | (Booleen true, Booleen true)   -> Booleen true
     | _                              -> Booleen false )
  | Ou (f, g) -> (match (evalue f, evalue g) with
     | (Booleen false, Booleen false) -> Booleen false
     | _                              -> Booleen true )
  | Non f     -> (match (evalue f) with
     | Booleen true  -> Booleen false
     | Booleen false -> Booleen true
     |  _            -> failwith "Erreur dans l'évaluation d'une négation" )
  | Implique (f, g) -> (match (evalue f, evalue g) with
     | (_, Booleen true  )   -> Booleen true
     | (Booleen false, _ )   -> Booleen true
     | _                     -> Booleen false )
  | _ -> failwith "Erreur dans l'évaluation d'une formule" ;;

val evalue : formule -> formule = <fun>
```

Dans ce qui précède, nous avons procédé à du filtrage par motifs sur le couple de valeurs `(simplifie f, simplifie g)`. Le résultat est un couple, sur lequel nous pouvons de nouveau procéder à du filtrage par motifs. Nous avons aussi utilisé la fonction `failwith`, qui provoque une erreur et interrompt immédiatement le calcul. Nous reparlerons de cette fonction lorsque nous nous intéresserons à la gestion des erreurs.

Notez aussi que nous avons été obligés de mettre entre parenthèses les `match...` intérieurs, pour éviter toute ambigüité.

Nous pourrions étendre la fonction précédente de manière à simplifier autant que possible les expressions contenant des variables :

```
# let rec simplifie arbre = match arbre with
   | Booleen _  -> arbre    (*Impossible de simplifier un booléen  *)
   | Variable _ -> arbre    (*Impossible de simplifier une variable*)
   | Et (f, g)  -> (match (simplifie f, simplifie g) with
       ( ( Booleen false, _ ) -> Booleen false   (*Faux et x => Faux *)
       | ( _, Booleen false ) -> Booleen false   (*x et Faux => Faux *)
       | ( Booleen true,  x ) -> x               (*Vrai et x => x    *)
       | ( x,  Booleen true ) -> x               (*x et Vrai => x    *)
```

```
      | (f', g')              -> Et (f', g'))    (*Sinon, on ne peut pas simplifier plus
  | Ou (f, g)   -> match (simplifie f, simplifie g) with
      | ( Booleen true, _  ) -> Booleen true    (* ... *)
      | ( _, Booleen frue  ) -> Booleen true
      | ( Booleen false,  x) -> x
      | ( x,  Booleen false) -> x
      | (f', g')             -> Ou (f', g') )
  | Non f       -> ( match simplifie f with
      | Booleen true         -> Booleen false
      | Booleen false        -> Booleen true
      | x                    -> Non x )
  | Implique (f, g) -> (match (simplifie f, simplifie g) with
    | ( _, Booleen true  )   -> Booleen true
    | ( Booleen false, _ )   -> Booleen true
    | ( Booleen true, Booleen false ) -> Booleen false
    | ( f', g')              -> Implique (f', g') );;
val simplifie : formule -> formule = <fun>
```

**Exercices**

1. À ce qui précède, nous pourrions ajouter des simplifications supplémentaires, telles que
   - tirer parti du fait que, pour tout *x*, *non non x* a la même valeur que *x*
   - tirer parti du fait que, pour tout *x*, *x et non x* a la même valeur que *faux*
   - tirer parti du fait que, pour tout *x*, *x ou non x* a la même valeur que *vrai*

complétez la fonction `simplifie` pour procéder à toutes ces simplifications.

1. Modifiez la fonction `evalue` pour supprimer l'utilisation de `assert False` -- au lieu de cela, pour gérer

les erreurs, vous utiliserez le type `option formule`.

1. Écrivez une fonction `substitue : string -> bool -> formule -> formule`, qui prend en argument un nom de variable, une valeur `v` à donner à cette variable et une formule et renvoie une formule dans laquelle toutes les occurrences de la variable ont été remplacées par `Booleen v`.
2. Modifiez la fonction `substitue` pour que, au lieu d'un booléen, elle accepte n'importe quelle `formule`.

# Produit cartésien

Jusqu'à présent, nous avons manipulé les types somme. Ceux-ci permettent de représenter aisément l'union de plusieurs ensembles. Comme nous l'avons vu dans le cadre de ces types somme, un constructeur peut servir à accéder à plusieurs informations. La possibilité de combiner plusieurs informations en une seule valeur est fondamentale pour la programmation. Ainsi, l'emplacement d'un point à l'écran est en fait composé de deux entiers, l'un représentant son abscisse, l'autre son ordonnée. Une couleur pourra de même être représentée par trois entiers qui détermineront les proportions de rouge, de vert et de bleu, un nombre rationnel pourra être identifié à un entier relatif (son numérateur) et un entier strictement positif (son dénominateur)...

Mathématiquement, dans chacun de ces cas, la représentation peut être modélisée sous la forme d'un produit cartésien. En OCaml, plusieurs techniques sont disponibles pour implanter ces structures de données. Avec les types `pair`, `triple`,... nous avons déjà vu comment utiliser les types sommes pour arriver à ces fins. Dans cette section, nous verrons deux alternatives souvent plus confortables : les *n-uplets* et les *enregistrements*.

## Couples et n-uplets

Au cours du chapitre sur les fonctions, nous avons déjà rencontré les n-uplets, sans les nommer. En OCaml, ceux-ci servent à rassembler plusieurs valeurs en une seule, sans plus de précisions. Ainsi, considérons l'extrait suivant :

```
# ( 1, 2 ) ;;
- : (int * int) = (1, 2)
```

La réponse d'OCaml s'énonce "le résultat de la dernière évaluation est de type `int * int`, c'est-à-dire un couple d'entiers, et vaut `(1, 2)`".

Cette syntaxe se généralise aux triplets et autres n-uplets, qui peuvent eux-mêmes contenir des éléments de n'importe quel type :

```
# ( true, 2, "autre chose" ) ;;
- : (bool * int * string) = (true, 2, "autre chose")

# ( 1, 'a', ( 1, 2, 3 ), "encore des éléments", false ) ;;
- : (int * char * (int * int * int) * string * bool) = (1, 'a', (1, 2, 3), "encore des él
```

Sans surprise, nous pouvons aussi produire un n-uplet comme résultat d'une fonction :

```
# let creer_couple a b = ( a, b ) ;;
val creer_couple : 'a -> 'b -> ('a * 'b) = <fun>

# creer_couple 1 2 ;;
- : (int * int) = (1, 2)
```

> **Vocabulaire** On dit que `( _, _ )` est le *constructeur* du type `'a * 'b`. De même, `( _, _, _ )` est le constructeur du type `'a * 'b * 'c`, `( _, _, _, _ )` celui de `'a * 'b * 'c * 'd`... Une fois de plus, cette notion de constructeur n'a rien à voir avec les constructeurs de la programmation orientée objets.

Profitons de ce dernier exemple pour remarquer que les couples forment un type polymorphe à deux paramètres, ici `'a` et `'b`.

Les deux fonctions duales de `creer_couple`, prédéfinies dans OCaml (ou triviales à redéfinir, comme nous le verrons plus loin), sont `fst` et `snd`, les *projections* :

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

Comme leur type l'indique, `fst` et `snd` prennent chacune en argument un couple et renvoient respectivement son premier élément ou son second élément :

```
# fst ( 1, 2 );;
- : int = 1
# snd ( 1, 2 );;
- : int = 2
```

Insistons sur le fait que `fst` et `snd` ne fonctionnent que sur des couples -- comme en mathématiques, un triplet n'est pas un couple, si bien que :

```
# fst ( 1, 2, 3 );;
       ^^^^^^^^^^^
This expression has type (int * int * int) but is here used with type
  ('a * 'b)

# snd ( 1, 2, 3 );;
       ^^^^^^^^^^^
This expression has type (int * int * int) but is here used with type
  ('a * 'b)
```

Comment faire, alors, pour consulter le premier, le deuxième ou le troisième élément d'un n-uplet ? En fait, le mécanisme le plus simple pour *déconstruire* un n-uplet est de nouveau le filtrage par motifs. Ici, partant du fait que `( _, _, _)` est un constructeur, nous pourrons écrire, comme pour les types somme :

```
# let fst_of_3 triplet = match triplet with
   ( a, b, c ) -> a ;;
val fst_of_3 : ('a * 'b * 'c) -> 'a = <fun>
```

ou, de manière plus concise,

```
# let fst_of_3 = function ( a, b, c ) -> a ;;
val fst_of_3 : ('a * 'b * 'c) -> 'a = <fun>
```

ou encore, en combinant le filtrage par motifs avec la définition de fonction,

```
# let fst_of_3 triplet = let ( a, b, c ) = triplet in a ;;
val fst_of_3 : ('a * 'b * 'c) -> 'a = <fun>

# let fst_of_3 ( a, b, c ) = a ;;
val fst_of_3 : ('a * 'b * 'c) -> 'a = <fun>

# let fst_of_3 ( a, _, _ ) = a ;;
val fst_of_3 : ('a * 'b * 'c) -> 'a = <fun>
```

Profitons-en pour rappeler l'une des utilisations des n-uplets, pour définir plusieurs valeurs en une seule fois :

```
# let (a, b, c) = ( 1, 2, 3 );;
val a : int = 1
val b : int = 2
val c : int = 3
```

> **Note** En OCaml, puisque les couples, triplets et autres n-uplets n'ont pas le même type, il n'est pas possible d'écrire une fonction `fst` qui fonctionnera à la fois sur les couples, triplets et autres n-uplets. Il serait cependant possible de définir par méta-programmation une famille de fonctions `fst_of_N` pour tout `N` telle que, pour chaque `N`, `fst_of_N` puisse agir sur les N-uplets. Nous reviendrons dans un chapitre ultérieur sur les techniques de méta-programmation.

## Enregistrements

Si les paires et autres tuples offrent une manière simple de manipuler deux, trois ou plus d'informations, les utiliser en permanence pose un risque d'ambigüité. En effet, un couple de nombres flottants peut servir à représenter aussi bien des coordonnées sur une carte, un nombres complexe sous forme cartésienne, un nombre complexe sous forme polaire, la probabilité d'un évènement, etc. Si chacun de ces usages est tout à fait acceptable, utiliser le même type de données pour représenter des informations sans rapport est une pratique découragée car elle interdit à OCaml de vérifier le bon usage de ces informations.

Pour plus de sûreté, nous pourrions donc employer des types sommes dotés d'un seul constructeur. Malheureusement, pour ce genre d'usage, leur lisibilité peut laisser à désirer. Ainsi, considérons un moment le type suivant :

```
# type polaire = Polaire of float * float ;;
```

Si nous souhaitons employer ce type pour représenter les nombres complexes sous forme polaire, c'est-à-dire sous la forme d'un argument et d'un module, nous devrons nous munir d'une convention. Par exemple, convenir que, dans `Polaire (x, y)`, `x` est toujours l'argument et `y` toujours le module. Si ceci est toujours acceptable, de nouveau, nous avons introduit une fragilité, puisque c'est à nous de vérifier à chaque utilisation que nous ne confondons pas argument et module. L'idéal serait de permettre à OCaml de vérifier ceci systématiquement.

C'est pour cette raison qu'OCaml propose un autre style de produit cartésien : les *enregistrements*. Ainsi, la déclaration suivante définit un nouveau type, nommé `cartesien`, et qui va contenir deux nombres flottants, que nous choisissons de nommer `re` et `im` :

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

> **Vocabulaire** Dans ce qui précède, `re` et `im` sont les *champs* de l'*enregistrement* `cartesien`.
>
> **Note** Le terme "enregistrement" n'a rien à voir avec le fait d'enregistrer un fichier sur le disque dur. Il s'agit uniquement d'une manière de représenter des informations durant l'exécution d'un programme. Nous verrons dans un chapitre ultérieur comment enregistrer et récupérer des informations dans un fichier.

À partir du moment où `cartesien` est défini, nous pouvons créer de nouvelles valeurs de ce type :

```
# { re = 5.0; im = 10.0 } ;;
- : cartesien = {re=5; im=10}

# let i = { im = 1.; re = 0.} ;;
val i : cartesien = {re=0; im=1}
```

Ce dernier extrait s'énonce "Appelons `i` la valeur dont `im` vaut 1.0 et `re` vaut 0.0." Notons que l'ordre dans lequel sont donnés `im` et `re` est sans importance.

Toute tentative de créer une valeur de type `cartesien` contenant trop ou trop peu de définitions de champs provoquera une erreur lors de l'analyse de types :

```
# let x = { im = 5. } ;;
```

```
          ^^^^^^^^^^^
Some record field labels are undefined: re

# let y = { im = 5.; re = 3.; stuff = 8. } ;;
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Unbound record field label stuff

# let z = { im = 5.; re = 3.; re = 8. } ;;
          ^^^^^^^^^^^^^^^^^^^^^^^^^^
The record field label re is defined several times
```

> **Vocabulaire** `{ im = _; re = _}` est le *constructeur* de type `cartesien`. Comme l'ordre des champs est sans importance, on pourra considérer, au choix, que le constructeur de ce type est `{ re = _; im = _}`.

Pour consulter la valeur d'un champ, on pourra employer la notation `.` (le point), similaire à celle des langages orientés-objets :

```
# let partie_reelle c = c.re ;;
val partie_reelle : cartesien -> float = <fun>

# partie_reelle i ;;
- : float = 0
```

Une autre possibilité consiste à utiliser le filtrage par motifs :

```
# let partie_relle c = match c with
   { re = x; im = _ } -> x  ;;
val partie_reelle : cartesien -> float = <fun>

# let partie_relle = function
   { re = x; im = _ } -> x ;;
val partie_reelle : cartesien -> float = <fun>

# let partie_relle { re = x; im = _ } = x ;;
val partie_reelle : cartesien -> float = <fun>
```

Qui plus est, OCaml propose un raccourci supplémentaire : il n'est pas nécessaire de faire figurer dans le motif les champs qui ne nous intéressent pas. Ainsi, nous pourrons écrire encore plus simplement

```
# let partie_relle c = match c with  { re = x } -> x ;;
val partie_reelle : cartesien -> float = <fun>

# let partie_relle = function { re = x } -> x;;
val partie_reelle : cartesien -> float = <fun>

# let partie_relle { re = x } = x ;;
val partie_reelle : cartesien -> float = <fun>
```

Et, pour achever la simplification de la syntaxe, comme avec les n-uplets, nous pouvons affecter directement une valeur, par exemple

```
# let { re = y } = i;;
val y : float = 0;;
```

Muni de ces outils, nous pouvons définir les opérations habituelles sur les nombres complexes :

```
# let add_cartesiens {re = x; im = y} {re = x'; im = y'} = {re = x +. x'; im = y +. y'} ;
val add_cartesiens : cartesien -> cartesien -> cartesien = <fun>

# let mult_cartesiens {re = x; im = y} {re = x'; im = y'} = {re = x *. x' -. y *. y'; im =
val mult_cartesiens : cartesien -> cartesien -> cartesien = <fun>
```

etc.

On définirait de même une représentation polaire des nombres sous la forme :

```
# type polaire   = { arg : float ; mo : float };;
type polaire   = { arg : float ; mo : float }
```

> **Note** Nous n'avons utilisé ni le nom `module` ni le terme `mod` pour désigner le module d'un nombre sous forme polaire car ces deux mots sont réservés par le langage.

Pour le moment, nous n'avons pas vu de méthode pour interdire de construire un `polaire` avec un module négatif, ce qui constituerait une erreur grossière mais potentiellement difficile à retrouver dans le code. Nous verrons plus tard comment gérer ce cas, avec les mêmes méthodes que nous emploierons aussi pour interdire de construire des cartes de tarot dont le numéro d'Atout n'est pas compris entre 0 et 21.

Sans surprise, OCaml interdit d'appliquer à la représentation polaire une fonction prévue pour la représentation cartésienne :

```
# add_cartesiens { arg = 0.; mo = 0. } ;;
                 ^^^^^^^^^^^^^^^^^^^
This expression has type polaire but is here used with type cartesien
```

Pour faire interagir complexes cartésiens et complexes polaires, il sera nécessaire de définir des fonctions de conversion. Par convention, nous les appellerons `cartesien_of_polaire` et `polaire_of_cartesien` :

```
# let cartesien_of_polaire { arg = a; mo = m } = { re = m *. cos a ; im = m *. sin a } ;;
val cartesien_of_polaire : polaire -> cartesien = <fun>

# let polaire_of_cartesien { re = r; im = i } =
   {
      mo  = sqrt ( r *. r +. i *. i ) ;
      arg = atan ( i /. r )
   } ;;
val polaire_of_cartesien : cartesien -> polaire = <fun>
```

Notons que cette fonction est correcte même si `re` vaut `0.`, en raison de la manière dont `atan` gère les valeurs infinies.

Nous pourrons alors manipuler cartésiens et polaires de la même manière que nous avons manipulé entiers et nombres flottants : en introduisant des conversions manuelles dès que nécessaire. Nous pourrons aussi définir de nouveaux opérateurs sur les nombres complexes de manière à simplifier légèrement les notations :

```
# let ( +++. ) = add_cartesiens ;;
val ( +++. ) : cartesien -> cartesien -> cartesien = <fun>
```

```
# let ( ***. ) = mult_cartesiens ;;
val ( ***. ) : cartesien -> cartesien -> cartesien = <fun>

#  { mo = sqrt 2. /. 2. ; arg = 3.1415926 }  +++. { re = 1.; im = 2. } ;;
   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
This expression has type polaire but is here used with type cartesien

# polaire_of_cartesien ( cartesien_of_polaire { mo = sqrt 2. /. 2. ; arg = 3.1415926 }  +
- : polaire = {arg=1.42538337901; mo=2.02133287442}
```

Cette obligation de convertir explicitement est parfois lourde mais nécessaire pour garantir la bonne manipulation des nombres cartésiens et polaires sans sacrifier la vitesse ni introduire d'ambigüités.

## Enregistrements vs. N-uplets vs. Sommes

La construction de nouveaux types sous la forme d'enregistrements ou de n-uplets ou de types sommes avec un seul constructeur servira les mêmes objectifs. Pour cette raison, de nombreux langages ne proposent comme structure de donnée que des enregistrements (ou des classes) ou que des n-uplets (ou des listes non typées). Par opposition, OCaml offre la possibilité de choisir une syntaxe différente et adaptée au problème. S'il n'y a pas un critère universel pour décider entre ces possibilités, on peut tout de même citer quelques conseils de bon sens.

### Cas traités avec des types sommes

On emploiera des types sommes dès que les données peuvent prendre plusieurs formes, comme c'est le cas pour la liste (qui peut être vide ou non-vide, seul cas où on aura besoin de connaître une valeur et une suite de liste). Dans les autres cas, on préférera se passer des types sommes.

#### Cas traités avec des N-uplets

Comme nous l'avons déjà vu, il est possible d'utiliser des N-uplets pour lier en une seule évaluation plusieurs valeurs, que ce soit localement ou globalement. S'il serait possible de faire la même chose avec des enregistrements, on préférera souvent les N-uplets, plus confortables à manipuler.

```
# let (x,y) = (10, 15) ;;
val x : int = 10
val y : int = 15

# let (valeur_de_pi, incertitude_entre_valeurs) =
    let ( pi_1, pi_2 ) =
      ( 2. *. acos ( 0. ), 2. *. asin ( 1. ) )
    in
      ( pi_1, abs_float ( pi_1 -. pi_2 ) );;
val valeur_de_pi : float = 3.14159265359
val incertitude_entre_valeurs : float = 0
```

Le mécanisme est aussi utilisé fréquemment dès qu'il s'agit de produire plusieurs résultats à l'intérieur d'une sous-fonction.

### Cas traités avec des enregistrements

Comme nous l'avons vu, utiliser des enregistrements permet à OCaml de détecter plus d'erreurs lors de l'analyse de types que les N-uplets. On aura donc tendance à utiliser des enregistrements dès que les informations qui constituent la valeur vont devoir être utilisées ensemble fréquemment et seront

généralement conservées ensemble.

> **Note** Nous verrons aussi dans le chapitre consacré à la programmation impérative comment définir et utiliser des enregistrements *mutables*.

## À titre de comparaison

### N-uplets en Java

En Java, on pourrait définir de même les n-uplets sous la forme d'une série de classes

```
1 public class Pair<A,B>
2 {
3   public final A fst;
4   public final B snd;
5   public Pair(A fst, B snd)
6   {
7     this.fst = fst;
8     this.snd = snd;
9   }
10 }
11
12 public class Triple<A,B,C>
13 {
14   public final A fst;
15   public final B snd;
16   public final C thrd;
17   public Pair(A fst, B snd, C thrd)
18   {
19     this.fst = fst;
20     this.snd = snd;
21     this.thrd= thrd;
22   }
23 }
24
25 //...autant de classes que nécessaire
```

Pour construire et manipuler un n-uplet, on emploierait

```
1 Triple <Integer, String, Float> myTriple = new Triple<Integer, String, Float>(100, "blu
2 System.out.println (myTriple.fst);
3 //...
```

En général, la version OCaml sera plus concise, plus rapide et prendra moins de mémoire vive -- mais les idées sont les mêmes.

### Enregistrements en Java

Les enregistrements se traduisent naturellement en classes Java : la notion d'enregistrement OCaml est très proche de la notion de classe en Java, sans constructeur, sans modificateurs d'accès, sans héritage et sans accès à `this`. Nous verrons dans un chapitre ultérieur comment retrouver avec (ou sans) les enregistrements des fonctionnalités comparables en OCaml.

# Types de fonctions

Nous avons déjà rencontré et manipulé les types de fonctions en OCaml.

Les types de fonctions se manipulent exactement de la même manière que tous les autres types en OCaml:

```
# type int_to_int = int -> int ;;
type int_to_int = int -> int

# (fun x -> x, fun x -> x + 1) ;;
- : ('a -> 'a * int -> int) = (<fun>, <fun>)
```

# Précautions et limitations

## Masquage de types

De la même manière qu'un nom de valeur peut être masqué si l'on donne le même nom à une autre valeur, le nom d'un type, d'un champ ou d'un constructeur peut être masqué si l'on donne le même nom à un autre type/champ/constructeur.

Cela peut donner lieu à des messages d'erreur surprenants :

```
# type t = { champ : int };;
type t = { champ : int }

# let unmake_t { champ = x } = x;;
value unmake_t : t -> int = <fun>
# unmake_t { champ = 5 };
- : int = 5

# type t = { champ : int };;
type t = { champ : int }

# unmake_t { champ = 5 };;
           ^^^^^^^^^^^^^^
This expression has type t but is here used with type t
```

Ici, le problème est que l'ancien type `t` a été remplacé par un nouveau type `t`. Même si ce nouveau type est identique au précédent, OCaml fait la différence. Si ce genre de choses ne devrait pas vous poser de problèmes lorsque vous compilez un programme, faites attention à ces redéfinitions de types lorsque vous utilisez la ligne de commande OCaml.

De même, on rencontrera le problème suivant si un constructeur de type somme est redéfini :

```
# type a = A | B ;;
type a = A | B
# type b = B | C ;;
type b = B | C
# B ;;
- : b = B
# A ;;
- : a = A
# fun A -> 1 | B -> 2 ;;
               ^
This pattern matches values of type b
but is here used to match values of type a
```

# Exemple en taille réelle : vers un langage de programmation

Jusqu'à présent, nous avons travaillé sur des exemples de quelques lignes. Cela dit, rien ne nous oblige à nous cantonner à de petits extraits. De fait, maintenant, que nous avons fait un premier tour des structures de données en OCaml, nous disposons de toutes les connaissances nécessaires pour définir le noyau d'un langage de programmation, sous la forme d'un *interpréteur à grands pas*. C'est ce que nous allons faire tout de suite. Dans la suite de l'ouvrage, nous allons progressivement compléter cet interpréteur pour en faire une implantation complète d'un langage de programmation minimal.

Avant toute autre chose, commençons par décider à quoi ressemblera notre langage. Ce langage sera un petit sous-ensemble de JavaScript, doté de

- variables dynamiquement typées
- une boucle `for` et une boucle `while`
- fonctions d'ordre supérieur (donc avec la possibilité de renvoyer une fonction depuis une fonction ou de prendre une fonction comme argument d'une fonction).

Un code source pourra ressembler à :

```
print ("Bonjour, le monde !");
var x = 0;
var add = function(x, y)
{
  return x + y
};
x = 7;
print(add(x, 5));
```

## Syntaxe abstraite

Une fois que nous avons à peu près décidé des fonctionnalités du langage, il convient de choisir sa *syntaxe abstraite*, c'est-à-dire de décider à quoi ressemble un programme, une fois qu'on a supprimé tous les aspects purement lexicaux tels que les commentaires, les espaces, le choix du caractère de séparation, etc., pour se plonger dans le monde des mathématiques ou des structures de données. Ici,

- un programme est une suite d'instructions
- une instruction est
  - soit un bloc d'instructions, qui est à son tour une suite d'instructions
  - soit une définition de variable
  - soit une expression
  - soit une affectation
  - soit `return`
  - soit un test avec une condition (qui est une expression), une instruction à exécuter si la condition est vraie et une instruction à exécuter si la condition est fausse
- une définition de variable est la donnée de
  - un nom de variable
  - et en option une valeur donnée à la variable, c'est-à-dire une expression
- une expression est
  - soit une constante
  - soit une variable, caractérisée uniquement par son nom

    - soit un appel de fonction (nous considérerons que les opérations arithmétiques sont des fonctions comme les autres)
- une constante est
    - soit un nombre entier
    - soit un texte
    - soit une fonction
    - soit la valeur spéciale "pas de valeur"
- la valeur d'une fonction est la donnée de
    - une liste d'arguments, c'est-à-dire une liste de noms
    - et le corps de la fonction, c'est-à-dire une liste d'instructions
    - et la valeur des variables locales utilisées par la fonction (la clôture proprement dite), c'est-à-dire une liste de noms de variables et de constantes
- un appel de fonction est la donnée de
    - une fonction appelée, c'est-à-dire une expression
    - et des arguments, c'est-à-dire une liste d'expressions
- une affectation est la donnée de
    - un nom de variable
    - et une valeur donnée à la variable, c'est-à-dire ici aussi une expression
- un `return` est la donnée d'une expression
- un nombre est un entier OCaml (pour le moment -- nous déciderons peut-être plus tard de remplacer les entiers par des nombres dont la taille est limitée uniquement par la quantité de mémoire de l'ordinateur)
- un texte est une chaîne de caractères OCaml (pour le moment -- ici aussi, nous déciderons peut-être plus tard de remplacer ceci par une autre manière de représenter le texte)
- un nom est une chaîne de caractères OCaml (de nouveau, ceci pourra changer)

Reprenons tout ceci point par point et construisons les types dont nous aurons besoin :

- un programme est une suite d'instructions

Ceci se traduit immédiatement par

```
type programme = instruction list
```

à charge pour nous de définir le type `instruction`

- une instruction est
    - soit un bloc d'instructions, qui est à son tour une suite d'instructions
    - soit une définition de variable
    - soit une expression
    - soit une affectation
    - soit `return`

Cette succession de "soit" se traduit immédiatement en un type somme tel que

```
and instruction =
| Bloc        of instruction list
| Definition  of definition
| Expression  of expression
| Affectation of affectation
| Return      of return
| Si          of expression * instruction * instruction
```

Pour que ce type ait un sens, il nous reste à définir `instruction`, `definition`, `expression`, `affectation` et `return`.

- une définition de variable est la donnée de
  - un nom de variable
  - et en option une valeur donnée à la variable, c'est-à-dire une expression

Ici, il s'agit d'un "et", donc un type produit. Pour plus de clarté, nous allons employer un enregistrement.

```
and definition =
{
  definition_nom : nom;
  definition_val : expression option
}
```

Nous avons ici opté pour un type `nom` plutôt que d'employer directement `string`, de manière à pouvoir remettre à plus tard la décision sur la manière dont est représenté un nom. Ainsi, si nous souhaitons uniquement manipuler des langages européens, `string` est parfaitement approprié mais si nous souhaitons pouvoir programmer dans n'importe quelle langue, nous utiliserons plutôt un type de *ficelles* de caractères *Unicode* plutôt que le type par défaut des *chaînes* européennes.

Nous avons aussi opté pour une `option`, introduit précédemment, pour les informations optionnelles.

Enfin, nous avons choisi des noms relativement complexes pour les champs pour éviter toute collision entre les divers enregistrements que nous allons manipuler.

- une expression est
  - soit une constante
  - soit une variable, caractérisée uniquement par son nom
  - soit la valeur d'une fonction
  - soit un appel de fonction (nous considérerons que les opérations arithmétiques sont des fonctions comme les autres)
  - soit un test avec une condition (qui est une expression), une instruction à exécuter si la condition est vraie et une instruction à exécuter si la condition est fausse

De nouveau, cette succession de "soit" se traduit naturellement en un type somme :

```
and expression =
| Constante of constante
| Variable  of nom
| Appel     of appel
```

- une constante est
  - soit un nombre entier
  - soit un texte
  - soit une fonction
  - soit la valeur spéciale "pas de valeur"

```
type constante =
| Nombre    of nombre
| Texte     of texte
| Fonction  of fonction
```

```
| Pas_de_valeur
```

- la valeur d'une fonction est la donnée de
  - une liste d'arguments, c'est-à-dire une liste de noms
  - et le corps de la fonction, c'est-à-dire une liste d'instructions
  - et la valeur des variables locales utilisées par la fonction (la clôture proprement dite), c'est-à-dire une liste de noms de variables et de constantes

```
type fonction =
{
 fonction_arguments : nom list;
 fonction_corps     : instruction list;
 fonction_cloture   : ( nom * constante ) list;
};;
```

- un appel de fonction est la donnée de
  - une fonction appelée, c'est-à-dire une expression
  - et des arguments, c'est-à-dire une liste d'expressions

```
type appel =
{
 appel_fonction  : expression;
 appel_arguments : expression list
};;
```

- une affectation est la donnée de
  - un nom de variable
  - et une valeur donnée à la variable, c'est-à-dire ici aussi une expression

```
type affectation =
{
 affectation_nom : nom;
 affectation_val : expression
};;
```

- un `return` est la donnée d'une expression

```
type return = expression;;
```

- un nombre est un entier OCaml (pour le moment -- nous déciderons peut-être plus tard de remplacer les entiers par des nombres dont la taille est limitée uniquement par la quantité de mémoire de l'ordinateur)

```
type nombre = int;;
```

- un texte est une chaîne de caractères OCaml (pour le moment -- ici aussi, nous déciderons peut-être plus tard de remplacer ceci par une autre manière de représenter le texte)

```
type texte = string;;
```

- un nom est une chaîne de caractères OCaml (de nouveau, ceci pourra changer)

```
type nom = string;;
```

En fait, ceci est un peu trop permissif : en général, on considère que seules certaines chaînes de caractères peuvent être des noms de variables. Par exemple, `"0"`, `" "`, `"("` ou `"f x"` ne sont généralement pas admis comme noms de variables. Pour le moment, nous allons nous contenter de supposer que le type `nom` ne contient que des noms corrects. Au prochain chapitre, nous allons introduire le mécanisme nécessaire pour empêcher les chaînes mal formées d'être employées comme noms.

Une fois tout ceci rassemblé et mis dans un ordre compréhensible par OCaml, nous obtenons :

```
 1 type programme = instruction list
 2
 3 and instruction =
 4 | Bloc        of instruction list
 5 | Definition  of definition
 6 | Expression  of expression
 7 | Affectation of affectation
 8 | Return      of return
 9 | Si          of expression * instruction * instruction
10
11 and definition =
12 {
13   definition_nom : nom;
14   definition_val : expression option
15 }
16
17 and expression =
18 | Constante of constante
19 | Variable  of nom
20 | Appel     of appel
21
22 and constante =
23 | Nombre    of nombre
24 | Texte     of texte
25 | Fonction  of fonction
26 | Pas_de_valeur
27
28 and fonction =
29 {
30  fonction_arguments : nom list;
31  fonction_corps     : instruction list;
32  fonction_cloture   : ( nom * constante ) list;
33 }
34
35 and appel =
36 {
37  appel_fonction  : expression;
38  appel_arguments : expression list
39 }
40
41 and affectation =
42 {
43  affectation_nom  : nom;
44  affectation_val  : expression
45 }
46
47 and return = expression
48 and nombre = int
49 and texte = string
```

```
  50 and nom = string;;
```

## Comment créer et afficher un arbre de syntaxe abstraite ?

Nous disposons maintenant de la possibilité de manipuler un programme pour notre mini-JavaScript. Ainsi, l'extrait plus haut se notera

```
let programme_de_test =
[
 Expression (Appel {appel_fonction  = Variable "print"; appel_arguments = [
    Constante (Texte "Bonjour, le monde !")
 ]});
 Definition {definition_nom = "x";   definition_val = Some (Constante (Nombre 0))};
 Definition {definition_nom = "add"; definition_val =
   Some (Constante (Fonction {fonction_arguments = ["x"; "y"];
    fonction_corps     = [Return (Appel {
      appel_fonction = Variable "+";
      appel_arguments= [
        Variable "x";
        Variable "y"
      ]})];
    fonction_cloture   = []
   }))
 };
 Affectation {affectation_nom = "x"; affectation_val = Constante (Nombre 7)};
 Expression (Appel {appel_fonction  = Variable "print"; appel_arguments = [
    Appel {appel_fonction = Variable "add"; appel_arguments = [
      Variable "x";
      Constante (Nombre 0)
    ]}
 ]})
];;
```

Ceci est assez peu agréable aussi bien à écrire qu'à relire. Pour le moment, nous n'avons pas vu les concepts nécessaires pour permettre à l'utilisateur de notre langage d'écrire un programme directement en mini-JavaScript. Par contre, nous pouvons définir une fonction qui affichera un programme plus esthétique.

Pour ce faire, nous allons définir autant de fonctions qu'il y a de types dans notre arbre de syntaxe abstraite, fonctions qui renverront une chaîne de caractères en mini-JavaScript.

> **Note** En OCaml, la concaténation de chaînes de caractères est une opération lente. En temps normal, nous utiliserions une implantation différente qui éviterait toute concaténation. Nous verrons cette implantation dans le chapitre consacré aux entrées/sorties. Cette implantation nous permettra aussi d'automatiser une partie des traitements que nous sommes sur le point de programmer manuellement. Pour le moment, nous allons donc nous contenter de la version relativement verbeuse et peu efficace, qui nous suffira largement pour des programmes courts.

Pour ce faire, commençons par définir une fonction qui convertit une liste d'instructions en une chaîne de caractères. Cette fonction nous servira plusieurs fois :

```
let rec string_of_instruction_list l = match l with
 | []   -> ""
 | h::t -> (string_of_instruction h)^";\n"^(string_of_instruction_list t)
```

Rappelons que le symbole `^` sert à concaténer deux chaînes de caractères. La chaîne de caractères `";\n"` est un peu inhabituelle, puisqu'elle fait intervenir le symbole spécial `\n`, qui représente le retour à la ligne. À

partir d'une liste d'instructions, cette cette fonction produit donc une chaîne de caractères, dans laquelle chaque instruction s'achève par un point-virgule et un retour à la ligne.

Bien entendu, cette fonction n'a aucun sens sans la fonction `string_of_instruction_list`, définie comme suit :

```
and string_of_instruction = function
 | Bloc b        -> "{\n" ^ string_of_instruction_list b ^ "}\n"
 | Definition d  -> string_of_definition d
 | Expression e  -> string_of_expression e
 | Affectation a -> string_of_affectation a
 | Return r      -> string_of_return r
 | Si (x, y, z)-> "if("^(string_of_expression x)^")\n"^(string_of_instruction y)^"\n else
```

cette fonction appelle la fonction `string_of_instruction_list` pour traiter les blocs et appelle `string_of_definition`, `string_of_expression`, `string_of_affectation` ou `string_of_return` pour traiter les autres cas -- à charge pour nous d'écrire toutes ces fonctions.

Intéressons-nous aux définitions de variables dans notre mini-JS :

```
and string_of_definition d = match d.definition_val with
 | None   -> "var "^d.nom
 | Some v -> "var "^d.nom^" = "^(string_of_expression v)
```

cette fonction se construit par cas. Si la définition ne contient pas de valeur, nous allons juste renvoyer `var toto`, où `toto` est le nom de la variable que nous venons de créer. Si une valeur est présente, nous devons renvoyer `var toto = blop` où `blop` est l'expression correspondante. Pour déterminer dans quel cas nous sommes, il nous suffit de vérifier si `d.definition_val` est de la forme `None` ou de la forme `Some v`, où `v` est une expression -- rappelons qu'il s'agit des deux seules formes possibles pour `d.definition_val` car ce champ est de type `expression option`.

La prochaine étape est donc de définir `string_of_expression`, à peu près de la même manière que `string_of_instruction`:

```
and string_of_expression = function
| Constante c -> string_of_constante c
| Variable  v -> string_of_nom       v
| Appel     a -> string_of_appel     a
```

Enchaînons sur `string_of_constante`, `string_of_nombre` et `string_of_texte` qui sont aussi simples:

```
and string_of_constante = function
| Nombre n      -> string_of_nombre n
| Texte  t      -> string_of_texte t
| Fonction  f   -> string_of_fonction  f
| Pas_de_valeur -> ""
and string_of_nombre n = string_of_int n
and string_of_texte  t = t
and string_of_nom    n = n
```

Il nous reste encore à gérer la définition de fonctions. Pour ce faire, commençons par spécifier la conversion d'une liste de noms d'arguments en une chaine de caractères. Toute la difficulté est dans la gestion de la

virgule, qui doit apparaître entre deux éléments consécutifs :

- pour une liste, vide, le résultat est vide
- pour une liste composée d'exactement un élément, le résultat est cet élément
- pour une liste composée d'au moins deux éléments, le résultat est le premier élément, suivi d'une virgule, suivi du résultat appliqué à la liste privée de son premier élément.

En d'autres termes,

```
and string_of_nom_list = function
| []      -> ""
| [x]     -> string_of_nom x
| x::y::t -> string_of_nom x ^ ", "^ string_of_nom_list (y::t)
```

À partir de cette définition, nous pouvons construire `string_of_fonction`.

```
and string_of_fonction f =
 "function ("^string_of_nom_list f.fonction_arguments^") {\n"^string_of_instruction_list
```

Cette version de `string_of_function` ne prend pas en compte la clôture elle-même. Si nécessaire, nous réécrirons `string_of_function` plus tard de manière à afficher plus de détails.

De la même manière que nous avons implanté l'affichage d'une liste de noms, nous pouvons construire l'affichage d'une liste d'expressions, dont nous déduisons l'appel de fonction :

```
and string_of_expression_list = function
| []      -> ""
| [x]     -> string_of_expression x
| x::y::t -> string_of_expression x ^ ", "^ string_of_expression_list (y::t)
and string_of_appel a =
(string_of_expression a.appel_fonction) ^ "(" ^(string_of_expression_list a.appel_argumen
```

Il nous reste enfin à construire l'affectation et le renvoi de valeur :

```
and string_of_affectation a =
 a.affectation_nom ^ " = " ^ string_of_expression a.affectation_val
and string_of_return r =
 "return "^string_of_expression r
```

Enfin, comme un programme est une suite d'instructions, nous pouvons construire :

```
and string_of_programme = string_of_instruction_list;;
```

Nous pouvons maintenant tester le programme :

```
# let _ = print_endline (string_of_instruction_list programme_de_test);;
print(Bonjour, le monde !);
var x = 0;
var add = function (x, y) {
return +(x, y);
```

```
};
x = 7;
print(add(x, 0));

- : unit = ()
```

À la lecture de ceci, quelques remarques s'imposent :

1. la première ligne du résultat est incorrecte, puisque "Bonjour, le monde !" devrait être entre guillemets
2. l'affichage de la ligne `return` est elle aussi incorrecte, puisque le `+` devrait être entre `x` et `y`
3. le résultat pourrait être plus esthétique si `return` était écarté de quelques caractères de la marge de gauche.

Laissons ces trois problèmes en exercice au lecteur.

#### Exercices

1. Résolvez le premier problème, de manière à ce que "Bonjour, le monde !" s'affiche correctement. Pour ce faire, vous aurez besoin de retrouver dans le code source quelle fonction est responsable de renvoyer cette chaîne de caractères et d'utiliser la fonction `String.quote : string -> string`. Cette fonction sert à ajouter des guillemets autour d'une chaîne de caractères, tout en protégeant les guillemets qui peuvent apparaître à l'intérieur de la chaîne, de manière à ce qu'elle s'affiche avec les guillemets par exemple lors d'un appel à `print_endline` :

```
# print_endline "Bonjour, le monde !";;
Bonjour, le monde !
- : unit = ()

# String.quote "Bonjour, le monde !";;
- : string = "\"Bonjour, le monde !\""

# print_endline (String.quote "Bonjour, le monde !");;
"Bonjour, le monde !"
- : unit = ()
```

1. Résolvez le deuxième problème, c'est-à-dire débrouillez-vous pour que les appels à la fonction `+` s'affichent sous la forme `x+y` et non pas `+(x,y)`. Vous en profiterez pour faire de même avec les fonctions `-`, `*`, `/`, `<`, `>`, `<=`, `>=`, `=`.
2. Écrivez une fonction `espaces : int -> string` qui, à partir d'un nombre entier [n] positif ou nul, construit une chaîne de [n] caractères espaces. Ainsi, `espaces 5` produira `" "`.
3. À l'aide de la fonction `espaces`, modifiez les fonctions `string_of_...` de manière à ajouter des tabulations. Chacune de vos fonctions prendra un argument supplémentaire `n`, qui représentera le nombre d'espaces à afficher au début de chaque ligne. À chaque fois que vous entrerez dans un bloc ou dans une fonction, vous augmenterez `n` de 4.

## Exécuter un programme

La prochaine étape de notre développement consistera à implanter la **sémantique** de notre langage de programmation, c'est-à-dire de définir exactement comment s'exécute un programme. S'il existe de nombreuses manières de spécifier cette sémantique, nous allons employer un raccourci, sous la forme d'une compilation vers OCaml. En d'autres termes, à partir d'un programme défini dans notre arbre de syntaxe abstraite, nous allons générer un fichier de texte contenant un code source OCaml, que nous pourrons ensuite demander à OCaml de compiler et d'exécuter.

Tout ceci attendra que nous ayons vu comment lire et écrire des fichiers.

## ... et plus si affinités

Le système de types de OCaml dispose d'autres fonctionnalités, considérées comme plus avancées. Ces fonctionnalités sont liées à la notion de sous-typage : il s'agit d'une part des *classes* et des *objets*, à un sens proche de Java ou plutôt de Python, et d'autre part des *variants polymorphes*, qui généralisent les types sommes. Nous ne détaillerons ces concepts que dans un chapitre ultérieur.

Un autre aspect des structures de données que nous n'avons pas encore abordé est l'architecture de ces structures : comment concevoir les types et les fonctions de manière à permettre aux structures de données de tirer parti des structures existantes de OCaml, et réciproquement. Nous détaillerons ces pratiques un peu plus tard, lorsque toutes les techniques sous-jacentes auront été introduites.

# Modules

Jusqu'à présent, nous avons vu comment il était possible d'utiliser OCaml en mode interactif. Ce mode est très utile pour apprendre OCaml, pour tester des extraits ou des fonctions existantes ou pour écrire des programmes qui se comporteront eux-mêmes comme OCaml en mode interactif. Cependant, ce mode n'est généralement pas approprié pour le développement de logiciels que vous voudrez distribuer, qu'il s'agisse d'applications bureau, d'applications serveur web ou d'utilitaires. Bien entendu, OCaml permet aussi d'écrire et de compiler des logiciels autonomes adaptés à toutes ces situations. Le mécanisme employé pour ce faire porte le nom de *modules*.

Les modules ont plusieurs utilités en OCaml. En plus de permettre de produire des applications exécutables, ils permettent aussi de diviser un programme long en plus petites unités, plus simples à gérer, à documenter et à partager entre plusieurs développeurs -- c'est le concept de *modularité*. C'est encore ce mécanisme qui est utilisé pour garantir l'indépendance entre parties d'un programme ou pour garantir que certaines parties du programme ne peuvent pas être manipulées par l'extérieur -- c'est la notion d'*abstraction*.

Ainsi, si nous considérons de nouveau l'exemple des nombres complexes, nous emploierons typiquement les modules pour les raisons suivantes :

- rassembler la définition du type « nombre complexe » et toutes les fonctions qui agissent sur ce type ;
- interdire à un utilisateur de construire un nombre complexe dont le module est strictement négatif ;
- normaliser les nombres complexes pour vous assurer que l'argument est toujours compris entre 0 et 2*pi, même si l'utilisateur a donné un argument inférieur ou supérieur.

De même, dans le cadre des opérations sur les fichiers, la bibliothèque standard d'OCaml utilise des modules pour les raisons suivantes :

- rassembler toutes les opérations sur les fichiers en un endroit, pour des questions de documentation et de cohérence dans les noms de fonctions ;
- vérifier une bonne fois pour toutes, lors de l'ouverture du fichier, si le fichier existe et si l'utilisateur a bien le droit de le lire ou/et de l'écrire ;
- cacher à l'utilisateur le type de données utilisé pour représenter les fichiers, de manière à ce que les développeurs de la bibliothèque standard puissent ultérieurement modifier ce type de données en étant certains de ne pas rendre incorrects les programmes développés à l'aide d'une version donnée de la bibliothèque standard.

Au cours de ce chapitre, nous allons construire et manipuler les modules, ainsi que les informations sur les modules. À partir du chapitre suivant, nous emploierons en permanence les modules fournis avec OCaml Batteries Included.

# Objectifs du chapitre

Ce chapitre vous enseignera

- comment écrire un programme complet
- comment compiler un programme
- comment diviser un programme en modules
- comment utiliser des modules existants, qu'ils aient été écrits par vous ou par les développeurs de la bibliothèque standard d'OCaml
- comment compiler une bibliothèque pour la distribuer séparément
- comment cacher les fonctions, types et valeurs privées d'un programme ou d'un module pour éviter qu'elles soient utilisées par d'autres modules
- comment générer la documentation d'un programme ou d'une bibliothèque.

# Conventions typographiques

Dans la suite de cet ouvrage, pour différencier les extraits tapés dans le mode calculatrice des codes sources sauvegardés dans des fichiers, nous emploierons deux styles différents.

Pour le mode calculatrice, nous noterons :

```
# print_endline "Bonjour, le monde" ;;
```

Le même programme, une fois sauvegardé, sera noté avec le style suivant :

```
1 print_endline "Bonjour, le monde" ;;
```

Tout comme le symbole # du mode calculatrice ne doit pas être effectivement tapé, les numéros de lignes ne font pas partie du code source.

# Logiciels utilisés au cours de ce chapitre

## Compilation

Comme nous l'avons mentionné précédemment, OCaml est un langage *compilé*, c'est-à-dire un langage qui dispose des outils nécessaires pour transformer un code source en un fichier exécutable autonome. Plusieurs outils de compilation existent, qui à partir d'un fichier source peuvent produire soit un exécutable portable mais non-optimisé, comparable à du code compilé Java (`ocamlc` et `ocamlc.opt`), soit un exécutable optimisé mais réservé à une plate-forme, comparable à du code compilé C (`ocamlopt` et `ocamlopt.opt`), soit du code source utilisable avec une autre syntaxe (`camlp4of`, `camlp4of.opt`, `camlp4rf`, `camlp4rf.opt`), soit encore un exécutable modifié pour permettre l'analyse de performances (`ocamlcp`), de la documentation (`ocamldoc`) etc.

L'outil principal que nous allons manipuler dans le restant de cet ouvrage est OCamlBuild, dont le rôle est d'automatiser la construction d'un exécutable en gérant intelligemment tous les programmes précédents. Cet

outil, comme les précédents, est fourni avec OCaml.

La manière la plus simple de compiler un fichier `mon_fichier.ml` à l'aide d'OCamlBuild est, dans une ligne de commande, de taper

```
ocamlfind batteries/ocamlbuild fichier_de_destination
```

Cette commande invoque la version de OCamlBuild pour OCaml Batteries Included en lui indiquant que nous voulons que, à l'issue de la compilation, le fichier `fichier_de_destination` soit créé. Selon le nom du fichier de destination, la compilation produira soit un fichier exécutable, soit une des informations résumant le contenu du programme, soit une documentation sous la forme d'une page web, etc.

Dans tous les cas, OCamlBuild examine tous les fichiers OCaml du répertoire, détermine les fichiers qui sont nécessaires pour compiler vos fichiers, lesquels de ces fichiers ont été modifiés et doivent donc être recompilés, puis se charge des étapes de compilation et de liaison. En cas de problème de compilation, OCamlBuild affiche des messages d'erreur comparables à celles de la ligne de commande OCaml.

Pour la majorité des projets, cette utilisation d'OCamlBuild suffit. Nous verrons plus tard comment demander à OCamlBuild de ne construire que certaines parties d'un projet incomplet ou comment générer de la documentation depuis un fichier source.

## Édition de code source

Vous pouvez écrire vos programmes OCaml avec n'importe quel *éditeur de texte* pour langues occidentales. Par contre, Word ou autre *traitement de texte* ne vous servira à rien.

### Emacs 22

Emacs est un éditeur de texte pour programmeurs, extrêmement puissant, lui-même programmable et disposant de centaines d'extensions pour des langages de programmation ou des situations particulières. Emacs est disponible gratuitement sur à peu près toutes les plate-formes existantes.

Pour utiliser Emacs avec OCaml, vous aurez besoin de

- Emacs lui-même (téléchargement de la dernière version Windows -- en cas de doute, choisissez emacs-22.X-bin-i386.zip (http://ftp.gnu.org/gnu/emacs/windows/) [archive], téléchargement de la version MacOS X (http://aquamacs.org/download.shtml) [archive], si vous utilisez Linux, vous pouvez installer Emacs 22 depuis votre gestionnaire de paquets)
- l'extension Tuareg (téléchargement pour Windows ou MacOS X (http://www-rocq.inria.fr/~acohen/tuareg/) [archive], si vous utilisez Linux, vous pouvez installer Tuareg depuis votre gestionnaire de paquets).

Une fois que vous avez installé et configuré Emacs et Tuareg, Emacs reconnaîtra et lancera automatiquement Tuareg dès que vous ouvrirez un fichier dont l'extension est `.ml` ou `.mli`.

#### Exécuter un extrait depuis Emacs

Une fois Tuareg lancé, vous pouvez demander au mode interactif de OCaml d'exécuter une commande en plaçant votre curseur sur la commande et en appliquant la combinaison de touches Contrôle-C-E. La première fois que vous invoquerez le mode interactif, Emacs vous demandera quelle version d'OCaml vous voulez employer.

Répondez

```
ocamlfind batteries/ocaml
```

#### Compiler un projet depuis Emacs

De même, une fois Tuareg lancé, vous pouvez demander à Emacs de compiler les fichiers sur lesquels vous travaillez. Pour ce faire, appliquez la combinaison de touches Contrôle-C-C. La première fois que vous invoquerez le compilateur, Emacs vous demandera quel compilateur vous voulez employer.

Répondez

```
ocamlfind batteries/ocamlbuild XXXX
```

où `XXXX` est le nom du fichier compilé que vous souhaitez obtenir.

### VI / Vim

VI / Vim est un autre éditeur de textes pour programmeurs, tout aussi puissant, programmable, extensible et disponible.

Pour utiliser VI avec OCaml, vous aurez besoin de

- VI lui-même (version Windows (http://www.winvi.de/en/) [archive], version MacOS X (http://macvim.org/OSX/index.php) [archive], si vous utilisez Linux, vous pouvez installer Vim et/ou gvim depuis votre gestionnaire de paquets)
- l'extension OMlet (téléchargement pour Windows ou MacOS X (http://www.lix.polytechnique.fr/~dbaelde/productions/omlet.html) [archive], si vous utilisez Linux, vous pouvez installer OMLet depuis votre gestionnaire de paquets).

Si vous pensez que vous serez amené à éditer des fichiers en ssh sur un serveur, vim est un meilleur choix que emacs.

## Programmes complets

OCaml a une définition très simple de ce qu'est un programme complet : tout extrait de OCaml qui fonctionne en mode calculatrice, une fois sauvegardé dans un fichier avec l'extension .ml, est un programme complet valide.

Ainsi,

```
1 print_endline "Bonjour, le monde";;
```

est un programme complet. On peut même d'ailleurs supprimer le `;;`, qui n'est pas utile dans ce cas précis.

```
1 print_endline "Bonjour, le monde !"
```

Pour utiliser ce programme, commençons le sauvegarder, sous le nom `bonjour.ml`.

## Lancer un programme

Nous pouvons maintenant le lancer notre programme ; pour ce faire :

- ouvrez un terminal (sous Windows, le terminal s'appelle « Invite de commande ») ;
- allez dans le répertoire dans lequel vous avez sauvegardé votre fichier `bonjour.ml` en tapant `cd le/nom/du/répertoire` ;
- tapez.

```
$ ocamlfind batteries/ocaml bonjour.ml
```

(le symbole `$` est affiché par votre terminal, vous n'avez pas besoin de l'écrire — si vous êtes sous Windows, le symbole sera plutôt >)

vous verrez s'afficher

```
Bonjour, le monde !
```

Félicitations, vous venez de lancer votre premier programme OCaml.

## Compiler et exécuter un programme

Il existe en fait plusieurs manières de lancer un programme OCaml. La manière directe, que nous venons de voir, s'appelle l'*interprétation* : OCaml lit le programme, vérifie qu'il a un sens, puis l'exécute immédiatement. Il s'agit de la manière la plus immédiate de tester si un programme s'exécute correctement mais le prix à payer est que le programme s'exécute plus lentement qu'avec les autres méthodes. De plus, dans un certain nombre de cas, il n'est pas raisonnable de demander à l'utilisateur, à chaque fois qu'il veut lancer son programme, de taper une ligne de code aussi surprenante que

```
$ ocamlfind batteries/ocaml bonjour.ml
```

L'alternative consiste à *compiler* le programme, c'est-à-dire à demander à OCaml de transformer votre fichier `bonjour.ml` en un autre fichier que l'ordinateur comprendra directement. Pour ce faire, toujours dans un terminal, tapez l'une des lignes suivantes :

```
$ ocamlfind batteries/ocamlbuild bonjour.byte
```

ou

```
$ ocamlfind batteries/ocamlbuild bonjour.native
```

La première de ces lignes demande à OCamlBuild de construire `bonjour.byte`, c'est-à-dire un fichier exécutable portable, petit, rapide à compiler et théoriquement capable de fonctionner sur n'importe quel type d'ordinateur. La deuxième ligne demande à OCamlBuild de construire `bonjour.native`, c'est-à-dire un fichier exécutable optimisé pour un ordinateur précis et donc inutilisable sur tout autre type d'ordinateur, plus lent à compiler mais plus rapide à l'exécution.

Dans le reste de cet ouvrage, nous utiliserons généralement des fichiers portables (donc ici `bonjour.byte`).

Ce choix est purement arbitraire.

Dans les deux cas, sauf accident, vous verrez apparaître un indicateur de progression puis, à l'issue de la compilation, un message de la forme

```
Finished, 3 targets (0 cached) in 00:00:00.
```

Ceci signifie qu'OCamlBuild a du produire 3 fichiers (dont `bonjour.byte`), qu'aucun de ces fichiers n'avait déjà été produit lors d'une compilation précédente, et que le temps total de compilation a été arrondi à 0 secondes.

> **Anecdote** OCamlBuild est un peu optimiste dans son affichage des durées de compilation.

Pour lancer le programme compilé, sous MacOS X ou Linux, il suffit alors d'écrire, toujours dans le terminal :

```
$ ./bonjour.byte
```

ou, sous Windows

```
> bonjour.byte
```

De nouveau, le symbole `$` ou `<` est inscrit par votre terminal, ce n'est pas à vous de le recopier.

Vous obtiendrez alors

```
Bonjour, le monde !
```

Théoriquement, vous pourriez aussi ouvrir le programme depuis l'Explorateur de fichiers (sous Windows), le Finder (sous MacOS X), Konqueror, Nautilus ou votre gestionnaire de fichiers préféré (sous Linux), en ouvrant le répertoire qui contient `bonjour.byte` et en double-cliquant sur l'icône correspondante. Pour ce programme, qui se referme immédiatement dès qu'il a affiché une phrase, cela ne servirait à rien. Cependant, une fois que vous aurez doté votre programme d'une interface graphique ou textuelle, ce sera une manière de le lancer.

> **Note** Il existe encore d'autres manières d'exécuter un programme OCaml, notamment la possibilité de transformer votre fichier `bonjour.ml` en un *script exécutable*. Cette fonctionnalité n'est quasiment jamais utilisée car le programme ainsi lancé reste interprété et donc beaucoup plus lent qu'un programme compilé. Nous ne détaillerons donc pas la technique.

## Effets et résultats

Si nous avions tapé le contenu de `bonjour.ml` depuis le mode calculatrice, nous aurions obtenu quelque chose comme :

```
# print_endline "Bonjour, le monde !"
Bonjour, le monde !
```

```
- : unit = ()
```

Ici, comme nous venons de le voir, le lancement de `bonjour.ml`, `bonjour.byte` ou `bonjour.native` a uniquement causé l'affichage de

```
Bonjour, le monde !
```

De la même manière, dans le mode calculatrice, le nombre 5 est une expression valide, qui produira

```
# 5;;
- : int = 5
```

Par contre, si nous exécutons le programme correspondant (donc un fichier qui contiendra en tout et pour tout 5), rien ne s'affichera. Cela ne signifiera pas qu'il y a eu le moindre problème -- de fait, le programme se sera exécuté correctement et aura un résultat. Par contre, comme nous n'avons pas précisé dans le programme que nous voulions afficher ce résultat, le programme n'aura aucun effet visible.

Pour faire afficher le résultat, nous aurions du écrire le programme suivant :

```
1 print_int 5
```

À l'exécution, ce programme calculera 5 et aura pour effet d'afficher cette valeur. Comme `print_int` a pour type `int -> unit`, ce programme aura de plus pour résultat `()`, résultat qui ne sera lui-même pas affiché.

> **Vocabulaire** On désigne sous le nom d'*effet observable* ou, tout simplement, *effet* toute influence visible du programme sur le reste du monde. Ainsi, afficher un texte, ouvrir une boîte de dialogue, modifier un fichier ou envoyer un mail constituent autant d'effets. Par opposition, calculer le résultat d'une expression arithmétique (sans l'afficher) ou calculer la longueur d'une liste (sans l'afficher) n'a aucun effet. Par extension, on considère que lire un fichier, réagir à un clic ou télécharger une page web sont encore des effets.

## Précautions et limitations

### Séparateurs

Dans le mode calculatrice, le symbole `;;` sert à prévenir OCaml que vous avez fini de taper. Dans un fichier .ml, OCaml considère que vous avez fini de taper une fois qu'il atteint la fin du fichier. L'utilisation du `;;` est donc beaucoup plus rare dans un fichier .ml que dans le mode calculatrice. On s'en sert sert surtout pour séparer deux expressions.

Ainsi, le programme suivant est tout à fait légitime

```
1 5 + 2;;
2 print_endline "Bonjour, le monde"
```

Ce programme commencera par calculer le résultat de `5 + 2`, résultat qu'il n'affichera pas, puis il affichera le texte `"Bonjour, le monde"`.

### Fichiers portables

Sur le principe, les fichiers exécutables portables `.byte` que nous venons de voir et les fichiers modules portables `.cmo` que nous verrons bientôt sont comparables aux fichiers portables `.class` de Java.

En raison de priorités de développement différentes dans la conception du langage, les fichiers portables de OCaml sont malheureusement moins portables que leurs homologues en Java, puisque, s'ils sont compatibles avec tous les types d'ordinateur, ils ne sont compatibles qu'avec la version de OCaml et des bibliothèques de programmation qui ont servi à les construire.

L'une des conséquences de ces choix de développement est que, le plus souvent, un programme OCaml sera distribué soit sous la forme d'un exécutable natif optimisé, qui sera indépendant des bibliothèques installées, soit sous la forme de code source.

### À retenir

- N'importe quel code source OCaml peut être utilisé en tant que programme autonome.
- Un programme autonome n'affiche pas son résultat.
- Un programme autonome peut être compilé à l'aide de OCamlBuild ou exécuté à l'aide de `ocaml`.

## Modules

Comme la quasi-totalité des langages de programmation modernes, OCaml Batteries Included est fourni avec un grand nombre de structures de données et de fonctions de manipulation de données, sous la forme d'une *bibliothèque standard*. Ces structures et fonctions sont rassemblées selon leur rôle en *modules*. Ainsi, les fonctions qui servent à manipuler des listes forment le module `List`, les fonctions qui servent à afficher des informations à l'écran forment le module `Print`, les fonctions de manipulation de texte encodé en *Latin-1* (caractères européens) sont rassemblées dans le module `String`, les fonctions de manipulation de texte encodé en UTF-8 (caractères internationaux) sont rassemblées dans le module `UTF8`, etc. Au cours de cette section, nous allons voir comment utiliser les fonctions d'un module puis comment créer de nouveaux modules.

> **Note** En OCaml, les noms de modules commencent toujours par une majuscule. Ceci est une obligation du langage.

### Utilisation d'un module

De base, OCaml est fourni avec plus d'une centaine de fonctions de manipulation des listes. Ainsi, pour calculer la longueur d'une liste, on invoquera la fonction `List.length`, c'est-à-dire la fonction `length` du module `List` :

```
# List.length ;;
- : 'a list -> int = <fun>
```

De même, pour déterminer le dernier élément d'une liste, on emploiera la fonction `List.last`, c'est-à-dire la fonction `last` du module `List` :

```
# List.last ;;
- : 'a list -> 'a = <fun>
```

`List.length` et `List.last` sont des fonctions parfaitement ordinaires, qui peuvent être manipulées comme toutes les autres fonctions. Ainsi, on écrira :

```
# let longueur = List.length ;;
val longueur : 'a list -> int = <fun>
```

De la même manière, le module `List` définit un type `'a t`, qui est identique au type `'a list` et qui peut se manipuler comme tout autre type :

```
# type associations = (string * int) List.t;;
type associations = (string * int) Batteries.List.t
```

Nous reviendrons plus tard sur la raison pour laquelle OCaml a répondu `Batteries.List.t` et non `List.t`. Pour le moment, nous nous contenterons d'accepter que `'a Batteries.List.t` et `'a List.t` désignent bien les mêmes types.

Pour consulter la documentation sur un module, vous pouvez employer la directive `#man_module`. Comme toutes les directives, `#man_module` ne peut être utilisée que depuis le mode calculatrice, comme suit :

```
# #man_module "List";;
```

Vous pouvez aussi consulter directement la documentation d'une des valeurs ou d'un des types du module

```
# #man_value "List.length";;
# #man_type "List.t";;
```

Vous pouvez aussi consulter la liste des fonctions et types d'un module (sans documentation) à l'aide de la directive `#browse`, comme suit :

```
# #browse "List";;
module List :
  sig
    type 'a t = 'a list
    type 'a enumerable = 'a t
    type 'a mappable = 'a t
    val length : 'a list -> int
    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list
    val is_empty : 'a list -> bool
    val cons : 'a -> 'a list -> 'a list
    val first : 'a list -> 'a
    val last : 'a list -> 'a
    val at : 'a list -> int -> 'a
    val rev : 'a list -> 'a list
    val append : 'a list -> 'a list -> 'a list
    val rev_append : 'a list -> 'a list -> 'a list
    val concat : 'a list list -> 'a list
    val flatten : 'a list list -> 'a list
    val make : int -> 'a -> 'a list
    val init : int -> (int -> 'a) -> 'a list
    val iter : ('a -> unit) -> 'a list -> unit
    val iteri : (int -> 'a -> 'b) -> 'a list -> unit
    val map : ('a -> 'b) -> 'a list -> 'b list
    val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list
    val rev_map : ('a -> 'b) -> 'a list -> 'b list
    val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
    val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

```
val reduce : ('a -> 'a -> 'a) -> 'a list -> 'a
val max : 'a list -> 'a
val min : 'a list -> 'a
val iter2 : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
val fold_right2 :
  ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
val for_all : ('a -> bool) -> 'a list -> bool
val exists : ('a -> bool) -> 'a list -> bool
val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
val mem : 'a -> 'a list -> bool
val memq : 'a -> 'a list -> bool
val find : ('a -> bool) -> 'a list -> 'a
val find_exn : ('a -> bool) -> exn -> 'a list -> 'a
val findi : (int -> 'a -> bool) -> 'a list -> int * 'a
val find_map : ('a -> 'b option) -> 'a list -> 'b
val rfind : ('a -> bool) -> 'a list -> 'a
val filter : ('a -> bool) -> 'a list -> 'a list
val filter_map : ('a -> 'b option) -> 'a list -> 'b list
val find_all : ('a -> bool) -> 'a list -> 'a list
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
val index_of : 'a -> 'a list -> int option
val index_ofq : 'a -> 'a list -> int option
val rindex_of : 'a -> 'a list -> int option
val rindex_ofq : 'a -> 'a list -> int option
val unique : ?cmp:('a -> 'a -> bool) -> 'a list -> 'a list
val assoc : 'a -> ('a * 'b) list -> 'b
val assoc_inv : 'a -> ('b * 'a) list -> 'b
val assq : 'a -> ('a * 'b) list -> 'b
val mem_assoc : 'a -> ('a * 'b) list -> bool
val mem_assq : 'a -> ('a * 'b) list -> bool
val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
val remove_assq : 'a -> ('a * 'b) list -> ('a * 'b) list
val split_at : int -> 'a list -> 'a list * 'a list
val split_nth : int -> 'a list -> 'a list * 'a list
val remove : 'a list -> 'a -> 'a list
val remove_if : ('a -> bool) -> 'a list -> 'a list
val remove_all : 'a list -> 'a -> 'a list
val take : int -> 'a list -> 'a list
val drop : int -> 'a list -> 'a list
val take_while : ('a -> bool) -> 'a list -> 'a list
val takewhile : ('a -> bool) -> 'a list -> 'a list
val drop_while : ('a -> bool) -> 'a list -> 'a list
val dropwhile : ('a -> bool) -> 'a list -> 'a list
val interleave : ?first:'a -> ?last:'a -> 'a -> 'a list -> 'a list
val enum : 'a list -> 'a Enum.t
val of_enum : 'a Enum.t -> 'a list
val backwards : 'a list -> 'a Enum.t
val of_backwards : 'a Enum.t -> 'a list
val split : ('a * 'b) list -> 'a list * 'b list
val combine : 'a list -> 'b list -> ('a * 'b) list
val make_compare : ('a -> 'a -> int) -> 'a list -> 'a list -> int
val sort : ?cmp:('a -> 'a -> int) -> 'a list -> 'a list
val stable_sort : ('a -> 'a -> int) -> 'a list -> 'a list
val fast_sort : ('a -> 'a -> int) -> 'a list -> 'a list
val merge : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list
exception Empty_list
exception Invalid_index of int
exception Different_list_size of string
val t_of_sexp : (Sexplib.Sexp.t -> 'a) -> Sexplib.Sexp.t -> 'a t
val sexp_of_t : ('a -> Sexplib.Sexp.t) -> 'a t -> Sexplib.Sexp.t
val print :
  ?first:string ->
```

```
    ?last:string ->
    ?sep:string ->
    ('a IO.output -> 'b -> unit) ->
    'a IO.output -> 'b t -> unit
  val sprint :
    ?first:string ->
    ?last:string ->
    ?sep:string ->
    ('a IO.output -> 'b -> unit) -> 'b t -> string
  val nth : 'a list -> int -> 'a
  module Exceptionless :
    sig
      val rfind : ('a -> bool) -> 'a list -> 'a option
      val findi : (int -> 'a -> bool) -> 'a list -> (int * 'a) option
      val split_at :
        int ->
        'a list -> [ `Invalid_index of int | `Ok of 'a list * 'a list ]
      val at : 'a list -> int -> [ `Invalid_index of int | `Ok of 'a ]
      val assoc : 'a -> ('a * 'b) list -> 'b option
      val assoc_inv : 'a -> ('b * 'a) list -> 'b option
      val assq : 'a -> ('a * 'b) list -> 'b option
    end
  module Labels :
    sig
      val init : int -> f:(int -> 'a) -> 'a list
      val iter : f:('a -> unit) -> 'a list -> unit
      val iteri : f:(int -> 'a -> 'b) -> 'a list -> unit
      val map : f:('a -> 'b) -> 'a list -> 'b list
      val mapi : f:(int -> 'a -> 'b) -> 'a list -> 'b list
      val rev_map : f:('a -> 'b) -> 'a list -> 'b list
      val fold_left : f:('a -> 'b -> 'a) -> init:'a -> 'b list -> 'a
      val fold_right : f:('a -> 'b -> 'b) -> 'a list -> init:'b -> 'b
      val iter2 : f:('a -> 'b -> unit) -> 'a list -> 'b list -> unit
      val map2 : f:('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
      val rev_map2 : f:('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
      val fold_left2 :
        f:('a -> 'b -> 'c -> 'a) -> init:'a -> 'b list -> 'c list -> 'a
      val fold_right2 :
        f:('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> init:'c -> 'c
      val for_all : f:('a -> bool) -> 'a list -> bool
      val exists : f:('a -> bool) -> 'a list -> bool
      val for_all2 : f:('a -> 'b -> bool) -> 'a list -> 'b list -> bool
      val exists2 : f:('a -> 'b -> bool) -> 'a list -> 'b list -> bool
      val find : f:('a -> bool) -> 'a list -> 'a
      val find_exn : f:('a -> bool) -> exn -> 'a list -> 'a
      val findi : f:(int -> 'a -> bool) -> 'a list -> int * 'a
      val rfind : f:('a -> bool) -> 'a list -> 'a
      val filter : f:('a -> bool) -> 'a list -> 'a list
      val filter_map : f:('a -> 'b option) -> 'a list -> 'b list
      val find_all : f:('a -> bool) -> 'a list -> 'a list
      val partition : f:('a -> bool) -> 'a list -> 'a list * 'a list
      val remove_if : f:('a -> bool) -> 'a list -> 'a list
      val take_while : f:('a -> bool) -> 'a list -> 'a list
      val drop_while : f:('a -> bool) -> 'a list -> 'a list
      val stable_sort : ?cmp:('a -> 'a -> int) -> 'a list -> 'a list
      val fast_sort : ?cmp:('a -> 'a -> int) -> 'a list -> 'a list
      val merge : cmp:('a -> 'a -> int) -> 'a list -> 'a list -> 'a list
    end
```

**Vocabulaire** L'information qui vient d'être affichée sur le module `List` et qui commence par `sig` et s'achève par `end` s'appelle la *signature* du module. La notion de signature généralise aux modules la notion de *type* d'une valeur.

Vous constaterez que certaines fonctions du module `List` font appel à des types d'autres modules `Enum` et `IO`.

Nous nous intéresserons à ces modules dans des chapitres ultérieurs. Vous constaterez aussi que le module `List` contient deux sous-modules `Labels` et `Exceptionless`, dont nous parlerons plus tard. Pour le moment, il suffit de savoir qu'un module peut contenir un sous-module. Comme pour les valeurs et les types, le nom complet d'un sous-module est le nom du module qui le contient, suivi d'un point, suivi du nom local du sous-module. Ainsi, le module `Labels` que nous avons sous les yeux a pour nom complet `List.Labels` et le module `Exceptionless` aura pour nom complet `List.Exceptionless`.

### Ouverture de module

Considérons la fonction suivante :

```
# let second_half l =
    let l' = List.unique l in
    List.take (List.length l' / 2) l';;
val second_half : 'a list -> 'a list = <fun>
```

Cette fonction prend en argument une list `l`, en extrait la liste `l'` des valeurs uniques de `l` (c'est-à-dire supprime les doublons), puis calcule la longueur de `l'`, la divise par deux et se sert de cette valeur pour renvoyer la première moitié de la liste `l'`. Si cette fonction n'est probablement guère utile en pratique, nous pouvons constater qu'elle fait appel trois fois au préfixe `List.`, peut-être aux dépens de la lisibilité.

Il serait certainement plus agréable de n'avoir à préciser qu'une seule fois que nous manipulons des fonctions du module `List`. C'est à cela que sert la commande `open`. Nous l'utiliserons comme suit :

```
# open List;;
# let second_half l =
    let l' = unique l in
    take (length l' / 2) l';;
val second_half : 'a list -> 'a list = <fun>
```

Ou encore

```
# let second_half l =
    open List in
      let l' = unique l in
      take (length l' / 2) l';;
val second_half : 'a list -> 'a list = <fun>
```

Dans ces deux extraits, nous n'avons eu à mentionner `List` qu'une seule fois. Respectivement, ces extraits sont équivalents à :

```
# type 'a t  = 'a List.t
  let unique = List.unique
  let take   = List.take
  let length = List.length;;
  (* ... *)
# let second_half l =
    let l' = unique l in
    take (length l' / 2) l';;
val second_half : 'a list -> 'a list = <fun>
```

ou à

```
# let second_half l =
    let unique = List.unique
    and take   = List.take
    and length = List.length
    (* ... *)
    in
      let l' = unique l in
      take (length l' / 2) l';;
val second_half : 'a list -> 'a list = <fun>
```

Dans tous les cas, la fonction `second_half` obtenue est identique à celle que nous avions initialement. La seule différence est la notation plus légère.

> **Vocabulaire** L'opération `open List` est l'*ouverture* du module `List`. L'opération `open List in (*...*)` est l'*ouverture locale* du module `List`.

Pour des raisons de confort de notation, il est aussi possible d'ouvrir plusieurs modules en une seule opération. Ainsi, pour ouvrir le module `List` et le module `String`, on pourra écrire

```
# open List, String;;
```

On utilisera fréquemment ce mécanisme pour ouvrir en une opération le module `List` et son sous-module `Labels`, qui redéfinit certaines opérations de `List` pour permettre d'employer des notations plus robustes (mais plus lourdes), comme suit :

```
# open List, Labels;;
```

Nous reviendrons dans un chapitre ultérieur sur le rôle exact de ce module `Labels`.

### Le module Standard

Au cours des chapitres précédents, nous avons manipulé un certain nombre de fonctions sans préciser de quel module elles sont tirées : `print_endline`, `int_of_string`, `float_of_int` etc. Ces fonctions et bien d'autres viennent d'un module spécial nommé `Standard`, qui est automatiquement ouvert. Ce module définit aussi de nombreuses fonctions de gestion des erreurs, d'affichage, de manipulation de fichiers, de calculs sur les entiers ou les nombres flottants, etc.

Comme pour les autres modules, vous pouvez consulter le contenu du module `Standard` à l'aide de `#browse` ou `#man_module`.

### Précautions et limitations

#### Majuscules

Insistons sur le fait qu'un nom de module commence toujours par une majuscule. Ainsi, si vous écrivez `List.length`, OCaml sait qu'il doit chercher la fonction `length` du module `List` et, si nécessaire, préalablement charger ce module. À l'inverse, si vous écrivez `list.length`, OCaml sait qu'il doit chercher le champ `length` d'un enregistrement nommé `list`, ce qui est une opération beaucoup plus simple.

En particulier, les deux constructions produiront des messages différents :

```
# List.longueur;;
  ^^^^^^^^^^^^^
Error: Unbound value List.longueur
# Liste.longueur;;
  ^^^^^^^^^^^^^^
Error: Unbound value Liste.longueur
# liste.longueur;;
  ^^^^^
Error: Unbound value liste
```

#### Constructeurs et modules

De la même manière qu'un nom de valeur ou de type, un nom de constructeur doit être préfixé par le nom du module qui le contient.

Ainsi, le module `Unix` définit de nombreux constructeurs, un pour chaque erreur qui peut avoir lieu lors d'une interaction avec le système d'exploitation. Parmi ces erreurs, citons arbitrairement `EBUSY`, qui peut être invoquée lorsque le programme tente d'accéder à une ressource qui est déjà occupée par un autre logiciel. Comme nous pouvons le constater, le nom complet de cette ressource est `Unix.EBUSY`:

```
# EBUSY;;
  ^^^^^
Error: Unbound constructor EBUSY
# Unix.EBUSY;;
- : Batteries.Unix.error = Batteries.Unix.EBUSY

# open Unix;;

# EBUSY;;
- : Batteries.Unix.error = Batteries.Unix.EBUSY
```

Ceci est aussi valable pour les enregistrements. Ainsi, le module `Complex` définit une structure `type t = {re : float; im : float}`. Le nom complet des champs est en fait `Complex.re` et `Complex.im`. Pour nous en convaincre, voici un exemple

```
# {re = 5.0; im = 1.0};;
  ^^^^^^^^^^^^^^^^^^^
Error: Unbound record field label re

# {Complex.re = 5.0; Complex.im = 1.0};;
- : Batteries.Complex.t =
{Batteries.Complex.re = 5.; Batteries.Complex.im = 1.}

# open Complex;;

# {re = 5.0; im = 1.0};;
- : Batteries.Complex.t = {re = 5.; im = 1.}
```

#### Type t

Nous avons vu que le module `List` définit un type `'a t` (le type des listes contenant des éléments de type `'a`, c'est-à-dire un synonyme de `'a list`). De la même manière, le module `Int` définit un type `t` (le type des entiers, c'est-à-dire un synonyme de `int`), le module `Rope` définit un type `t` (le type des textes contenant des

caractères internationaux au standard Unicode), etc. De fait, une très grande partie des modules de OCaml Batteries Included définissent un type `t`. Il s'agit d'une convention de nommage simple : si un module est conçu spécialement pour accueillir une structure de données, le nom du type correspondant est `t`.

Afin d'éviter de confondre tous ces types qui portent le même nom, on prendra l'habitude d'utiliser le nom complet de chaque type, donc `Rope.t`, etc.

#### Batteries

Revenons sur un de nos exemples :

```
# type associations = (string * int) List.t;;
type associations = (string * int) Batteries.List.t
```

Comme nous pouvons le constater, OCaml a accepté cette définition mais la réponse mentionne `Batteries.List.t` et non `List.t`. Nous laissons au lecteur le soin de vérifier que ces deux types sont bien identiques.

À quoi est du ce renommage ?

De fait, le nom complet du module `List` est `Batteries.List`, ce qui signifie que le module `List` fait partie du module `Batteries`. Ce dernier est un module spécial, qui contient tous les autres modules de OCaml Batteries Included, y compris `Standard` et dont vous n'avez jamais besoin de taper le nom. Nous aurions cependant tout aussi bien pu taper :

```
# type associations = (string * int) Batteries.List.t;;
type associations = (string * int) Batteries.List.t
```

En général, nous éviterons de mentionner explicitement `Batteries`, pour de simples raisons de lisibilité.

#### Ouverture et écrasement

Comme nous l'avons mentionné plus haut, l'ouverture d'un module définit implicitement de nombreuses valeurs. Ainsi, nous aurons

```
# let length = 6;;
val length : int = 6
# length;;
- : int = 6
# open List;;
# length;;
- : 'a list -> int = <fun>
```

Dans cet extrait, la constante `length` a été écrasée par `List.length` lors de l'ouverture. Si ce comportement est parfois désiré, il s'agit aussi d'une source d'erreurs assez fréquente. Pour éviter tout accident, on préférera fréquemment employer l'ouverture locale, moins fragile puisqu'elle ne déborde pas d'un domaine visible au premier coup d'œil.

## Exercice

1. Formulez une expérience pour permettre de vérifier que le type `'a List.t` est bien le même que le

type `'a Batteries.List.t`.
2. Écrivez une fonction d'une seule ligne qui renvoie le deuxième élément d'une liste. Pour ce faire, vous utiliserez uniquement les fonctions `List.hd` et `List.tl`.

## Créer de nouveaux modules

Bien entendu, il est possible d'écrire de nouveaux modules et de les utiliser comme les modules d'OCaml Batteries Included. Ceci se fait en deux étapes : définir le *corps* du module (c'est-à-dire les fonctionnalités du module) puis, si nécessaire, sa *signature* (c'est-à-dire les restrictions sur l'usage du module).

Commençons par nous intéresser au corps du module.

Créer un nouveau module peut se faire aussi simplement que d'écrire un programme complet. De fait, la seule différence entre un programme complet et un module est la manière de le compiler. Ainsi, pour créer un module on commencera par sauvegarder du code source dans un fichier d'extension .ml, que l'on compilera vers un fichier d'extension .cmo (pour obtenir un module portable non-optimisé) ou .cmx (pour obtenir un module portable optimisé). De plus, si vous ne souhaitez utiliser votre module que dans un programme compilé, vous n'avez pas besoin de le compiler du tout, OCamlBuild s'en chargera pour vous.

Ainsi, si nous souhaitons placer dans un module les fonctions que nous avons définies précédemment sur les couleurs de cartes `couleur.ml`. Comme ce module servira uniquement à accueillir une structure de données qui permettra de manipuler les couleurs de cartes, profitons-en pour renommer notre type de `couleur_de_carte` en `t`. Pour respecter les conventions de nommage habituelles des modules, nous en profiterons pour renommer quelques fonctions.

```
type t =
   | Trefle
   | Pique
   | Coeur
   | Carreau ;;

let to_string = function
   | Trefle -> "trefle"
   | Pique  -> "pique"
   | Coeur  -> "coeur"
   | Carreau-> "carreau" ;;
```

De la même manière, nous pouvons définir un nouveau module qui contiendra les fonctions de manipulation des cartes elles-mêmes. Appelons le fichier `cartes.ml`. Ce module utilisera le module `Couleur`, défini à l'aide du fichier précédent. Profitons-en de même pour renommer `t` notre type de cartes.

```
type t =
  | Atout    of int
  | Roi      of Couleur.t
  | Dame     of Couleur.t
  | Cavalier of Couleur.t
  | Valet    of Couleur.t
  | As       of Couleur.t
  | Nombre   of int * Couleur.t ;;

let string_of_chiffre_francais = function
   | 1 -> "Un"
   | 2 -> "Deux"
   | 3 -> "Trois"
   | 4 -> "Quatre"
```

```
   15    | 5 -> "Cinq"
16    | 6 -> "Six"
17    | 7 -> "Sept"
18    | 8 -> "Huit"
19    | 9 -> "Neuf"
20    | 0 -> "Zero"
21    | n -> string_of_int n ;;
22
23 let to_string carte =
24   open Couleur in match carte with
25   | Atout    i    -> string_of_int i ^ " d'atout"
26   | Roi      c    -> "Roi de "       ^ ( to_string c ) (*Notre définition de to_string
27   | Dame     c    -> "Dame de "      ^ ( to_string c )
28   | Cavalier c    -> "Cavalier de "  ^ ( to_string c )
29   | Valet    c    -> "Valet de "     ^ ( to_string c )
30   | As       c    -> "As de "        ^ ( to_string c )
31   | Nombre (i, c) -> string_of_chiffre_francais i ^ " de " ^ ( to_string c ) ;;
32
33 let get_couleur = function
34     | As       c -> Some c
35     | Roi      c -> Some c
36     | Valet    c -> Some c
37     | Dame     c -> Some c
38     | Cavalier c -> Some c
39     | Nombre _ c -> Some c
40     | _          -> None ;;
```

Si nous le souhaitons, nous pouvons maintenant compiler nos modules :

```
$ ocamlfind batteries/ocamlbuild couleur.cmo carte.cmo
Finished, 4 targets (0 cached) in 00:00:02.
```

> **Note** L'ordre `couleur.cmo` `carte.cmo` est sans importance. Si un module dépend d'un autre, OCamlBuild compilera les modules dans l'ordre de dépendances.

Rappelons que nous n'avons pas besoin de compiler les modules nous-mêmes, à moins d'avoir envie de nous en servir en mode interactif, car OCamlBuild les compilera lui-même si nécessaire.

Testons maintenant nos modules dans un programme :

```
1 (*Fichier test.ml*)
2
3 let premiere_carte_qui_porte_malheur = Carte.As Couleur.Pique;;
4 let deuxieme_carte_qui_porte_malheur = Carte.Atout 16;;
5
6 print_endline ("La première carte qui porte malheur est "^Carte.to_string premiere_carte
7 print_endline ("La deuxième carte qui porte malheur est "^Carte.to_string deuxieme_carte
```

Compilons et testons :

```
$ ocamlfind batteries/ocamlbuild test.byte
Finished, 7 targets (4 cached) in 00:00:01.
$ ./test.byte
La première carte qui porte malheur est As de pique
La deuxième carte qui porte malheur est 16 d'atout
```

### Charger un module

Une fois qu'un fichier .ml a été compilé sous la forme d'un `.cmo`, nous pouvons le charger dans le mode interactif. Pour ce faire, nous utiliserons les directives `#cd` (pour aller dans le répertoire qui contient le fichier `.cmo`) et `#load` (pour charger effectivement le fichier `.cmo`). Si un `.cmo` dépend d'un autre `.cmo`, les fichiers doivent être chargés par ordre de dépendance.

Par défaut, OCamlBuild sauvegarde les fichiers `.cmo` dans un répertoire nommé `_build`. Nous pouvons donc faire

```
(*Nous n'avons pas encore chargé le module Carte*)
# Atout 16;;
  ^^^^^
Error: Unbound constructor Atout

# Carte.Atout 16;;
  ^^^^^^^^^^^
Error: Unbound constructor Carte.Atout

(*Chargeons les deux modules*)
# #cd "_build";;

# #load "couleur.cmo";;

# #load "carte.cmo";;

# Atout 16;;
  ^^^^^
Error: Unbound constructor Atout

# Carte.Atout 16;;
- : Carte.t = Carte.Atout 16
```

Comme l'indique ce dernier extrait, charger un fichier `.cmo` n'*ouvre* pas automatiquement le module. Il est donc possible de charger un module sans crainte d'écraser les noms de fonctions ou de types existants.

Une autre manière d'ouvrir un ou plusieurs fichiers .cmo est de donner la liste sur la ligne de commande

```
$ ocamlfind batteries/ocaml couleur.cmo carte.cmo
#         Objective Caml version 3.11.0

        ___________________________________
      |        | |                         |
    [| +      | | Batteries Included  - |
      |________|_|_________________________|
        ___________________________________
      |                        | |        |
      | -     Type '#help;;'   | |      + |]
      |________________________|_|________|


# Carte.Atout 16;;
- : Carte.t = Carte.Atout 16
```

Vous disposez maintenant de toutes les connaissances nécessaires pour créer et utiliser un module. Dans la section suivante, nous allons voir comment et pourquoi restreindre l'accès à un module.

> **Vocabulaire** Indépendamment du langage de programmation, le fait de concevoir un programme comme un ensemble de modules dotés de dépendances simples et précises s'appelle la *modularité*.

## Précautions et limitations

### Un programme est un module

Insistons sur un point : un module peut contenir exactement le même genre de choses qu'un programme. Le fichier suivant, sauvegardé sous le nom `bonjour.ml` peut donc être compilé comme un module tout à fait acceptable :

```
1 print_endline "Bonjour, le monde !";;
```

Il s'agit d'un module qui ne définit aucune fonction, aucun type, mais qui, lorsqu'il est utilisé par un programme, affiche `Bonjour, le monde`.

Pour nous en convaincre, commençons par le compiler :

```
$ ocamlfind batteries/ocamlbuild bonjour.cmo
Finished, 3 targets (0 cached) in 00:00:00.
```

Puis chargeons ce fichier en mode interactif

```
# #cd "_build";;
# #load "bonjour.cmo";;
Bonjour, le monde !
#
```

### Ordre de chargement

Insistons sur un point important : si les fichiers `.cmo` peuvent être compilés dans n'importe quel ordre, ils doivent être chargés par ordre de dépendance, c'est-à-dire en ne chargeant que des fichiers `.cmo` qui dépendent de modules déjà chargés.

Ainsi, l'extrait suivant charge les modules dans le bon sens :

```
# #cd "_build";;
# #load "couleur.cmo";;
# #load "carte.cmo";;
```

Par contre, si nous essayons de charger `carte.cmo` avant `couleur.cmo`, nous obtiendrons :

```
# #cd "_build";;
# #load "carte.cmo";;
Error: Reference to undefined global `Couleur'
```

Ce choix de conception du langage permet de garantir qu'un programme qui se charge peut aussi s'exécuter. Par contraste, des langages tels que Python offrent plus de souplesse au niveau du chargement mais un programme une fois lancé peut encore s'arrêter en catastrophe lorsque des modules s'avèrent manquants.

### Dépendances cycliques

Une conséquence directe de cette contrainte sur l'ordre de chargement est qu'il n'est pas possible, en OCaml, d'écrire deux modules qui dépendent chacun de l'autre. Ainsi, il est impossible de compiler les deux fichiers suivants ensemble :

```
1 (*Fichier a.ml*)
2 let a = 5;;
3 let b = B.b;;
```

```
1 (*Fichier b.ml*)
2 let a = A.a;;
3 let b = 6;;
```

Si nous essayons de compiler l'un ou l'autre, nous obtiendrons

```
$ ocamlfind batteries/ocamlbuild a.cmo
Circular build detected (b.cmi already seen in [ a.cmi; b.cmi; a.cmo ])
Compilation unsuccessful after building 2 targets (0 cached) in 00:00:00.
```

**Vocabulaire** On appelle ce type de problème une *dépendance cyclique*.

Contrairement à ce qui est autorisé en Java ou en Python, il est impossible d'écrire un programme ou un module contenant des dépendances cycliques en OCaml. Cette limitation est liée directement à la contrainte sur l'ordre de chargement et au fait qu'un module peut contenir du code qui sera exécuté lors de son chargement. Si un programme contenait deux modules `A` et `B` mutuellement dépendants, ni le code de l'un ni le code de l'autre ne pourrait s'exécuter lors du chargement, puisque le chargement de chacun des deux nécessiterait le chargement préalable de l'autre.

Il est généralement admis en informatique qu'une dépendance cyclique entre deux sous-ensembles distincts d'un programme est une erreur de conception. Il existe une plusieurs techniques pour transformer un projet contenant des dépendances cycliques de manière à supprimer ces dépendances. La technique la plus simple, qui n'est pas toujours envisageable ou satisfaisante, consiste à isoler les fonctionnalités qui introduisent ces dépendances cycliques et à les rassembler dans un module propre, puis à masquer cette technique d'implantation à l'aide de la signature appropriée.

Une autre technique consiste à introduire de la récursivité mutuelle entre modules à l'aide d'un opérateur de point fixe sur module. Nous verrons cette technique bien plus tard, lors du chapitre consacré aux usages avancés des modules.

## Exercices

1. Définissez un module `Ast`, qui regroupera les types `programme`, `instruction`... et les fonctions `string_of_programme`, `string_of_instruction` définis au chapitre précédent.

## Contraindre l'utilisation d'un module

Dans ce qui précède, nous avons vu comment créer des modules. Si ceci est déjà très utile, OCaml propose quelques concepts supplémentaires qui serviront notamment à documenter l'utilisation d'un module ou à la restreindre l'utilisation d'un module. Cette notion de restriction sur l'utilisation a de nombreux usages. On s'en servira d'une part pour garantir au développeur du module que des changements dans le fonctionnement interne du module n'interfèreront pas avec le fonctionnement des programmes qui utilisent ce module, et

d'autre part pour garantir que l'utilisateur ne peut pas, par accident, introduire des valeurs incorrectes ou contraires aux hypothèses, telles qu'un 25 d'Atout dans notre jeu de tarot.

Pour introduire ces restrictions, on emploiera des *signatures* (ou *interfaces*).

### Signatures

Un peu plus tôt dans ce chapitre, nous avons rencontré la signature du module `List`. À l'aide de la directive `#browse`, nous avions demandé à OCaml de nous afficher la liste des fonctions et types de ce module. Une signature n'est rien d'autre qu'une telle liste, sauvegardée dans un fichier d'extension `.mli` . Si nous désirons compiler une signature, ce qui n'est jamais nécessaire mais peut être utile le temps d'un test, il nous suffit de demander à OCamlBuild de produire un fichier d'extension `.cmi`

Ainsi, pour notre exemple du Tarot, nous allons commencer par écrire le fichier `couleur.mli`, qui accompagnera le fichier `couleur.ml`.

> **Note** OCaml cherchera toujours la signature correspondant à un fichier `toto.ml` dans le fichier correspondant `toto.mli`.

```
1 type t =
2    | Trefle
3    | Pique
4    | Coeur
5    | Carreau ;;
6
7 val to_string : t -> string ;;
```

De la même manière, nous pouvons écrire le fichier `carte.mli`, qui accompagnera le fichier `carte.ml` :

```
 1 type t =
 2   | Atout    of int
 3   | Roi      of Couleur.t
 4   | Dame     of Couleur.t
 5   | Cavalier of Couleur.t
 6   | Valet    of Couleur.t
 7   | As       of Couleur.t
 8   | Nombre   of int * Couleur.t ;;
 9
10 val string_of_chiffre_francais : int -> string ;;
11
12 val to_string : t -> string ;;
13
14 val get_couleur : t -> t option ;;
```

À ce point, nous pourrions nous arrêter : nous disposons d'une interface complète pour chacun de nos deux modules, interface qui ne restreint rien. De fait, accordons-nous une petite pause, le temps de tester que tout ceci compile correctement :

```
$ ocamlfind batteries/ocamlbuild test.byte
Finished, 11 targets (3 cached) in 00:00:01.
```

Mais nous pouvons faire mieux.

Commençons par cette fonction `string_of_chiffre_francais`. Elle est certes nécessaire pour écrire la

fonction `to_string` mais sa présence est plutôt embarrassante dans un module consacré à des cartes de tarot. Faisons-la donc simplement disparaître.

Voici donc une nouvelle version de `carte.mli` :

```
 1 type t =
 2   | Atout    of int
 3   | Roi      of Couleur.t
 4   | Dame     of Couleur.t
 5   | Cavalier of Couleur.t
 6   | Valet    of Couleur.t
 7   | As       of Couleur.t
 8   | Nombre   of int * Couleur.t ;;
 9
10 val to_string : t -> string ;;
11
12 val get_couleur : t -> Couleur.t option ;;
```

Comme nous pouvons le constater, cette fonction n'était effectivement pas nécessaire pour les utilisateurs du module `Carte` :

```
$ ocamlfind batteries/ocamlbuild test.byte
Finished, 11 targets (6 cached) in 00:00:01.
```

Maintenant que nous l'avons supprimée de l'interface du module, cette fonction ne peut plus être manipulée par les utilisateurs de ce module :

```
# #cd "_build" ;;

# #load "couleur.cmo" ;;

# #load "carte.cmo" ;;

# Carte.to_string ;;
- : Carte.t -> string = <fun>

# Carte.string_of_chiffre_francais ;;
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: unbound value Carte.string_of_chiffre_francais
```

Lorsque nous avons supprimé la fonction de l'interface du module, nous avons interdit à tout utilisateur de la manipuler. Par contre, cette fonction est toujours présente dans le module, car elle est nécessaire à `Carte.to_string`, fonction dont nous venons de vérifier qu'elle est toujours présente.

> **Vocabulaire** La possibilité de cacher certaines parties d'un module pour empêcher leur utilisation à l'extérieur du module est une forme limitée d'*encapsulation*.

Maintenant que nous avons légèrement nettoyé l'interface, intéressons-nous un peu plus au cas des Atouts : un Atout est une carte dont le numéro est compris entre 1 et 22 -- ou entre 0 et 21, le choix doit être fait lors de la conception du module. À l'heure actuelle, rien ne m'interdit d'introduire, par accident, une valeur `Atout 25`. C'est malheureux et cela peut provoquer des bugs. Fort heureusement, nous pouvons corriger ce problème.

Pour ce faire, nous allons commencer par modifier l'implantation pour faire en sorte que le constructeur `Atout` n'utilise pas des entiers mais des valeurs d'un nouveau type, que nous allons appeler `atout` :

```
1 (*Fichier carte.ml*)
2 type t =
3   | Atout    of atout
4   | Roi      of Couleur.t
5   | Dame     of Couleur.t
6   | Cavalier of Couleur.t
7   | Valet    of Couleur.t
8   | As       of Couleur.t
9   | Nombre   of int * Couleur.t ;;
```

Bien entendu, pour que ceci compile, il faut que nous ayons préalablement défini le type `atout`. Comme nous souhaitons que les valeurs d'atouts soient bien des entiers, nous allons faire de `atout` un synonyme de `int` -- pour le moment.

```
1 (*Fichier carte.ml*)
2 type atout = int;;
```

Qu'avons-nous gagné pour le moment ? En apparence rien. En fait, l'intérêt de cette manipulation est que, si nous avons bien défini que le type `atout` est un synonyme de `int`, nous ne sommes pas obligés de rendre cette information publique. Ainsi, dans `carte.mli`, nous allons écrire, sans plus de détails :

```
 1 (*Fichier carte.mli*)
 2 type atout;;
 3
 4 type t =
 5   | Atout    of atout
 6   | Roi      of Couleur.t
 7   | Dame     of Couleur.t
 8   | Cavalier of Couleur.t
 9   | Valet    of Couleur.t
10   | As       of Couleur.t
11   | Nombre   of int * Couleur.t ;;
```

> **Vocabulaire** Un type dont le nom est donné dans l'interface mais dont la définition complète reste cachée est appelé un *type abstrait*.

En rendant le type `atout` abstrait, nous avons interdit à l'utilisateur de fabriquer lui-même des valeurs de ce type. Il nous reste à définir deux fonctions triviales qui permettront de transformer un `atout` en `int`, ce qui peut servir pour comparer des valeurs d'atouts, ou pour transformer un entier compris entre 1 et 22 en `atout` :

```
1 (*Fichier carte.ml*)
2
3 let int_of_atout x = x;;
4
5 let atout_of_int x =
6  if 1 <= x && x <= 22 then Some x
7  else                      None;;
```

Quel type donner à ces fonctions ? La fonction `int_of_atout` pourrait avoir pour type `'a -> 'a`, tandis que `atout_of_int` pourrait avoir pour type `int -> int option`. Mais, comme nous avons en tête l'idée que la première de ces fonctions sert à transformer un `atout` en nombre entier et que la deuxième de ces fonctions sert à transformer un nombre entier en `atout`, nous allons leur donner les types suivants :

```
1 (*Fichier carte.mli*)
2
3 val int_of_atout : atout -> int ;;
4
5 val atout_of_int : int -> atout option ;;
```

Nous disposons maintenant de deux fonctions qui permettent de manipuler les valeurs de type `atout`. Tant que nous n'ajoutons pas d'autre fonction qui renvoie une valeur de type `atout`, nous pouvons être certains que, <emph>par construction et malgré toute la mauvaise volonté du monde, aucun utilisateur n'arrivera à avoir un `atout` dont la valeur n'est pas comprise entre 1 et 22</emph>.

> **Vocabulaire** La valeur d'un élément de type `atout` est dite *sure par construction*, puis qu'il n'existe aucune manière pour un utilisateur de violer sa propriété fondamentale, qui est d'être comprise entre 1 et 22.
>
> **Note** En programmation fonctionnelle, on considère comme primordial cet question de sûreté par construction. L'essentiel du temps passé à concevoir une bibliothèque de programmation fonctionnelle est généralement consacré à essayer d'obtenir le plus de garanties de sûreté.
>
> **Note** Cette question de sûreté par construction est un très proche parent des *capacités par objets* présentes dans quelques langages de programmation conçus pour la sûreté (dont une variante de OCaml).
>
> **Vocabulaire** Profitons-en pour préciser que, indépendamment du langage de programmation, une bibliothèque est dite *sûre* s'il n'y a pas moyen de causer de comportements imprévisibles à l'aide de cette bibliothèque.
>
> **Vocabulaire** Un langage est dit *sûr* s'il est possible de programmer des bibliothèques sûres. On considère généralement que le langage C et les langages dynamiques ne sont pas sûrs, puisqu'il y a moyen de détourner totalement le fonctionnement d'une bibliothèque pour lui faire adopter un comportement imprévisible. Java, C#, OCaml, Haskell ou Coq seront des langages beaucoup plus sûrs. Entre eux, on considère généralement Java comme moins sûr que C#, lui-même moins sûr que OCaml, lui-même moins sûr que Haskell, lui-même moins sûr que Coq.

Nous pouvons encore améliorer légèrement les choses en autorisant l'utilisateur à utiliser soit 0 soit 22 pour l'excuse (carte qui, rappelons-le, peut avoir deux valeurs distinctes) mais en nous débrouillant pouru que les valeurs employées en internes soient toujours comprises entre 1 et 22. Pour ce faire, il nous suffit de remplacer la définition de `atout_of_int` par

```
1 (*Fichier carte.ml*)
2 let atout_of_int x =
3  if x = 0 then                  Some 22
4  else if 1 <= x && x <= 22 then Some x
5  else                           None;;
```

De la même manière que nous avons introduit un type `atout` et des fonctions `atout_of_int` et `int_of_atout` pour les atouts dont la valeur doit être comprise entre 1 et 22 (ou 0 et 22), nous allons introduire un type `nombre` et des fonctions `nombre_of_int` et `int_of_nombre` pour les cartes numérotées, dont la valeur doit être comprise entre 2 et 10.

```
1 (*Fichier carte.mli*)
2 type nombre;;
```

```
   3
 4 type t =
 5   | Atout    of atout
 6   | Roi      of Couleur.t
 7   | Dame     of Couleur.t
 8   | Cavalier of Couleur.t
 9   | Valet    of Couleur.t
10   | As       of Couleur.t
11   | Nombre   of nombre * Couleur.t ;;
12
13 val nombre_of_int : int -> nombre option ;;
14
15 val int_of_nombre : nombre -> int ;;
```

```
 1 (*Fichier carte.ml*)
 2 type nombre = int;;
 3
 4 type t =
 5   | Atout    of atout
 6   | Roi      of Couleur.t
 7   | Dame     of Couleur.t
 8   | Cavalier of Couleur.t
 9   | Valet    of Couleur.t
10   | As       of Couleur.t
11   | Nombre   of nombre * Couleur.t ;;
12
13 let int_of_nombre x = x;;
14
15 let nombre_of_int x =
16   if 2 <= x && x <= 10 then Some x
17   else                      None;;
```

> **Note** Il n'est pas nécessaire de déclarer les types et valeurs dans le même ordre dans le fichier `.ml` et dans le fichier `.mli` correspondant.

Et voilà !

### Documentation

L'habitude veut que tous les fichiers `.mli` soient documentés à l'aide de commentaires écrits selon quelques conventions simples. Ces commentaires pourront plus tard être extraits et transformés en documentation HTML ou PDF. Pour le moment, contentons-nous de quelques annotations, que nous complèterons un peu plus tard lorsque nous aurons introduit les outils d'extraction.

Commençons par le fichier `couleur.mli` :

```
 1 (**
 2  Gestion des couleurs de cartes à jouer.
 3 *)
 4
 5 (**La couleur d'une carte.
 6    Chaque carte hormis les Atouts est doté d'une couleur de carte.*)
 7 type t =
 8    | Trefle
 9    | Pique
10    | Coeur
11    | Carreau ;;
12
13 val to_string : t -> string ;;
```

```
14 (**Renvoie une représentation de la couleur de la carte, sous la forme d'un texte el
```

Puis le fichier `carte.mli` :

```
1 (*Fichier carte.mli*)
2
3 type atout (**La valeur d'un atout.
4               Cette valeur peut être transformée en un entier compris entre 1 et 22. L
5
6 val int_of_atout : atout -> int
7 (**Renvoie le nombre correspondant à la valeur d'un Atout.
8    Ce nombre est un entier compris entre 1 inclus et 22 inclus. L'Excuse vaut 22.*)
9
10 val atout_of_int : int -> atout option
11 (**Convertit un nombre en l'Atout correspondant.
12    Le nombre doit être un entier compris entre 0 et 22 inclus. Les valeurs 0 et 22 rep
13    Cette fonction renvoie None si le nombre n'est pas compris entre 0 et 22.*)
14
15 type nombre (**La valeur d'une carte numérotée.
16               Cette valeur peut être transformée en un entier compris entre 2 et 10. L
17
18 val int_of_nombre : nombre -> int
19 (**Renvoie le nombre correspondant à la valeur d'une carte numérotée.
20    Ce nombre est un entier compris entre 2 inclus et 10 inclus.*)
21
22 val nombre_of_int : int -> nombre option
23 (**Convertit un entier en la valeur de la carte numérotée correspondante.
24    Le nombre doit être un entier compris entre 2 et 10 inclus.
25    Cette fonction renvoie None si le nombre n'est pas compris entre 2 et 10 inclus.*)
26
27 (** Une carte de Tarot*)
28 type t =
29   | Atout    of atout
30   | Roi      of Couleur.t
31   | Dame     of Couleur.t
32   | Cavalier of Couleur.t
33   | Valet    of Couleur.t
34   | As       of Couleur.t
35   | Nombre   of nombre * Couleur.t (**Une carte numérotée (l'As est considéré comme un
36
37 val to_string : t -> string ;;
38 (**Renvoie une représentation e la carte sous la forme d'un texte en français.*)
39
40 val get_couleur : t -> Couleur.t option ;;
41 (**Renvoie la couleur d'une carte. Si la carte est un atout, elle n'a pas de couleur.*
```

### Types privés

Nous venons de voir la notion de types abstraits, qui nous a permis de garantir que les cartes numérotées et les atouts étaient forcément des valeurs valides. Le mécanisme consiste à masquer toute information sur la manière dont est implanté un type, et à ne fournir que des fonctions sûres pour la manipulation des valeurs de ce type.

Dans certains cas, cela semble un peu exagéré. On pourrait préférer laisser publique l'information sur la manière dont est implanté un type et de se contenter de restreindre la possibilité de *créer* des valeurs de ce type. Ainsi, dans notre exemple en cours, la seule manière d'utiliser une valeur de type `atout` ou `nombre` est de commencer par la convertir en entier à l'aide de `int_of_atout` ou `int_of_nombre`. Nous gagnerions en simplicité (et peut-être en vitesse) si nous pouvions autoriser un programmeur à utiliser toute valeur de type `nombre` ou `atout` directement comme un `int`, à condition de disposer d'un mécanisme pour protéger la

création des valeurs de type `nombre` ou `atout`.

C'est le principe des *types privés*.

Ainsi, dans `carte.mli`, nous pouvons remplacer

```
type atout (**La valeur d'un atout.
              Cette valeur peut être transformée en un entier compris entre 1 et 22. L'Ex

val int_of_atout : atout -> int
(**Renvoie le nombre correspondant à la valeur d'un Atout.
   Ce nombre est un entier compris entre 1 inclus et 22 inclus. L'Excuse vaut 22.*)

type nombre (**La valeur d'une carte numérotée.
              Cette valeur peut être transformée en un entier compris entre 2 et 10.
              L'As n'est pas représenté comme un nombre mais comme une figure.*)

val int_of_nombre : nombre -> int
(**Renvoie le nombre correspondant à la valeur d'une carte numérotée.
   Ce nombre est un entier compris entre 2 inclus et 10 inclus.*)
```

par

```
type atout = private int
 (**La valeur d'un atout.
    Il s'agit d'un entier compris entre 1 et 22. L'Excuse vaut donc 22.*)

type nombre = private int
  (**La valeur d'une carte numérotée.
     Il s'agit d'un entier compris entre 2 et 10.
     L'As n'est pas représenté comme un nombre mais comme une figure.*)
```

Dans le fichier `carte.ml`, nous pouvons de même nous débarrasser de `int_of_nombre` et `int_of_atout`, devenus inutiles.

Pour nous convaincre que ceci fonctionne correctement, commençons par compiler le tout :

```
$ ocamlfind batteries/ocamlbuild carte.cmo couleur.cmo
Finished, 6 targets (2 cached) in 00:00:00.
$ ocamlfind batteries/ocaml
```

Puis chargeons et testons dans le mode interactif :

```
# #cd "_build";;
# #load "couleur.cmo";;
# #load "carte.cmo";;

(*Un atout s'affiche comme un nombre.*)
# Carte.atout_of_int 5;;
- : Carte.atout option = Some 5

# Carte.atout_of_int 0;;
- : Carte.atout option = None


(*Mais un nombre n'est pas un atout.*)
# Carte.Atout 5;;
```

```
              ^
Error: This expression has type int but is here used with type
         Carte.atout = Carte.atout
```

Et voilà.

## À titre de comparaison

### Types abstraits

La notion de type abstrait apparait, sous une forme légèrement différente, dans la majorité des langages de programmation orientée objet statiquement typés.

Ainsi, nous pouvons considérer qu'un type abstrait est un type tel que :

- il est impossible de construire directement une valeur de ce type
- il est impossible de se servir d'informations sur la manière dont le type est effectivement implanté.

Dans un langage à objets, ces caractéristiques se traduiront par :

- un type abstrait est la donnée d'une interface d'objets
- la ou les classes qui implantent cette interface sont protégées.

En Java, pour implanter la première version de notre type `nombre`, nous écrirons donc :

```java
//Fichier Nombre.java
public interface Nombre
{
   public int getInt();
}

//Fichier NombreImpl.java
protected class NombreImpl implements Nombre
{
  private final int valeur;
  protected NombreImpl(int valeur)
  {
     this.valeur = valeur;
  }
  public int getInt()
  {
    return this.valeur;
  }
}
```

En plus de ceci, de la même manière que nous avons fourni une fonction `nombre_of_int` en OCaml, nous devons fournir une fonction ou une méthode *usine* en Java, telle que :

```java
//Fichier NombreFactory.java
public class NombreFactory
{
  public static Nombre makeNombre(int valeur) throws Exception
  {
    if(2 <= valeur && valeur <= 10)
      return new NombreImpl(valeur);
    else
      throw new Exception();
```

```
   10   }
11 }
```

La version Java est plus verbeuse et prête plus à confusion. En contrepartie, elle permet plus d'extensibilité et plus de modularité, puisqu'une implantation de `Nombre` peut être remplacée par une autre sans avoir à réécrire le code qui manipule les instances type `Nombre`.

#### Types privés

De la même manière que les types abstraits, les types privés apparaissent, sous une autre forme, dans les langages orientés objet.

Ainsi, nous pouvons considérer qu'un type privé est un type tel que :

- il est impossible de construire directement une valeur de ce type
- toutes les informations sont disponibles sur la manière dont le type est effectivement implanté.

Dans un langage à objets, ces caractéristiques se traduiront naturelle par une classe dotée d'un constructeur protégé. Ainsi, en Java, pour implanter la deuxième version de notre type nombre, nous écrirons

```
1 //Fichier Nombre.java
2 public class Nombre
3 {
4    private final int valeur;
5    protected Nombre(int valeur) throws Exception
6    {
7       if(2 <= valeur && valeur <= 10)
8         this.valeur = valeur;
9       else
10        throw new Exception();
11   }
12 }
```

#### Exercices

1. Donnez une signature à votre module `Ast`.
2. Adaptez la signature et le module de manière à ce que `nombre`, `text` et `nom` soient des types abstraits. Pour l'implantation, pour le moment, vous emploierez respectivement les types concrets `int`, `string` et `string`.
3. Adaptez le module `Ast` de manière à ce que les seules valeurs de type `nom` soient des chaînes de caractères qui ne contiennent que des lettres (donc ni espace ni ponctuation, etc). Pour ce faire, vous pourrez utiliser les fonctions

- `Char.is_letter`, qui vérifie si un caractère est bien une lettre
- `String.length`, qui renvoie la longueur d'une chaîne de caractères
- `String.get`, qui renvoie le nème caractère d'une chaîne.

## Sous-modules et modules locaux

# À faire

# Énumérations

Compréhension

# Entrées/sorties

## Affichage de texte

module Print

## Gestion de texte

Différents encodages (Unicode, Ascii, etc.)

# Optimisation

## Aspects impératifs

# Utilisation avancée des modules

## Modules récursifs

## Points fixes entre modules

# Génération de code durant la compilation

Camlp4

# Génération de code durant l'exécution ?

LLVM

termes peuvent s’appliquer.
Voyez les termes d’utilisation pour plus de détails.
Développeurs