1 Introduction au langage Caml

1 Présentation

1.1 Le langage Caml

Le langage Caml est un langage créé par l'INRIA en 1985. La version actuelle du langage, OCaml, activement développée et largement utilisée, est celle imposée par le programme d'option informatique [1]. La documentation officielle du langage peut être obtenue sur le site de l'INRIA, à l'adresse `http://caml.inria.fr/`

En principe, Caml est un langage compilé, c'est-à-dire qu'une fois le programme Caml écrit, on utilise un *compilateur* qui traduit, une fois pour toute, le code Caml en une suite d'instructions qui seront directement compréhensibles et exécutables par le processeur. Toutefois, des environnements de développement tels que WinCaml et MacCaml [2] (respectivement pour Windows et OS-X) permettent de simuler une utilisation interactive du langage, ce qui permet, si on le souhaite, d'exécuter les commandes une à une et d'étudier le résultat qu'elles fournissent.

1.2 Philosophie du langage

Il existe différentes façons d'aborder la notion de programmation.

La programmation *impérative* est basée sur la notion de machine abstraite, constituée d'une mémoire et d'une suite d'instructions qui modifient l'état de la mémoire [3]. La gestion de la mémoire est, souvent, en grande partie à la charge du programmeur.

La programmation *fonctionnelle*, quant à elle, repose avant tout sur la définition et l'évaluation de fonctions, et évite le mécanisme d'affectation. Il n'est donc pas besoin de se soucier de la façon dont la mémoire est gérée.

Il existe également une programmation dite *objet*, centrée autour des données que l'on manipule, auxquelles sont directement associées des méthodes agissant sur ces données.

1. À compter de janvier 2018. Avant cette date, le programme imposait Caml Light, une version antérieure du langage, dont la syntaxe est quelque peu différente. Gardez cela en tête si vous parcourez des ouvrages ou des annales se référant à l'ancien langage.
2. Que l'on trouvera à l'adresse `http://jean.mouric.pagesperso-orange.fr/`. Attention, OCaml et Caml Light sont tous deux disponibles, il faudra bien choisir le bon langage via le menu *CamlTop*.
3. C'est le style de programmation que vous avez principalement utilisé en Python.

OCaml, comme la plupart des langages modernes [4] permet d'utiliser les trois styles de programmation, comme nous le verrons, mais est d'abord un langage fonctionnel. Pour cette raison, nous étudierons dans un premier temps cet aspect du langage, avant de revenir sur le style impératif qui vous est plus familier.

2 Premiers pas

2.1 Calculer avec l'interface

L'interface qui nous est proposée est divisée en deux. La partie gauche est un éditeur de texte dans lequel nous allons taper les différentes commandes. Ces commandes sont ensuite exécutées dans la partie droite, laquelle fournira la différents résultats. Un raccourci clavier (Ctrl-Entrée sous Windows, par exemple) permet d'exécuter la commande actuellement mise en valeur dans l'éditeur à gauche.

Effectuer un calcul avec Caml est simple. Il suffit d'entrer une expression que l'on termine par un double point-virgule. Ce double point-virgule indique, en Caml, la terminaison d'une expression, d'une commande ou d'une définition. **le symbole # ne fait pas partie de l'expression, mais il est indiqué ici pour distinguer les expressions envoyées à Caml des retours.**

```
# 2 + 3;;
- : int = 5
```

Intéressons-nous à la réponse de Caml. Il nous indique trois éléments. Au centre est précisé le type du résultat, ici un entier (int). Le résultat proprement dit se trouve à droite du signe égal, ici 5. Nous reviendrons plus tard sur l'élément à gauche des deux points.

Pour les calculs sur les entiers, on dispose des opérateurs d'addition (noté `+`), de soustraction (noté `-`), de multiplication (noté `*`) et de division *entière* (noté `/`). Il n'y a en revanche pas d'opérateur pour l'exponentiation.

Il est également possible de travailler avec des flottants, mais il y a une subtilité spécifique à Caml. En effet, la solution « naturelle » provoque une erreur :

```
# 1.41 + 3.14;;

Characters 2-6:
  1.41 + 3.14;;
  ^^^^
Error: This expression has type float but
       an expression was expected of type int
```

4. Dont Python!

La raison de cette erreur est que Caml utilise des opérateurs différents pour *chaque* type qu'il peut manipuler.

Pour les nombres flottants, les quatres opérateurs courants sont suivis d'un point (soit `+.` `-.` `*.` et `/.`). On dispose aussi d'un opérateur d'exponentiation, cette fois sans point (`**`).

```
# 1.41 +. 3.14 ** 2.0;;
- : float = 11.2696
```

Naturellement, les priorités habituelles des opérateurs (puissance prioritaire sur la multiplication et la division, elles-mêmes prioritaires sur l'addition et la soustraction) sont respectées

```
# 2 + 3 * 5;;
- : int = 17
```

Si l'on souhaite effectuer les opérations dans un ordre différent, il est naturellement possible d'utiliser des parenthèses :

```
# (2 + 3) * 5;;
- : int = 25
```

En l'absence de règle de priorités, l'évaluation se fait de gauche à droite, excepté l'exponentiation évaluée de droite à gauche (comme en Python) :

```
# 5 - 3 - 2;;
- : int = 0

# 2. ** 1. ** 2.;;
- : float = 2.0
```

2.2 Typage fort

Si l'utilisation d'opérateurs différents en fonction du type peut paraître contraignante, ce choix a été fait pour permettre à Caml de déterminer automatiquement, aussi souvent que possible, les types des opérandes.

Par exemple, lorsque l'on écrit `x + y`, Caml peut en déduire que les identifiants `x` et `y` sont à des entiers.

Par ailleurs, Caml utilise ce que l'on appelle un typage fort, c'est-à-dire qu'il n'essaiera jamais, de lui même, de changer le type d'un objet pour pouvoir réaliser une opération [5]. Ainsi, la somme suivante provoque une erreur :

```
# 3.0 +. 2;;

Characters 9-10:
  3.0 +. 2;;
         ^
Error: This expression has type int but
       an expression was expected of type float
```

En effet, l'opérateur `+.` attend impérativement des flottants pour chacun de ses deux opérandes, or 2 est un entier. Ce que laisse clairement comprendre la réponse de Caml dans l'exemple précédent.

Quand bien même la conversion d'un entier en flottant ne poserait ici aucun problème particulier, Caml ne le fera jamais de lui-même [6]. Il en est de même sur ce second exemple :

```
# 2 + 3.0;;

Characters 7-10:
  2 + 3.0;;
      ^^^
Error: This expression has type float but
       an expression was expected of type int
```

Il est heureusement possible de convertir un type en un autre, à condition de le faire explicitement. Ainsi, la fonction `float_of_int` permet de convertir un entier en flottant :

```
# 3.0 +. float_of_int 2;;
- : float = 5.0
```

De même, `int_of_float` permet d'effectuer la conversion inverse (en tronquant la valeur réelle si nécessaire) :

```
# 2 + int_of_float 3.0;;
- : int = 5

# 2 + int_of_float 3.5;;
- : int = 5
```

Notons ici l'absence de parenthèses encadrant l'argument des fonctions, et qu'en terme de priorités le calcul de `float_of_int 2` ou `int_of_float 3.0` ont été effectués avant l'addition, nous y reviendrons un peu plus tard.

5. Python a également un typage fort (par exemple, `range(2.0)` est refusé), mais comme c'est un langage polymorphe et que les fonctions et opérateurs usuels s'accomodent de types différents, c'est moins évident.

6. Python non plus, mais l'opérateur `+` en Python accepte une grande quantité de types pour chaque opérande, et notamment `float` pour l'opérande de gauche et `int` pour celui de droite ; le résultat est alors un flottant, le calcul étant fait sur des flottants en convertissant préalablement l'entier en flottant. La conversion n'est PAS automatique, c'est l'implémentation de l'opérateur `+` qui le demande explicitement dans cette situation.

3 Définitions

3.1 Définitions globales

Il est possible d'associer un *nom* (un ensemble de chiffres et de lettres, commençant par une lettre) à une valeur grâce à l'instruction **let**. C'est une *définition*.

Cette définition n'est pas modifiable, même s'il est possible de définir à nouveau le nom pour l'associer à une autre valeur [7].

```
# let x = 2;;
val x : int = 2
```

La troisième information que nous retourne Caml, la plus à gauche, correspond donc au *nom* auquel est associé le résultat. Si aucun nom n'est défini, on trouvera simplement un tiret - à gauche.

Une fois le nom défini, il peut être utilisé dans des calculs.

```
# let x = 2;;
val x : int = 2

# x * 3 + 4;;
- : int = 10
```

Dans la définition, on peut parfaitement utiliser une expression. Celle-ci est évaluée immédiatement, et c'est le résultat obtenu qui est associé au nom.

```
# let x = 2;;
val x : int = 2

# let y = x + 5;;
val y : int = 7

# y;;
- : int = 7
```

Puisque c'est le résultat qui est associé au nom y, une nouvelle définition de x n'a aucune incidence sur celle de y :

```
# let x = 2;;                (* On définit ici le nom x          *)
val x : int = 2

# let y = x + 5;;            (* On définit à présent le nom y    *)
val y : int = 7

# let x = 6;;                (* On redéfinit le nom x            *)
val x : int = 6

# y;;                        (* Cela n'a aucune incidence sur y *)
- : int = 7
```

Il est par ailleurs possible d'effectuer plusieurs définitions d'un seul coup, grâce au mot-clé **and** :

```
# let x = 7 and y = 8;;
val x : int = 7
val y : int = 8
```

Attention, les définitions sont interprétées *simultanément* et non successivement, comme on peut le voir si l'on redéfinit x et y en écrivant :

```
# let x = 7 and y = 8;;
val x : int = 7
val y : int = 8

# let x = 0 and y = x;;    (* Le valeur associée à x dans la seconde  *)
val x : int = 0            (* définition est celui de la définition   *)
val y : int = 8            (* précédente, c'est-à-dire x=7 et non x=0 *)
```

Cela permet de redéfinir deux noms en échangeant les valeurs associées :

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

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

3.2 Définitions locales

Il est également possible de définir un nom qui n'existera que le temps de l'évaluation d'une expression, grâce au mot-clé **in**.

7. La distinction est subtile, nous y reviendrons un peu plus tard.

```
# let a = 1 + 1 in a * 3;;
- : int = 6
```

On peut vérifier que la définition n'existe que le temps d'évaluer l'expression `a + 3` :

```
# a;;

Characters 2-3:
  a;;
  ^
Error: Unbound value a
```

On peut également faire des définitions locales multiples :

```
# let a = 1 and b = 2 in a + b;;
- : int = 3
```

Il est possible d'utiliser une définition globale d'un nom qui a déjà été défini globalement. La définition globale n'est pas affectée :

```
# let x = 0;;
x : int = 4

# let x = 5 in x + 6;;
- : int = 11

# x;;
- : int = 0
```

Il est possible d'imbriquer les définitions, par exemple

```
# let x = 7 in let x = x - 8 in x + 9;;
- : int = 8
```

En fait, on peut décoder cette instruction un peu obscure en identifiant plus clairement les deux définitions qui interviennent :

```
# let x = 7 in let xx = x - 8 in xx + 9;;
- : int = 8
```

4 Les fonctions

4.1 Fonctions avec un unique argument

Il existe de nombreuses façons de définir des fonctions en Caml. Par exemple, on peut vouloir créer une fonction f définie par[8] :

$$f \begin{cases} \mathbb{R} \longmapsto \mathbb{R} \\ x \longmapsto 3x^2 \end{cases}$$

La façon la plus simple de procéder est d'écrire

```
# let f x = 3. *. x ** 2.;;
val f : float -> float = <fun>
```

La signature obtenue indique que `f` désigne à présent une fonction (`<fun>`) qui prend en argument un flottant et retourne un flottant. L'usage d'un opérateur spécifique pour les nombres flottants a permi à Caml d'identifier correctement le type attendu pour l'argument de la fonction.

On utilise ensuite cette fonction de la sorte :

```
# f 4.0;;
- : float = 48.0
```

ou bien encore

```
# let z = 2.5 in f z;;
- : float = 18.75
```

Il est également possible de définir *localement* des fonctions, par exemple :

```
# let g x = x * 3 in g 4;;
- : int = 12

# g;;

Characters 2-3:
  g;;
  ^
Error: Unbound value g
```

8. La fonction ne sera pas réellement définie sur $\mathbb{R}$ mais simplement sur les flottants.

On remarque que le langage Caml n'utilise pas de parenthèses autour de l'argument lors de la définition de la fonction. En mettre ne provoquera pas une erreur, mais ce n'est pas l'usage car elles ne sont pas nécessaires. De la même façon, on n'en utilise pas non plus lorsque l'on fait appel à la fonction.

Il y a cependant une exception à cette règle, lorsque l'argument est négatif. Il convient d'écrire

```
# f (-4.0);;
- : float = 48.0
```

En effet, ne pas mettre les parenthèses déclenche une erreur :

```
# f -4.0;;

Characters 2-3:
  f -4.0;;
  ^
Error: This expression has type float -> float
       but an expression was expected of type int
```

Dans ce dernier cas, Caml pense que l'on a essayé de soustraire l'entier[9] `4.0` à l'entier `f`, et constaté que `f` était non pas un entier, mais une fonction prenant en argument un flottant et retournant un flottant, d'où le message d'erreur.

Compte tenu de l'ambiguité, il n'a pas pu reconnaître que le signe moins était l'opérateur unaire utilisé pour définir les nombres négatifs, et non l'opérateur binaire de soustraction. L'usage de parenthèses permet de résoudre cette difficulté.

4.2 Le mot-clé « `function` »

Une autre manière de définir une fonction est d'utiliser le mot-clé **`function`**, qui utilise une syntaxe très proche des mathématiques :

```
# let f = function x -> 3.0 *. x *. x;;
val f : float -> float = <fun>
```

La signature obtenue est exactement la même, et sont utilisation est identique :

```
# f 4.0;;
- : float = 48.0
```

9. `4.0` n'est évidemment pas un entier, mais pour Caml, les deux éléments à gauche et à droite de l'opérateur `-` devraient l'être. S'il n'y avait pas eu une erreur de type pour `f`, il y aurait eu une erreur de type sur `4.0`.

En fait, on retrouve en Caml deux syntaxes similaires à celles qui, en Python, permettent de définir une fonction, l'approche « classique » :

```
def f(x) :
    return 3.0 * x**2
```

et celle inspirée des langages fonctionnels utilisant le mot-clé **`lambda`** [10]

```
f = lambda x : 3.0 * x**2
```

Comme `lambda` en Python, **`function`** en Caml permet de définir anonymement une fonction. Il est ensuite possible d'associer un identifiant à la fonction via **`let`** (ou l'opérateur d'affectation en Python).

4.3 Arguments multiples

Il est possible d'utiliser les constructions précédentes pour définir ce qui s'apparente à des fonctions à plusieurs variables.

Par exemple, on peut écrire

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

En fait, la fonction `f` est une fonction qui prend en argument un élément de $\mathbb{Z}$ et retourne *une fonction* de $\mathbb{Z}$ à valeur dans $\mathbb{Z}$.

Une telle construction correspond, mathématiquement, à :

$$f \begin{cases} \mathbb{Z} \longmapsto (\mathbb{Z} \longmapsto \mathbb{Z}) \\ x \longmapsto \begin{cases} \mathbb{Z} \longmapsto \mathbb{Z} \\ y \longmapsto x+y \end{cases} \end{cases}$$

C'est l'interprétation qu'il faut donner à la signature fournie par Caml. Elle est équivalente à `int -> (int -> int)`, même si l'interpréteur n'indiquera pas, dans ce cas, les parenthèses, car la signature est lue de gauche à droite.

Si l'on fournit un `int` à la fonction `f`, on obtient donc une fonction de signature `int -> int`. Ainsi,

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

10. Cette seconde syntaxe n'est pas exigible aux concours, et est à utiliser avec parcimonie.

On peut donner un nom à cette fonction, puis s'en servir :

```
# let g = f 2;;
val g : int -> int = <fun>

# g 3;;
- : int = 5
```

Heureusement, il n'est pas nécessaire d'aller si loin pour utiliser la fonction `f`. Par exemple, on pourrait envisager de déterminer `f 2`, puis d'appliquer le résultat à `3`, en imposant cet ordre d'évaluation grâce à des parenthèses :

```
# (f 2) 3;;
- : int = 5
```

Plus simplement encore, Caml évaluant les expression de la gauche vers la droite (on parle *d'association à gauche*), l'expression `f 2 3` est équivalente à `(f 2) 3` :

```
# f 2 3;;
- : int = 5
```

Utiliser le mot-clé `fonction` (lequel ne permet de définir que des fonctions avec un unique argument) de la sorte étant un peu lourd, on dispose d'un autre mot-clé, **`fun`** qui est une sorte de raccourci :

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

# f 2 3;;
- : int = 5
```

On remarque que la signature est exactement la même, et que `f` se comporte exactement de la même façon.

On peut également définir `f` d'une troisième et dernière façon, encore plus brève :

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

# f 2 3;;
- : int = 5
```

4.4 Signature de la fonction

Comme on a pu le voir sur les exemples précédents, Caml détermine automatiquement le type des arguments de la fonction, ainsi que le type du résultat. Cela est rendu possible par le fait que les opérateurs nous renseignent sur la nature des opérandes. Il n'y a par exemple aucune ambiguité dans les définitions suivantes :

```
# let f x y = int_of_float x + y;;
val f : float -> int -> int = <fun>

# let g f = f 1 +. 2.;;
val g : (int -> float) -> float = <fun>
```

Cela permet à Caml de détecter très tôt d'éventuelles erreurs :

```
# let h x y = int_of_float x +. y;;

Characters 15-29:
  let h x y = int_of_float x +. y;;
              ^^^^^^^^^^^^^^
Error: This expression has type int but
       an expression was expected of type float
```

Il arrive parfois cependant qu'il ne soit pas possible de déterminer le type d'un argument, comme dans les exemples ci-dessous :

```
# let premier x y = x;;
val premier : 'a -> 'b -> 'a = <fun>

# let second x y = y;;
val second : 'a -> 'b -> 'b = <fun>
```

Ce sont des fonctions dites *polymorphes*. Le type `'a` (ou `'b`) indique que n'importe quel type est accepté. Cependant, dans la première fonction par exemple, le type du résultat sera, tout naturellement, le type du premier argument !

On peut ainsi utiliser les fonctions précédents avec des types différents :

```
# premier 1 2;;
- : int = 1

# premier 1.0 2.0;;
- : float = 1.0
```

Ces fonctions peuvent être réutilisées dans d'autres fonctions, et le mécanisme de détermination des types tâchera toujours de déterminer le type d'un maximum d'arguments et de résultats :

```
# let somme x y = premier x y + second x y;;
val somme : int -> int -> int = <fun>

# let somme x y = premier x y + second y x;;
val somme : int -> 'a -> int = <fun>
```

Dans les deux définitions précédentes, la présence de l'opérateur `+` impose que les résultats des appels à `premier` et `second` sont tous deux des entiers. Dans le premier cas, cela impose le type de `x` et `y`, mais dans le second cas, cela n'impose que le type de `x`, d'où les signatures différentes.

5 Les principaux types manipulés par Caml

5.1 Les entiers

Caml peut, on l'a vu, manipuler des entiers. Ils sont stockés sur 63 bits [11] (le 64e bit est en fait réservé pour un usage interne), en utilisant la règle du complément à deux pour représenter les entiers négatifs, ce qui signifie que l'on peut manipuler des entiers compris entre $-2^{62} = -4611686018427387904$ et $2^{62} - 1 = 4611686018427387903$.

En cas de dépassement, il y a un débordement, qui peut avoir des conséquences néfastes si l'on n'y prend pas garde :

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

# 4611686018427387903 * 2;;
- : int = -2
```

Le nom `max_int` désigne le plus grand entier positif représentable (soit $2^{62} - 1$), et `min_int` est son pendant négatif.

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

On dispose, on l'a vu, des opérateurs d'addition (`+`), de soustraction (`-`), de multiplication (`*`) et de division (`/`). Dans ce dernier cas, le résultat étant un entier, c'est le quotient de la division entière qui est retourné. Pour obtenir le reste de cette division entière, on dispose de l'opérateur **`mod`**.

```
# 42 / 17;;
- : int = 2

# 42 mod 17;;
- : int = 8
```

5.2 Les flottants

Il est également possible de manipuler des nombres flottants. Ce sont des nombres flottants sur 64 bits respectant la norme IEEE [12], avec les limitations inhérentes en terme de précision et de valeurs représentables.

La lettre « e » permet de définir un exposant (qui doit être entier). En l'absence d'exposant, le séparateur décimal (un point) est requis pour éviter toute confusion possible avec des entiers.

Nous avons déjà signalé l'existence des opérateurs courants (`+. -. *. /.` et `**`), auxquels s'ajouent de nombreuses fonctions mathématiques (`sqrt`, `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`...) Précisions que le logarithme fourni à travers la fonction `log` est le logarithme *népérien* et non décimal.

```
# let pi = 3.14159265359 in tan (pi /. 4.0);;
- : float = 1.0

# log 2.0e4;;
- : float = 9.9034875525361272

# sqrt (exp 1.0);;
- : float = 1.6487212707
```

5.3 Les caractères et chaînes de caractères

Caml propose un type **`char`** pour désigner les caractères. On les représente encadrés du symbole `'` (guillemets droits simples).

```
# let car = 'f';;
val car : char = 'f'
```

11. Ou bien 31 bits sur une machine 32 bits.

12. Soit, en particulier, un bit de signe, onze bits pour l'exposant et cinquante-deux bits pour la mantisse.

Les chaînes de caractères, qui regroupent plusieurs caractères, sont d'un type différent, `string`. On les représente entourées de " (guillemets droits supérieurs doubles).

```
# let ch = "nénuphar";;
val ch : string = "nénuphar"
```

La concaténation de chaîne est possible grâce à l'opérateur ^ (circonflexe) :

```
"micro" ^ "mega";;
- : string = "micromega"
```

La fonction `String.length` retourne la longueur d'une chaîne fournie en argument :

```
# String.length;;
- : string -> int = <fun>

String.length ch;;
- : int = 8
```

Il est possible extraire le i^e caractère d'une chaîne de caractères grâce à la fonction `String.get`, le résultat étant évidemment un caractère :

```
# String.get;;
- : string -> int -> char = <fun>

# String.get ch 4;;
- : char = 'p'
```

Notons que l'indexation commence à 0, comme très souvent en informatique.

Comme c'est une opération courante, il existe une autre manière d'accéder à un caractère dans une chaîne :

```
# ch.[4];;
- : char = 'p'
```

On peut également extraire une sous-chaîne d'une chaîne de caractères en spécifiant l'indice de départ et la longueur de la sous-chaîne souhaitée, grâce à la fonction `String.sub` :

```
# String.sub;;
- : string -> int -> int -> string = <fun>

# String.sub ch 4 2;;
- : string = "ph"
```

De même qu'il est possible de convertir des entiers en flottants et inversement, on peut convertir des valeurs numériques en chaînes de caractères et inversement avec les fonctions `string_of_int`, `string_of_float`, `float_of_string` et `float_of_int`. On dispose également de `char_of_int` et `int_of_char` pour convertir un code ASCII en caractère et inversement. La bibliothèque standard de Caml fournit encore bien d'autres fonctions pour manipuler les chaînes de caractères, que vous pourrez retrouver dans la documentation du langage.

5.4 Le type `unit`

Il n'est pas rare que certaines fonctions effectuent une opération particulière (modification de données, par exemple [13]) mais n'aient pas de résultat à retourner. Pour des raisons de cohérence, Caml impose que toute fonction retourne quelque chose, aussi dispose-t-on d'un type particulier désignant en fait, en quelque sorte, la notion de « rien ». C'est le type `unit`, qui compte `()` comme seul et unique représentant du type [14].

C'est par exemple le type retourné par les fonctions effectuant des affichages. Il existe une fonction d'affichage pour tous les types courants :

```
# print_int 3;;
3- : unit = ()

# print_float 3.14;;
3.14- : unit = ()

# print_string "Blop";;
Blop- : unit = ()
```

La lecture de la réponse n'est pas facile, ici, car l'affichage demandé se confond avec la valeur de retour de la fonction. Il ne faut pas oublier que Caml n'est pas, à l'origine, un langage interactif, et les types et valeurs retournés par les fonctions n'apparaissent pas lorsqu'on exécute normalement le programme. Seul les affichages produit par les fonctions `print_` sont visibles, d'où leur importance.

Caml n'effectue aucun retour à la ligne, afin de permettre plusieurs affichages sur la même ligne. La fonction `print_newline` permet d'obtenir ce retour à la ligne. En principe, elle ne devrait pas nécessiter d'argument, mais si l'on met uniquement le nom de la fonction, on obtient simplement sa signature !

```
# print_newline;;
- : unit -> unit = <fun>
```

13. On dit qu'elles ont un « effet de bord ».
14. Le type `unit` correspond au type `NoneType` de Python, dont le seul représentant est `None`, et qui existe pour des raisons similaires.

Aussi pour faire appel à la fonction, on lui fournit en argument un objet de type `unit`, soit nécessairement `()` :

```
# print_newline ();;

- : unit = ()
```

5.5 Les booléens

Caml dispose également d'un type booléen, `bool`, qui n'a que deux représentants, `true` et `false` (sans majuscule).

On note[15] `&&` l'opérateur logique « et », et `||` l'opérateur logique « ou ». L'opérateur unaire `not` permet d'obtenir la négation d'une valeur booléenne.

L'opérateur `not` a la plus grande priorité, suivi de `&&` et enfin de `||` :

```
# true && true || false && false;;
- : bool = true

# not false || true;;
- : bool = true
```

Mais on peut naturellement utiliser des parenthèses pour changer cet ordre d'évaluation :

```
# not (false || true);;
- : bool = false
```

Caml dispose de différents opérateurs pour comparer deux éléments : on écrira `=` pour tester l'égalité, `<>` pour tester la « non-égalité », et enfin `<  <=  >  >=` pour ce qui est des comparaisons. Ces opérateurs ont priorité sur les opérateurs logiques. Ce sont des opérateurs polymorphes, qui acceptent n'importe quel type :

```
# 3.14 = 1.41;;
- : bool = false

# 'a' <> 'z';;
- : bool = true

# "toto" <= "blop";;
- : bool = false
```

15. Signalons que Caml tolère l'utilisation de **or** à la place de `||` (mais pas de **and**, qui est réservé pour des définitions multiples !), de même que `&` à la place de `&&` (Mais pas de `|` ici, réservé pour un usage que nous verrons un peu plus tard).. Nous éviterons ces notations dans la suite.

On notera que l'égalité s'écrit avec un unique signe égal, et sa négation avec `<>`[16].

Pour qu'une comparaison soit valide, cependant, il faut que les deux éléments comparés soient impérativement de même type :

```
# 3 = 3.0;;

Characters 6-9:
  3 = 3.0;;
      ^^^
Error: This expression has type float but
       an expression was expected of type int
```

Dans le cas des chaînes de caractères, c'est l'ordre lexicographique qui est utilisé. Pour comparer les caractères entre eux, la relation d'ordre fait référence au code ASCII[17] [18].

Signalons enfin que l'évaluation des expressions booléennes est paresseuse, c'est-à-dire qu'elle cesse dès que l'on a pu déterminer avec certitude le résultat, sans évaluer la totalité des expressions, comme le montrent ces exemples (l'expression contenant une division par zéro dans la première expression n'est jamais évaluée) :

```
# 2 < 3 || (1/0) == 42;;
- : bool = true

# 2 > 3 || (1/0) == 42;;
Exception: Division_by_zero.
```

Caml définit d'autres fonctions polymorphes liées à la notion de comparaison. Les fonctions `max` et `min` acceptent deux arguments de même type, et retournent respectivement le plus grand et le plus petit des deux[19].

```
# max;;
- : 'a -> 'a -> 'a = <fun>

# max "toto" "blop";;
- : string = "toto"
```

16. On aura tôt fait de remarquer que `==` et `!=` existent, et semblent fonctionner de la même façon, mais ce sont des opérateurs d'identité et non d'égalité (ils correspondent aux opérateurs `is` et `is not` en Python).
17. Ce qui correspond à l'ordre alphabétique, mais uniquement pour comparer des minuscules non accentuées ou des capitales non accentuées. Une capitale est notamment toujours considérée plus petite qu'une minuscule.
18. Le type `string` en Caml manipule des caractères sur 8 bits. Même s'il est possible d'utiliser des caractères accentués, comme on l'a vu, la gestion de caractères non-ASCII en OCaml est compliquée, et nous ne nous y étendrons pas. Il existe des extensions à OCaml pour supporter plus efficacement les chaînes non-ASCII, et notamment unicode, tel que le module `Rope`.
19. En cas d'égalité, `min` retourne le premier argument et `max` le second.

La fonction `compare`, elle, attend deux arguments `x` et `y` de même type et retourne 0 si `x = y`, un entier positif si `x > y` et un entier négatif sinon [20].

```
# compare;;
- : 'a -> 'a -> int = <fun>

# compare 3 7;;
- : int = -4

# compare 3.2 7.5;;
- : int = -1
```

6 Plus loin dans les définitions de fonctions

6.1 Filtrage de motif

Supposons que l'on souhaite créer une fonction sinc, définie de $\mathbb{R}$ dans $\mathbb{R}$ comme le prolongement par continuité en 0 de $x \mapsto \sin(x)/x$, c'est-à-dire :

$$\text{sinc} : \begin{cases} \mathbb{R} \longmapsto \mathbb{R} \\ \begin{cases} 0 \longmapsto 1 \\ x \longmapsto \dfrac{\sin(x)}{x} \end{cases} \quad \text{si} \quad x \neq 0 \end{cases}$$

En l'absence du prolongement, nous avons vu que nous pouvions définir la fonction de cette façon :

```
# let sinc = function x -> sin(x) /. x;;

val sinc : float -> float = <fun>
```

Mais cette définition laisse de côté le cas $x = 0$. Caml nous propose une solution élégante de définir la fonction sinc, très similaire aux mathématiques :

```
# let sinc = function
    | 0.0 -> 1.0
    | x -> sin(x) /. x;;

val sinc : float -> float = <fun>
```

20. La valeur du résultat, en dehors de son signe, n'est pas spécifiée.

Ce type de structure est appelé *filtrage par motif*. On associe ainsi un ensemble de *motifs* (à gauche des flèches) avec des expressions. Dans cette situation, Caml essaiera d'associer successivement l'argument de la fonction avec chacun des motifs, et utilisera l'expression associée au *premier* motif qui convient.

Ainsi, `sinc 0.0` utilisera la première des expressions, tandis que `sinc 1.0` utilisera la seconde, l'identification de `1.0` avec le premier motif ayant échoué. `x` pouvant représenter n'importe quelle valeur, l'identification s'est bien passée. Dans l'expression associée, le nom `x` est défini localement comme l'argument de la fonction.

Dans le cas où l'argument correspond à plusieurs cas possibles, Caml s'arrête sur le premier qui convient. Il est donc important de faire figurer le motif `0.0 -> ...` avant celui `x -> ...`!

Si l'on n'a pas besoin de l'argument dans l'expression, on peut spécifier `_` dans le motif, qui peut être identifié avec n'importe quoi, la valeur étant « perdue » lorsque l'on évalue l'expression.

```
# let est_nul = function
    | 0 -> true
    | _ -> false;;

val est_nul : int -> bool = <fun>
```

Contrairement à ce que l'on pourrait penser, on ne peut pas se servir des motifs pour construire simplement une fonction polymorphe :

```
# let est_nul = function
    | 0   -> true
    | 0.0 -> true
    | _   -> false;;

Characters 49-52:
      | 0.0 -> true
        ^^^
Error: This pattern matches values of type float
       but a pattern was expected which matches values of type int
```

Dès qu'un motif permet de déterminer le type de l'argument (que ce soit explicitement, à gauche de la flèche, ou en étudiant l'utilisation du motif à droite de cette même flèche), celui-ci ne peut plus être changé.

Ainsi, dans l'exemple précédent, la première ligne du filtrage a permi de déterminer que l'argument était un entier. La seconde ligne du filtrage, qui fait référence à un argument flottant, ne saurait donc convenir.

6.2 Motifs gardés

Il est possible de définir des conditions dans un motif, grâce au mot-clé **when**. On parle de *motif gardé* :

```
# let est_positif = function
    | x when x >= 0 -> true
    | _             -> false;;

est_positif : int -> bool = <fun>
```

Caml détectera si un motif n'est pas exhaustif :

```
# let signe = function
    | 0 -> "nul"
    | x when x<0 -> "strictement négatif";;

Characters 18-85:
  ............function
      | 0 -> "nul"
      | x when x<0 -> "strictement négatif"..

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1

(However, some guarded clause may match this value.)

val signe : int -> string = <fun>
```

Dans le cas présent, il a raison, le cas de l'entier 1 n'est pas considéré.

On s'efforcera d'éviter, autant que possible, les motifs non-exhaustifs. Mais ce n'est qu'un avertissement, la fonction est quand même définie (Caml nous a fourni sa signature) et peut être utilisée, mais déclenchera une erreur[21] si l'on utilise comme argument une valeur qui ne figure pas parmi les motifs possibles :

```
# signe 0;;
- : string = "nul"

# signe 1;;
Exception: Match_failure ("//toplevel//", 225, 12).
```

21. En fait, une exception

Caml n'est pas toujours capable de déterminer l'exhaustivité d'un motif :

```
# let signe = function
   | 0            -> "nul"
   | x when x > 0 -> "strictement positif"
   | x when x < 0 -> "strictement négatif";;

Characters 18-139:
  ............function
      | 0            -> "nul"
      | x when x > 0 -> "strictement positif"
      | x when x < 0 -> "strictement négatif"..

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1

(However, some guarded clause may match this value.)

val signe : int -> string = <fun>
```

Pour cette raison, on préférera l'écriture suivante, équivalente (car le dernier motif ne sera considéré que si les précédents ne conviennent pas) :

```
# let signe = function
   | 0            -> "nul"
   | x when x > 0 -> "strictement positif"
   | _            -> "strictement négatif";;

signe : int -> string = <fun>
```

Dans certains cas, Caml identifiera l'inutilité d'un motif :

```
# let double = function
    | x -> 2.0 *. x
    | 0.0 -> 0.0;;

Characters 50-53:
      | 0.0 -> 0.0;;
        ^^^
Warning 11: this match case is unused.

val double : float -> float = <fun>
```

Remarquons que cela n'a pas empêché la définition de la fonction. Sur ce point encore, Caml n'est cependant pas capable de tout remarquer :

```
# let parité = function
    | x when x mod 2 = 0 -> "pair"
    | x when x mod 2 = 1 -> "impair"
    | x                  -> "ni pair ni impair";;

val parité : int -> string = <fun>
```

Dans ce dernier cas, si l'on retire le dernier motif, inutile, Caml protestera contre un filtrage non-exhaustif! Il faudra également réécrire le second motif pour le satisfaire.

Attention, **les noms dans les motifs ne désignent jamais de valeur**, même si le nom existe en dehors. Par exemple, dans le cas suivant, le x du premier motif ne désigne pas 0, comme en témoigne l'avertissement de Caml :

```
# let nul =
    let x = 0 in function
        | x -> true
        | _ -> false;;

Warning 26: unused variable x.

Characters 68-69:
          | _ -> false;;
            ^
Warning 11: this match case is unused.

val nul : 'a -> bool = <fun>
```

La signature également peut nous interpeler : si l'on comparait l'argument à zéro, nous ne devrions pas avoir une fonction polymorphe... Il faudrait plutôt écrire [22]

```
# let nul =
   let x = 0 in function
      | y when y = x -> true
      | _            -> false;;

val nul : int -> bool = <fun>
```

22. Enfin... il faudrait surtout l'écrire complètement autrement, car c'est une façon très tarabiscotée de vérifier si l'argument est égal à 0 !

Attention, si **function** permet le filtrage par motif, ce n'est pas [23] le cas de **fun**.

Il est possible d'utiliser ce mécanisme hors du cadre d'une fonction, grâce à la structure **match** *expr* **with** qui peut être employée à tout endroit où l'on peut mettre une expression. Par exemple, si n est égal à 17, on a :

```
# let parité n =
    let reste = n mod 2 in
        match reste with
            | 0 -> "pair"
            | _ -> "impair";;

val parité : int -> string = <fun>
```

Cette construction permet par ailleurs de filtrer le résultat d'une expression (ce que ne permet pas **function**) :

```
# let parité n =
    match n mod 2 with
        | 0 -> "pair"
        | _ -> "impair";;

val parité : int -> string = <fun>
```

6.3 Fonctions récursives

Sans vouloir entrer trop dans les détails (nous consacrerons un chapitre aux fonctions récursives un peu plus tard), une fonction récursive est, grossièrement, une fonction qui s'appelle elle-même. Il n'est pas possible de définir une fonction récursive en Caml sans une petite astuce, car au moment où l'on définit notre fonction `f`, elle n'existe pas encore, donc il n'est pas possible de l'utiliser dans la définition. On signale donc à Caml qu'il faut faire une petite entorse à ses habitudes en ajoutant un « **rec** » juste après le **let**.

Par exemple, il est très simple de définir récursivement la fonction `fact` représentant la factorielle en mathématiques :

```
# let rec fact = function
    | 0 -> 1
    | n -> n * fact (n-1);;

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

23. En OCaml en tout cas, car c'était possible avec le langage Caml Light, et vous risquez de le rencontrer dans des ouvrages ou des annales.

Nous verrons un peu plus tard que cette définition, très proche de la définition mathématique, n'est pas la plus efficace, mais nous nous en contenterons bien pour l'instant.

On rappelle que Caml travaille sur des entiers sur 31 bits, donc pour de « grandes » valeurs de n (plus de... 20), on aura un résultat incorrect (`fact 21` est négatif, puis le résultat sera nul pour $n \geqslant 64$).

On peut, de façon similaire, réécrire une fonction déterminant la parité d'un entier[24] :

```
# let rec est_pair = function
    | 0 -> true
    | 1 -> false
    | n -> est_pair (n-2);;

val est_pair : int -> bool = <fun>
```

Pour tenir compte un peu plus efficacement des nombres négatifs, on peut ajouter, dans le filtrage, une condition supplémentaire entre la seconde et la troisième :

```
# let rec est_pair = function
    | 0           -> true
    | 1           -> false
    | n when n<0 -> est_pair (-n)
    | n           -> est_pair (n-2);;

val est_pair : int -> bool = <fun>
```

Une autre façon de déterminer la parité d'un entier serait d'utiliser deux fonction récursives, qu'ils nous faut définir simultanément grâce au mot-clé **and** :

```
# let rec est_pair = function
    | 0           -> true
    | n when n<0 -> est_pair (-n)
    | n           -> est_impair (n-1)
and est_impair = function
    | 0           -> false
    | n when n<0 -> est_impair (-n)
    | n           -> est_pair (n-1);;

val est_pair : int -> bool = <fun>
val est_impair : int -> bool = <fun>
```

24. En principe positif, mais dans le cas d'un nombre négatif, on aura un débordement et la fonction marcherait quand même en théorie, après plusieurs miliards de milliards d'appels récursifs.

Exercices

Ex. 1.1 – Calculs

Déterminer les résultats des calculs suivants, effectués à l'aide de Caml :

```
1 + 2 * 3;;

4 - 3 * 5 / 2;;

1e2 /. 0.5e1 -. 150e-1;;

3.0 *. 2.0 ** 1.0 ** 2.0 /. 3.0;;

log ( exp( 4.0 ) ** 2.0 );;
```

Ex. 1.2 – Définitions

Déterminer les réponses de Caml aux définitions suivantes :

```
let a = 1;;

let f n = 3 * n - 1;;

let a = 2 in f a;;

let a = f a in f a;;

let a = f a and b = f a in f b;;

let x =
  let x = 5 and y = 2 in
    let x = x+y and y = x-y in
      2 * x / y;;
```

Ex. 1.3 – Composition

On définit deux fonctions f et g de la façon suivante :

```
let f n = n + 2 and g n = 3 * n;;
```

On souhaite définir une fonction h comme la composition de ces deux fonctions, soit $h = g \circ f$, et calculer $h(5)$. Parmi les définitions suivantes, lesquelles sont correctes ?

```
let h   = g f       in h 5;;

let h n = g f n     in h 5;;

let h n = (g f) n   in h 5;;

let h n = g (f n)   in h 5;;

let h n = g(f) (n)  in h 5;;

let h n = g (f (n)) in h 5;;
```

Ex. 1.4 – Typage

Proposer des expressions Caml qui ont pu donner les signatures suivantes :

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

val g : (int -> int) -> int = <fun>

val h : int -> (int -> int) -> int = <fun>
```

Déterminer le type des expressions suivantes :

```
fun f x y -> f x y;;

fun f g x -> g (f x);;

fun f g x -> (f x) + (g x);;
```

Même chose avec ces expressions, où les noms sont moins révélateurs :

```
fun x y z -> (x y) z;;

fun x y z -> x y z;;

fun x y z -> x (y z x);;

fun x y z -> (x y) (z x);;
```

Ex. 1.5 – Différences finies

On suppose qu'une suite $(u_n)_{n\in\mathbb{N}}$ d'éléments de $\mathbb{R}$ est définie en Caml par une fonction u qui, à tout entier n positif, associe le terme $u_n \in \mathbb{R}$. Par exemple :

```
# let u = function n ->
              let fl_n = float_of_int n in
                  3.2 *. fl_n *. (1.0 -. fl_n);;

val u : int -> float = <fun>
```

Écrire une fonction `delta` qui, à une suite $(u_n)_{n\in\mathbb{N}}$ associe la suite $(u_{n+1} - u_n)_{n\in\mathbb{N}}$.

Quelle est sa signature ?

Ex. 1.6 – Logique

Écrire une fonction Caml prenant en argument deux booléens et retournant le booléen associé à l'opérateur logique $\Rightarrow$ dont la table de vérité est rappelée ci-dessous :

a	b	$a \Rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

On considère l'expression logique suivante, définie pour quatre booléens a, b, c et d :

« b et (a et d ou non a et non d) ou d et (a et non b ou b et non a) »

Remplir le tableau logique ci-dessous et en déduire une fonction Caml aussi simple que possible retournant l'expression logique demandée.

		c d	c $\overline{d}$	$\overline{c}$ $\overline{d}$	$\overline{c}$ d
a	b				
a	$\overline{b}$				
$\overline{a}$	$\overline{b}$				
$\overline{a}$	b				

Ex. 1.7 – Cherchez l'erreur

On souhaite écrire une fonction qui prend en argument un entier strictement positif p et un entier relatif n, et retourne l'entier n' compris entre 0 et p-1 vérifiant $n' - n = k \times p$ où k est un entier (pour n et p positif, il s'agit donc du reste de la division entière).

On propose la fonction suivante :

```
let modulo p n = fun
    | n when 0 <= n and n < p -> n
    | n when n < 0            -> modulo p n+p
    | n when n >= p           -> modulo p n-p;;
```

Saurez-vous retrouver les erreurs et modifier la fonction pour qu'elle soit juste ?

Ex. 1.8 – Fonction mystérieuse

Déterminer la signature et le résultat de la fonction suivante :

```
let f = function
    | 0 -> "0"
    | x -> let rec g = function
               | 0  -> ""
               | x when x mod 2 = 1 -> g (x/2) ^ "1"
               | x                  -> g (x/2) ^ "0"
           in g x;;
```

Ex. 1.9 – Facteurs premiers

Un entier positif est un nombre de Hamming si et seulement s'il s'écrit sous la forme $2^n \times 3^p \times 5^q$ où n, p et q sont des entiers positifs ou nuls.

Écrire une fonction récursive `hamming` de signature `int -> bool` prenant en argument un entier positif et retournant un booléen indiquant si l'argument est un nombre de Hamming.

Proposer une fonction récursive `divisible` de signature `int -> int -> bool` prenant deux arguments entiers strictement positifs `n` et `d` et retournant un booléen qui indique si `n` est divisible par un entier compris entre `d` et $\sqrt{n}$.

En déduire une fonction `premier` de signature `int -> bool` qui indique si l'entier strictement positif passé en argument est un nombre premier.

2 Structures de données complexes

Dans ce second chapitre, nous verrons comment créer, en fonction des besoins, de nouveaux types, et comment il est possible de définir de la sorte des listes chaînées. Puis nous verrons comment utiliser le type `'a list` que nous fournit le langage Caml. Enfin, nous introduirons le concept d'*arbre* et nous verrons comment les représenter en Caml.

1 Les couples

1.1 Principe

Il est parfois utile de pouvoir « regrouper » plusieurs éléments. Ceci est possible à travers des « tuples » où les différents éléments sont regroupés entre parenthèses [1] et séparés par des virgules.

Par exemple, on peut considérer une paire d'entiers telle que :

```
# ( 2, 3 );;
- : int * int = (2, 3)
```

Il est possible de grouper ainsi un nombre quelconque d'éléments, qu'ils aient ou non le même type. Le type de l'objet ainsi construit correspond au produit cartésien des types des différents éléments, et est noté `*`.

L'exemple suivant regroupe par exemple un flottant, une chaîne de caractères, et une fonction :

```
# ( 3.14, "pi", function x -> x**2. );;
- : float * string * (float -> float) = (3.14, "pi", <fun>)
```

On peut utiliser une définition pour associer un nom à un tel couple, telle que

```
# let grp = ( 3.14, "pi", function x -> x**2. );;
val grp : float * string * (float -> float) = (3.14, "pi", <fun>)
```

Dans l'exemple précédent, le nom `grp` désigne donc le groupe des trois éléments.

1. Comme en Python, les parenthèses ne sont en fait pas requises s'il n'y a pas d'ambiguïté.

De tels groupes peuvent être utilisés comme arguments d'une fonction. On peut par exemple écrire une fonction effectuant la somme de deux entiers ainsi :

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

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

Ces deux déclarations sont équivalentes. On peut voir qu'un groupe est bien considéré comme un seul et unique élément, ce qui permet d'utiliser `function`. Dans le second cas, on utilise en fait un filtrage par motif.

On aurait pu écrire (sans que cela soit très pertinent, mais pour illustrer le principe) :

```
# let somme = function
    | (0,0) -> 0
    | (x,y) -> x+y;;

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

Pour utiliser ces fonctions, on utilise un unique argument, un couple :

```
# somme (2,3);;
- : int = 5
```

La présence de parenthèses, et de virgules séparant chaque élément, rend la syntaxe très proche de ce qui est utilisé dans d'autres langages. Toutefois, on a affaire à un objet un peu différent des fonctions présentées dans le premier chapitre, que l'on avait définies avec l'une des variantes (toutes trois équivalentes) suivantes :

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

# let somme_cur = fun x y -> x + y;;
val somme : int -> int -> int = <fun>

# let somme_cur = function x -> function y -> x + y;;
val somme : int -> int -> int = <fun>
```

Les formes proposées dans le premier chapitre sont dite « currifiées [2] ». On remarquera

2. Du nom du mathématicien et logicien Haskell Brook Curry, qui a popularisé cette notation, même s'il semblerait qu'elle ait été initialement proposée par Moses Shönfinkel. Trois languages de programmation ont été nommés en son honneur, Haskell, Brook et Curry, le premier des trois restant une référence dans le domaine de la programmation fonctionnelle.

en particulier la différence de signature entre les deux approches, que l'on peut résumer ainsi (formes currifiées à gauche) :

$$\texttt{somme\_cur} \begin{cases} \mathbb{Z} \longmapsto (\mathbb{Z} \longmapsto \mathbb{Z}) \\ x \longmapsto \begin{cases} \mathbb{Z} \longmapsto \mathbb{Z} \\ y \longmapsto x+y \end{cases} \end{cases} \qquad \texttt{somme} \begin{cases} \mathbb{Z} \times \mathbb{Z} \longmapsto \mathbb{Z} \\ (x,y) \longmapsto x+y \end{cases}$$

En fait, la *currification* est l'opération qui consiste à transformer une fonction à plusieurs variables (par exemple ici définie sur $\mathbb{Z} \times \mathbb{Z}$ à valeur dans $\mathbb{Z}$) en une fonction à une unique variable retournant une fonction sur le reste des arguments (par exemple ici une fonction définie sur $\mathbb{Q}$ à valeur dans l'espace des fonctions $\mathbb{Z} \mapsto \mathbb{Z}$).

La forme currifiée, plus souple, est davantage utilisée en Caml. Il est en effet par exemple plus aisé de créer une *application partielle*, c'est-à-dire une fonction annexe où le premier des paramètres est fixé, avec la forme currifiée d'une fonction :

```
# let ajoute_1 = somme_cur 1;;
val ajoute_1 : int -> int = <fun>
```

Même s'il reste possible de créer une application partielle avec une fonction non currifiée :

```
# let ajoute_1 = function x -> somme (1,x);;
val ajoute_1 : int -> int = <fun>
```

Pour extraire un élément d'un groupe, on peut créer une fonction :

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

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

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

On peut alors écrire :

```
# premier_parmi_trois grp;;
- : float = 3.14

# troisième_parmi_trois grp (premier_parmi_trois grp);;
- : float = 9.8596
```

Pour des paires (des groupes de deux éléments), il existe déjà deux fonctions fournies par le langage, `fst` et `snd`, qui effectuent exactement ce travail :

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

# snd;;
- : 'a * 'b -> 'b = <fun>

# let paire = (2, 3);;
paire : int * int = 2, 3

# snd paire;;
- : int = 3
```

Il n'existe en revanche pas de solution toute prête dans le langage pour un nombre d'éléments supérieur à deux.

1.2 Couples et filtrages

Si l'on préfère en général les formes curryfiées, il est parfois pratique d'utiliser un couple (ou un tuple) comme argument si on souhaite effectuer un filtrage. En effet, seul le mot-clé **`function`** permet d'effectuer un filtrage par motif, **`fun`** ne le permet pas.

Par exemple, pour calculer le PGCD de deux nombres par l'algorithme d'Euclide, on ne peut pas écrire :

```
# let rec pgcd = fun
    | a 0 -> a
    | a b -> pgcd b (a mod b);;

Characters 23-24:
    | a 0 -> a
      ^
Error: Syntax error
```

On peut en revanche écrire :

```
# let rec pgcd = function
    | (a, 0) -> a
    | (a, b) -> pgcd (b, a mod b);;

val pgcd : int * int -> int = <fun>
```

Pour obtenir une fonction currifiée, on peut utiliser une fonction auxilliaire :

```
# let pgcd a b =
    let rec pgcd_aux = function
        | (a, 0) -> a
        | (a, b) -> pgcd_aux (b, a mod b)
  in pgcd_aux (a, b);;

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

Remarquons qu'ici, on peut en fait se passer de cette pirouette[3], car il suffit de filtrer le second paramètre de la fonction :

```
# let pgcd a = function
    | 0 -> a
    | b -> pgcd b (a mod b);;

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

2 Les types construits

Il est possible de définir ses propres types en Caml, grâce au mot-clé **type**. Ce sont des assemblages de différents types existants et/ou de constantes. Les possibilités offertes, comme nous allons le voir, sont assez importantes.

2.1 Type « union » (ou type « somme »)

On peut tout d'abord définir un type comme un choix entre plusieurs « constantes » que l'on précise. Ces constantes sont des identifiants commençant par une majuscule. Le « pipe » | joue le rôle de « ou » logique pour séparer plusieurs valeurs possibles.

Par exemple, on peut définir un type `direction` représentant les quatre points cardinaux par[4][5] :

```
# type direction = Nord | Est | Sud | Ouest;;
```

Dorénavant, `Nord`, `Est`, `Sud` et `Ouest` sont des constantes de type `direction` :

```
# let dir = Ouest;;
val dir : direction = Ouest
```

On peut également concevoir une direction comme un angle en degrés (0.0 correspondant au nord, 90.0 à l'est, 180.0 au sud et ainsi de suite) :

```
# type direction = Nord | Est | Sud | Ouest | Angle of float;;
```

Pour utiliser ensuite à cette possibilité, on utilise l'étiquette « `Angle` » qui a été définie dans le type, ce qui permet à Caml d'identifier le type correctement :

```
# let dir = Angle 45.0;;
val dir : direction = Angle 45.
```

En fait, `Angle` est un « constructeur » qui se comporte comme une fonction, prenant en argument un flottant et retournant un object de type `direction`.

Attention toutefois, Caml ne peut bien évidemment pas deviner nos intentions :

```
# Sud = Angle 180.0;;
- : bool = false
```

Les déclarations de types peuvent être récursives, par exemple :

```
# type direction = Nord | Est | Sud | Ouest | Angle of float
                 | Mediane of direction * direction;;
```

On considérera que la médiane de deux directions, dans la définition précédente, est la direction « médiane » de l'angle inférieur à 180 ° formé par les deux directions[6]. Ce qui permet de définir très librement d'autres directions à partir des quatres points cardinaux[7] :

```
# let sudEst = Mediane (Sud, Est);;
val sudEst : direction = Mediane (Sud, Est)

# let sudSudEst = Mediane (Angle 180.0, sudEst);;
val sudSudEst : direction = Mediane (Angle 180., Mediane (Sud, Est))
```

Là encore, Caml a défini ici, en même temps que le type, un constructeur `Mediane` qui prend cette fois un couple de deux objets de type `direction` et retourne un objet de type `direction`.

3. Il est en fait bien évidemment toujours possible de filtrer argument par argument, mais l'écriture peut devenir assez lourde.
4. La réponse de Caml pour une déclaration de type, lorsqu'elle est syntaxiquement correcte, consiste juste à afficher le type nouvellement défini ; on omettra donc les réponses de Caml pour les déclarations de type.
5. Le type booléen de Caml aurait ainsi pu être défini par `type bool = true` false|, si ce n'est que les constantes ne commencent pas ici par une majuscule.
6. On supposera que ces deux directions ne sont pas opposées.
7. Les noms de ces nouvelles directions commencent par des minuscules, car il ne s'agit pas de constructeurs ou de valeurs déclarés à l'intérieur d'une définition de type.

2.2 Type « enregistrement » (ou type « produit »)

Une coordonnée GPS est constituée de deux éléments : une latitude et une longitude.

Pour représenter un élément de ce genre, on peut créer un type regroupant des types existants. La déclaration se fait au moyen d'accolades, en donnant un nom (souvent appelé *étiquette*) à chacun des éléments du groupe, afin de pouvoir y référer ultérieurement.

On peut par exemple définir une position (définie donc par sa latitude nord/sud et sa longitude est/ouest) en écrivant :

```
# type position = { n_s : float;
                    e_o : float };;
type position = { n_s : float; e_o : float; }
```

Le type `position` est donc constitué de deux flottants, ces deux flottants étant désignés par les étiquettes `n_s` et `e_o`.

Pour créer un objet de ce type, on renseigne les différents « champs », en utilisant le signe « = » cette fois pour séparer l'étiquette et la valeur qui lui est associée :

```
# let pos = { e_o = 4.5;
              n_s = 10.2 };;
val pos : position = {n_s = 10.2; e_o = 4.5}
```

Il est à noter que l'ordre n'a pas d'importance [8] :

```
# let pos = { n_s = 10.2;
              e_o = 4.5 };;
val pos : position = {n_s = 10.2; e_o = 4.5}
```

Mais il faut impérativement renseigner tous les champs :

```
# let pos = { e_o = 4.5; };;

Characters 12-25:
  let pos = { e_o = 4.5; };;
            ^^^^^^^^^^^^^
Error: Some record field labels are undefined: n_s
```

On a défini en fait un type très similaire au type `float * float`, à ceci près que l'ordre, utilisé dans la paire pour identifier les deux éléments, est remplacé dans le cas du type `position` par des étiquettes.

8. Il n'est pas non plus nécessaire d'aller à la ligne entre chaque champ, on le fait ici uniquement dans un but de lisibilité.

Cela facilite la récupération des données, il suffit de faire suivre le nom désignant un objet de type `position` par un point suivi du nom d'une l'étiquette :

```
# pos.e_o;;
- : float = 4.5
```

À présent, on peut essayer de convertir la donnée d'une distance dans une direction en une position.

Dans un premier temps, nous allons écrire une fonction permettant de « ramener » un angle dans l'intervale $]-180.0, 180.0]$:

```
# let rec modulo = function
    | x when x >   180. -> modulo (x -. 360.)
    | x when x <= -180. -> modulo (x +. 360.)
    | x                 -> x;;

val modulo : float -> float = <fun>
```

Puis grâce à un filtrage pour analyser les différentes situations possibles, on détermine l'angle qu'une direction quelconque (plus précisément un élément de type `direction`) fait avec le nord [9] :

```
# let rec calcAngle = function
    | Angle a               -> a
    | Nord                  -> 0.0
    | Est                   -> 90.0
    | Sud                   -> 180.0
    | Ouest                 -> 270.0
    | Mediane (dir1, dir2)  ->
        let angle1 = calcAngle dir1
        and angle2 = calcAngle dir2
            in match modulo (angle2 -. angle1) with
                | 180.0 -> failwith "Directions opposées"
                | diff  -> angle1 +. diff /. 2.;;

val calcAngle : direction -> float = <fun>
```

Ce qui permet par exemple de calculer l'angle avec le nord de la direction `sud_sud_est` :

```
calcAngle sudSudEst;;
- : float = 157.5
```

9. On remarquera le `failwith "Directions opposées"` qui déclenchera une erreur si l'on tente de calculer la médiane de deux directions opposées.

Puis on crée la fonction qui nous intéresse :

```
# let calcPosition dist dir =
    let pi = 3.1415926535897932 in
    let angle = (calcAngle dir) *. pi /. 180.
        in { n_s = dist *. cos(angle);
             e_o = dist *. sin(angle) };;

val calcPosition : float -> direction -> position = <fun>
```

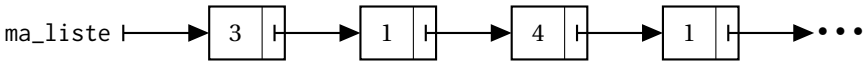
Ainsi, un déplacement de 50 m vers le sud-sud-est

```
# calcPosition 50.0 sudSudEst;;
- : position = {n_s = -46.193976625564339; e_o = 19.134171618254495}
```

correspond donc à un déplacement d'environ 46 m vers le sud et 19 m vers l'est!

3 Créer une liste chaînée

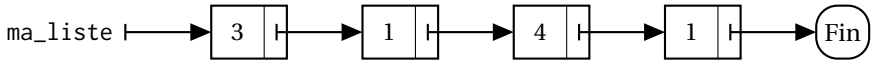
Supposons que l'on souhaite créer une liste d'entiers. Une liste est une suite ordonnée d'éléments. On peut donc considérer qu'une liste est constituée de « cellules » contenant une valeur entière, et un moyen d'accéder à la cellule suivante dans la liste.



Ce qui suit un élément dans la liste d'entier est une liste d'entiers. On peut donc envisager de décrire une liste d'entiers par le type suivant :

```
# type liste_int = { valeur : int; suivant : liste_int };;
```

L'ennui avec la définition précédente est que les listes n'ont pas de fin! On doit pouvoir avoir un « bouchon » qui indique la fin de la liste. Suivant doit donc désigner, au choix, soit une cellule [10] constituée d'un entier et d'un suivant, soit le « bouchon ».



On peut donc définir notre liste de la façon suivante :

```
# type liste_int =
   Fin | Cellule of { valeur : int ; suivant : liste_int };;
```

10. Dans la définition, le `cellule` sans majuscule désigne un type, le `Cellule` est une étiquette permettant d'identifier à quel « cas » du type union on a affaire.

On peut ensuite définir une liste :

```
# let ma_liste = Cellule { valeur = 3 ; suivant =
                   Cellule { valeur = 1 ; suivant =
                     Cellule { valeur = 4 ; suivant =
                       Cellule { valeur = 1 ; suivant = Fin } } } };;

val ma_liste : liste_int =
  Cellule
   {valeur = 3;
    suivant =
     Cellule
      {valeur = 1;
       suivant =
        Cellule {valeur = 4;
         suivant = Cellule {valeur = 1; suivant = Fin}}}}
```

Ce n'est pas une façon très pratique de définir la liste. On préférera enfiler les éléments un par un comme des perles sur un fil. On peut aisément définir une fonction qui ajoute un élément à gauche de la liste :

```
# let ajouteGauche liste elem =
      Cellule { valeur = elem ; suivant = liste };;

val ajouteGauche : liste_int -> int -> liste_int = <fun>
```

Il ne reste ensuite qu'à insérer les éléments un à un, en partant de la fin :

```
# let ma_liste = Fin;;
val ma_liste : liste_int = Fin

# let ma_liste = ajouteGauche ma_liste 1;;
val ma_liste : liste_int = Cellule {valeur = 1; suivant = Fin}

# let ma_liste = ajouteGauche ma_liste 4;;
val ma_liste : liste_int =
  Cellule {valeur = 4; suivant = Cellule {valeur = 1; suivant = Fin}}

# let ma_liste = ajouteGauche ma_liste 1;;
val ma_liste : liste_int = ...

# let ma_liste = ajouteGauche ma_liste 3;;
val ma_liste : liste_int = ...
```

On peut ensuite vouloir écrire des fonctions qui agissent sur la liste. On travaille alors par filtrage. Il faut toutefois prendre garde à la possibilité que la liste soit vide. Un nom désignant une liste peut désigner deux choses, d'après le type `liste_int` :

- Une `cellule`, désignée par `Cellule`, contenant une `valeur` (de type `int`) et un `suivant` (de type `liste_int`);
- La constante `Fin`, indiquant la terminaison de la liste.

Les fonctions agissant sur des objets `liste_int` vont donc généralement utiliser un filtrage correspondant aux deux cas du dessus. Par exemple, le seul élément directement accessible est l'élément en tête de liste, et il est aisé de l'obtenir.

```
# let têteListe = function
    | Cellule { valeur = v ; suivant = _ } -> v
    | Fin                                  -> failwith "Liste vide !";;

val têteListe : liste_int -> int = <fun>
```

On remarquera la façon dont le filtrage nous permet d'accéder aux différents éléments du type enregistrement.

On pourrait également écrire [11] :

```
# let têteListe = function
    | Cellule c -> c.valeur
    | Fin       -> failwith "Liste vide !";;

val têteListe : liste_int -> int = <fun>
```

Pour obtenir une liste privée du premier élément, c'est également assez simple :

```
# let queueListe = function
    | Cellule { valeur = _ ; suivant = s } -> s
    | Fin                                  -> failwith "Liste vide !";;

val queueListe : liste_int -> liste_int = <fun>
```

Ou bien, de façon équivalente :

```
# let queueListe = function
    | Cellule c -> c.suivant
    | Fin       -> failwith "Liste vide !";;

val queueListe : liste_int -> liste_int = <fun>
```

11. C'est essentiellement un choix de style, vous pouvez choisir celui avec lequel vous êtes le plus à l'aise.

Obtenir le dernier élément de la liste est un peu plus difficile, mais ce n'est pas insurmontable en utilisant la récursion :

```
# let rec dernierListe = function
    | Cellule { valeur = v ; suivant = Fin } -> v
    | Cellule { valeur = _ ; suivant = s }  -> dernierListe s
    | Fin                                   -> failwith "Liste vide !";;

val dernierListe : liste_int -> int = <fun>
```

Ou bien encore :

```
# let rec dernierListe = function
    | Cellule c when c.suivant = Fin -> c.valeur
    | Cellule c                      -> dernierListe c.suivant
    | Fin                            -> failwith "Liste vide !";;

val dernierListe : liste_int -> int = <fun>
```

De la même façon, on peut obtenir la longueur de la liste :

```
# let rec longueurListe = function
    | Cellule { valeur = _ ; suivant = s }   -> longueurListe s + 1
    | Fin                                    -> 0;;

val longueurListe : liste_int -> int = <fun>
```

Ou bien

```
# let rec longueurListe = function
    | Cellule c -> longueurListe c.suivant + 1
    | Fin       -> 0;;

val longueurListe : liste_int -> int = <fun>
```

L'un des inconvénients de notre type liste est qu'il ne peut contenir que des entiers. On peut faire un peu mieux, et définir une liste polymorphe, contenant des éléments certes tous de même type, mais d'un type que l'on choisira, en écrivant :

```
# type 'a liste =
   Fin | Cellule of { valeur : 'a ; suivant : 'a liste };;
```

Toutes les fonctions définies précédemment restent correctes, sous réserve de les définir *après* la définition du type `'a liste`, mais leur type va bien évidemment changer!

À commencer par la liste `ma_liste`, qui, quelle que soit la façon dont on la crée, aura un type différent :

```
ma_liste : int liste = Cellule {valeur = 3; suivant = ...
```

Et il est, à présent, possible d'y glisser des flottants à la place des entiers, sans changer la définition du type (mais la liste doit toujours contenir des éléments qui sont tous du même type) :

```
ma_liste : float liste = Cellule {valeur = 3.0; suivant = ...
```

Et les signatures de nos différentes fonctions ont également évolué, par exemple :

```
val ajouteGauche : 'a liste -> 'a -> 'a liste = <fun>
```

4 Les listes Caml

4.1 Création et manipulation

Nous n'irons pas plus loin dans cette direction car... Caml a son propre type `'a list` pour décrire des listes d'éléments, et dans la suite, nous utiliserons le type proposé par Caml. Toutefois, l'implémentées des listes Caml est similaire à celles que nous venons de construire, ce qui aidera à mieux comprendre ce qui se passe.

Les listes en Caml sont des conteneurs immutables pouvant contenir un nombre quelconque d'éléments, dont le type peut être librement choisi (les éléments peuvent d'ailleurs être des listes). Elles sont représentées entre crochets, les éléments étant séparés par des points-virgules [12] :

```
# let ma_liste = [ 1; 2; 3; 4 ];;
val ma_liste : int list = [1; 2; 3; 4]

# let ma_liste = [ "Hello"; "World" ];;
val ma_liste : string list = ["Hello"; "World"]

# let ma_liste = [ [ 1.41; 3.14 ]; [ 1.0; 2.0; 3.0 ]; [] ];;
val ma_liste : float list list = [[1.41; 3.14]; [1.; 2.; 3.]; []]

# let ma_liste = [ sin; cos ];;
val ma_liste : (float -> float) list = [<fun>; <fun>]
```

12. Attention, la liste `[ 1, 2, 3 ]` est une liste à un seul élément, un tuple (de type `int * int * int`).

Il est impératif que tous les éléments d'une liste soient du même type :

```
# let ma_liste = [ 1; 2.3; 5 ];;

Characters 22-25:
  let ma_liste = [ 1; 2.3; 5 ];;
                       ^^^
Error: This expression has type float
    but an expression was expected of type int
```

On dispose d'un nouvel opérateur, appelé « *conse* », noté « `::` », qui permet de construire une liste constituée d'un nouvel élément accroché à gauche d'une liste :

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

Il n'y a pas d'insertion à droite, car Caml implémente les listes comme des listes chaînées très semblables à celles que nous avons nous-même définies. L'opérateur conse attend donc un élément d'un certain type à gauche, et une liste d'éléments de même type (ou une liste vide) à droite. Il est donc incorrect d'écrire :

```
# [ 2; 3; 4; 5 ] :: 6;;

Characters 20-21:
  [ 2; 3; 4; 5 ] :: 6;;
                    ^
Error: This expression has type int
    but an expression was expected of type int list list
```

De même, on dispose d'un opérateur de concaténation de deux listes (contenant des éléments de même type), noté « `@` » :

```
# [ 1; 2; 3 ] @ [ 5; 6 ];;
- : int list = [1; 2; 3; 5; 6]
```

La liste vide est simplement désignée par `[]` et son type, faute d'élément, est `'a list`. C'est la seule liste qui accepte, via l'opérateur `::`, un élément de n'importe quel type :

```
# let ma_liste = [];;
val ma_liste : 'a list = []

# let ma_liste = 1 :: ma_liste;;
val ma_liste : int list = [1]
```

La fonction [13] `List.hd` permet d'obtenir le premier élément d'une liste :

```
# List.hd;;
- : 'a list -> 'a = <fun>

# List.hd [ 1; 2; 3; 4 ];;
- : int = 1
```

De même, la fonction `List.tl` retourne la liste privée de son premier élément [14] :

```
# List.tl;;
- : 'a list -> 'a list = <fun>

# List.tl [ 1; 2; 3; 4 ];;
- : int list = [2; 3; 4]
```

Notons que la « queue » d'une liste d'entiers sera toujours une liste d'entiers, même si elle est vide. On ne pourra pas insérer autre chose qu'un entier dans une telle liste.

```
# let ma_liste = [];;
val ma_liste : 'a list = []

# let ma_liste = 1 :: lst;;
val ma_liste : int list = [1]

# let ma_liste = List.tl ma_liste;;
val ma_liste : int list = []

# 3.14 :: ma_liste;;

Characters 9-12:
  3.14 :: ma_liste;;
          ^^^
Error: This expression has type int list
    but an expression was expected of type float list
```

4.2 Immutabilité des listes

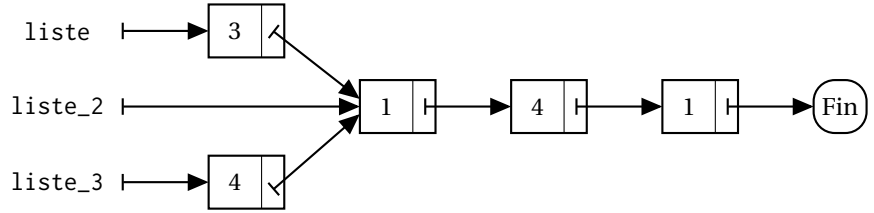
Il convient de bien garder en tête que les listes étant des objets immutables, l'utilisation de l'opérateur *conse* « `::` » ou de la fonction `List.tl` produisent de nouveaux objets sans toucher à la liste passée en paramètre.

Cependant, les listes (allogée ou raccourcie) qui en résultent sont créées sans que les éléments qui les constituent ne soient recopiés. Plusieurs listes peuvent partager les mêmes éléments, ce qui ne peut se comprendre qu'en interprétant les choses en terme de listes chaînées.

Par exemple, considérons la séquence d'instructions suivante :

```
# let liste = [ 3; 1; 4; 1 ];;
val liste : int list = [3; 1; 4; 1]

# let liste_2 = List.tl liste;;
val liste_2 : int list = [1; 4; 1]

# let liste_3 = 4 :: liste_2;;
val liste_3 : int list = [4; 1; 4; 1]
```

Le résultat, en mémoire, est quelque chose qui s'apparente à cette construction :

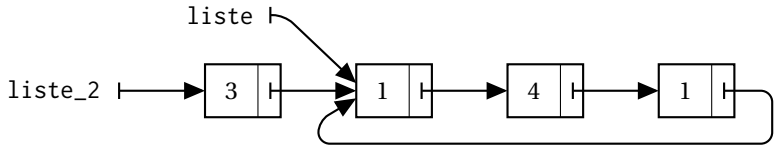


On remarquera en particulier que les listes ont une partie de leurs éléments en commun.

Conjugué avec le **`let rec`**, on peut même obtenir des listes « sans fin ». Par exemple,

```
# let rec liste = 1::4::1::liste;;
val liste : int list = [1; 4; 1; <cycle>]

# let liste_2 = 3::liste;;
val liste_2 : int list = [3; 1; 4; 1; <cycle>]
```

Ces définitions correspondent à la situation suivante :



On remarquera que Caml détecte la boucle dans la liste et indique `<cycle>` plutôt qu'une infinité de termes, même s'il ne précise pas quels sont les termes qui sont répétés.

13. Les noms `hd` et `tl` sont des abbréviations des termes anglais « head » et « tail », désignant respectivement la tête et la queue.
14. Aucun élément n'a été « enlevé », il s'agit bien d'une *partie* de la liste passée en argument.

4.3 Autres fonctions sur les listes

La fonction `List.length` permet d'obtenir le nombre d'éléments qu'elle contient :

```
# List.length;;
- : 'a list -> int = <fun>

# List.length [ 1; 2; 3; 4 ];;
- : int = 4
```

La fonction `List.nth` permet d'obtenir le n^{e} élément de la liste :

```
# List.nth;;
- : 'a list -> int -> 'a = <fun>

# List.nth [ 1; 2; 3; 4 ] 3;;
- : int = 4
```

Attention, cette opération peut être couteuse, car elle implique de parcourir la liste jusqu'à l'élément souhaité (comme c'était le cas pour nos listes chaînées) !

La fonction `List.mem` permet de tester l'appartenance [15] d'un élément dans une liste :

```
# List.mem;;
- : 'a -> 'a list -> bool = <fun>

# List.mem 42 [ 1; 2; 3; 4 ];;
- : bool = false
```

Enfin, la fonction `List.rev` permet d'obtenir une nouvelle liste contenant les mêmes éléments que la liste passée en argument, mais dans l'ordre inverse :

```
# List.rev;;
- : 'a list -> 'a list = <fun>

# List.rev [ 1; 2; 3; 4 ];;
- : int list = [4; 3; 2; 1]
```

4.4 Écrire des fonctions sur les listes

Créer une fonction sur une liste est très similaire avec ce que nous avons écrit avec nos propres listes chaînées. On utilisera ainsi largement le filtrage par motif.

15. Ou, pour être plus précis, l'égalité entre un élément de la liste et l'élément fourni en paramètre.

L'opérateur [16] `::` permet d'extraire aisément les éléments pertinents. En effet, le motif `t::q` est reconnu par n'importe quelle liste non-vide, `t` désignant ensuite l'élément en tête de liste, et `q` le reste de la liste.

Par exemple, la fonction `List.hd`, retournant le premier élément d'une liste peut s'implémenter de la sorte :

```
# let tete = function
    | []   -> failwith "Liste vide"
    | t::_ -> t;;

val tete : 'a list -> 'a = <fun>
```

De même, pour obtenir la liste privée de son premier élément :

```
# let queue = function
    | []   -> failwith "Liste vide"
    | t::q -> q;;

val queue : 'a list -> 'a list = <fun>
```

On peut utiliser plusieurs « conse » dans un filtrage. Par exemple, la fonction suivante permet d'obtenir le deuxième élément d'une liste :

```
# let deuxieme = function
    | t1::t2::q -> t2
    | _         -> failwith "Liste trop courte";;

val deuxieme : 'a list -> 'a = <fun>
```

Une liste contenant un seul élément peut être reconnue de différente façons, par exemple `t::[]`. On peut ainsi écrire une fonction extrayant le dernier élément d'une liste :

```
# let rec dernier = function
    | t::[] -> t
    | t::q  -> dernier q
    | []    -> failwith "Liste vide";;

val dernier : 'a list -> 'a = <fun>
```

On aurait pu également écrire le premier motif `t::[]` de la sorte :

```
    | [t] -> t
```

16. Il ne s'agit pas en fait d'un opérateur dans cette situation.

On peut de la même façon écrire une fonction retournant la longueur d'une liste :

```
# let rec longueur = function



val longueur : 'a list -> int = <fun>
```

Une fonction retournant le n^e élément d'une liste :

```
# let rec nieme = function




val nieme : 'a list -> int -> 'a = <fun>
```

On peut également définir une fonction `membre` indiquant si le premier élément est présent dans la liste correspondant au second argument :

```
# let rec membre x = function
    | []            -> false
    | t::_ when t=x -> true
    | _::q          -> membre x q;;

val membre : 'a -> 'a list -> bool = <fun>
```

C'est l'occasion de revenir sur quelques points concernant le filtrage. Tout d'abord, un nom ne peut pas faire référence à une valeur dans un motif, aussi ne peut-on écrire :

```
# let rec membre x = function
    | []   -> false
    | x::q -> true              (* <- Attention, incorrect ! *)
    | _::q -> membre x q;;
```

Ou plus précisément, on « peut » l'écrire, mais cela ne correspond pas à ce que l'on pourrait espérer, comme en témoignent l'avertissement et la signature de la fonction :

```
Characters 86-90:
    | _::q -> membre x q;;
      ^^^^
Warning 11: this match case is unused.

val membre : 'a -> 'b list -> bool = <fun>
```

En effet, le nom « x » qui apparaît dans le motif filtrage est distinct du nom x qui identifie le premier paramètre (un nom apparaîssant dans un motif de filtrage est un nom qui n'existe que le temps du motif et de sa conséquence, et qui masquera tout autre nom identique).

Enfin, pour une fonction qui concatène deux listes :

```
# let rec concat lst1 lst2 =
    match lst1 with
      | []   -> lst2
      | t::q -> t::(concat q lst2);;

val concat : 'a list -> 'a list -> 'a list = <fun>
```

4.5 Coût en temps des opérations sur les listes

L'une des questions que l'on se posera souvent à l'avenir est le « coût », notamment en temps de calcul, d'une fonction.

Prenons par exemple le cas de la fonction `List.mem` (ou notre équivalent, `membre`). Dans le pire des cas, elle devra examiner les éléments de la liste un par un pour les comparer à l'élément recherché, aussi le temps de calcul, dans une telle situation, sera proportionnel à la longueur de la liste.

Il en est de même pour la fonction `List.length` (ou notre équivalent, `longueur`), déterminant la taille d'une liste. Dans *tous* les cas, cette fois, il faudra parcourir la totalité de la liste pour connaître sa longueur.

Les fonctions `List.hd` et `List.tl`, elles demandent un temps constant, que la liste contienne dix, cent ou cent mille éléments. En effet, les valeurs qu'elles retournent sont directement accessibles dans la première « cellule » de la liste.

En revanche, accéder à un élément au milieu (ou à la fin) de la liste avec `List.nth` est, comme on l'a déjà dit, d'autant plus coûteux qu'il est loin [17].

Il existe un ensemble de notations pour qualifier et manipuler plus facilement les coûts (en terme de temps de calcul, mais pas simplement), d'une fonction, d'un algorithme ou d'un programme.

On dira qu'une fonction travaillant sur un ensemble de n données a une complexité maximale en n en temps de calcul (et on notera cette complexité $O(n)$) si et seulement s'il existe $n_0 \in \mathbb{N}$ et $a \in \mathbb{R}^{+*}$ tels que, pour tout $n \geq n_0$, le temps $t(n)$ d'exécution de la fonction vérifie $t(n) \leqslant a \times n$ quelles que puissent être les n données à traiter.

17. Ce qui est très différent des listes en Python, pour lesquelles accéder à un élément est immédiat quelle que soit sa position (obtenir la taille d'une liste est également immédiat, sans besoin de la parcourir)... Dans d'autres situations, la méthode choisie par Caml pour représenter une liste conduira à des opérations plus efficaces, il y a donc des choix à faire en fonction des situations.

Cette notation pour majorer le coût fonctionne exactement comme la notation identique que vous avez peut-être déjà croisée en mathématiques. $O(n)$ n'est bien évidemment pas le seul majorant possible, les plus courants étant :

- $O(n)$ (ou *linéaire*) : le temps de calcul peut être majoré (pour n assez grand) par une fonction proportionnelle à la quantité de données à traiter ; c'est par exemple le cas de la recherche d'un maximum ou d'un minimum dans une liste, puisque l'on considère chacun des termes un à un
- $O(n^2)$ (ou *quadratique*) : le temps de calcul peut être majoré (pour n assez grand) par une fonction proportionnelle au carré du nombre d'éléments constituant les données ; par exemple dans le cas du tri sélection
- $O(1)$ (ou *constant*) : le temps de calcul peut être majoré par une constante qui ne dépend pas de la quantité de données à traiter ; c'est un cas assez rare, mais une fonction qui retourne le premier élément d'une liste a par exemple une complexité en $O(1)$
- $O(\log(n))$ (ou *logarithmique*) : le temps de calcul peut être majoré (pour n assez grand) par une fonction proportionnelle au logarithme du nombre de données à traiter ; c'est le cas de la recherche dichotomique dans une liste triée qui a été étudiée l'an dernier
- $O(n \times \log(n))$: le temps de calcul peut être majoré par une fonction proportionnelle au produit du nombre de données à traiter par le logarithme de ce même nombre ; c'est un cas relativement courant, et nous allons en voir de suite un exemple
- $O(k^n)$: le temps de calcul peut être majoré (pour n assez grand) par une fonction proportionnelle à k^n (k étant une constante donnée, fréquemment égale à 2)

Ce n'est pas une liste exhaustive, seulement les situations les plus courantes que l'on rencontre. Précisons qu'une fonction dont la complexité en temps est en $O(n)$ a, *de facto*, aussi une complexité en $O(n^2)$, puisque de façon évidente $n \leqslant n^2$. Lorsque l'on précise la complexité, on choisit le *plus « petit »* majorant possible. Parmi celles citées précédemment, ordonnées de la meilleure (algorithme le plus rapide pour de grands n) à la moins bonne, on trouve :

$$O(1) \preceq O(\ln(n)) \preceq O(n) \preceq O(n\ln(n)) \preceq O(n^2) \preceq O(k^n)$$

Attention toutefois, la complexité d'un algorithme peut changer d'un langage à l'autre, ou d'une machine à l'autre. Comme on a déjà eu l'occasion de le souligner, obtenir la longueur d'une liste, accéder à son dernier élément, ou insérer un élément dans une liste n'a pas du tout le même coût en Caml et, par exemple, en Python.

De façon générale, pour déterminer la complexité d'un algorithme, il suffit d'estimer, en fonction de n, le nombre de fois qu'est effectué l'instruction qui est exécutée le plus souvent par le programme (par exemple, dans le cas présent, une comparaison) et de ne conserver que le terme d'ordre le plus élevé (en prenant garde cependant que certaines opérations ne nécessitent pas, elles-mêmes, un temps constant).

Prenons par exemple le cas de la concaténation des lists `[ 1; 2; 3 ]` et `[ 4; 5 ]` avec notre fonction `Concat`. Si l'on décompose les appels, les choses se passent de la façon suivante :

```
concat [ 1; 2; 3 ] [ 4; 5 ]
1 :: ( concat [ 2; 3 ] [ 4; 5 ] )
1 :: ( 2 :: ( concat [ 3 ] [ 4; 5 ] ) )
1 :: ( 2 :: ( 3 :: concat ( [ ] [ 4; 5 ] ) ) )
1 :: ( 2 :: ( 3 :: [ 4; 5 ] ) )
1 :: ( 2 :: [ 3; 4; 5 ] )
1 :: [ 2; 3; 4; 5 ]
[1; 2; 3; 4; 5]
```

On utilise trois fois le motif de filtrage pour extraire les trois éléments de la liste utilisée comme premier argument, avant d'utiliser le second motif de notre fonction `Concat`, puis on utilise trois fois également l'opérateur `::` pour recoller chacun des éléments à la liste utilisée comme second argument. On effectue également quatre appels à `concat`, et autant de filtrages.

Le coût en terme de calcul de cette fonction est donc une fonction affine de la longueur de la liste de gauche. Si l'on note n le nombre d'éléments de cette liste, on dira que la fonction a un coût en $O(n)$.

Parfois, on peut faire mieux que simplement majorer le temps de calcul, et l'encadrer. S'il existe $n_0 \in \mathbb{N}$ et deux réels strictement positifs a et b tels que pour tout ensemble de n données avec $n > n_0$, le temps de calcul est compris entre $a \times n$ et $b \times n$, on dira que la complexité est en $\Theta(n)$.

Si l'on reprend les fonctions que l'on a déjà présentées sur les listes, en notant n le nombre d'éléments dans la liste :

- `List.hd` et `List.tl` ont une complexité en temps en $\Theta(1)$;
- `List.length` a une complexité en temps en $\Theta(n)$;
- `List.rev` a une complexité en temps en $\Theta(n)$;
- `List.mem` a une complexité en temps en $O(n)$ (on n'a pas besoin de parcourir toute la liste si on trouve l'élément recherché parmi les premiers) ;
- `List.nth` a une complexité en temps en $\Theta(k)$ où k est l'indice de l'élément recherché ;
- l'opérateur `::` a une complexité en temps en $\Theta(1)$;
- l'opérateur `@` a une complexité en temps en $\Theta(n)$ où n est le nombre d'éléments de la liste *de gauche*.

4.6 Fonctionnelles agissant sur les listes

Outre le filtrage, un certain nombre de mécanismes permettent d'écrire plus simplement des opérations sur des listes, en appliquant, de diverses façons, une fonction donnée à tous les éléments d'une liste.

Il convient de s'en servir *avec parcimonie*, car si ces écritures peuvent régulièrement simplifier les expressions, elles peuvent tout aussi bien les rendre illisibles, surtout sans un mot d'explication !

La fonction `List.iter`

Le mécanisme le plus simple est fourni par la fonction `List.iter`, qui prend en argument une fonction et une liste, la fonction devant accepter des éléments de même type que ceux présents dans la liste. Cette fonction est alors appliquée à tous les éléments de la liste, de gauche à droite :

```
# let ma_liste = [ 1; 2; 3; 4; 5 ];;
val ma_liste : int list = [1; 2; 3; 4; 5]

# List.iter print_int ma_liste;;
12345- : unit = ()
```

Puisqu'une fonction ne peut retourner qu'un seul élément, la fonction `List.iter` prend en paramètre des fonctions retournant un type `unit` (comme `print_int`) :

```
# List.iter;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

Bien évidemment, la fonction `List.iter` n'a d'intérêt que si la fonction passée en argument a un effet de bord sur l'environnement, comme par exemple un affichage !

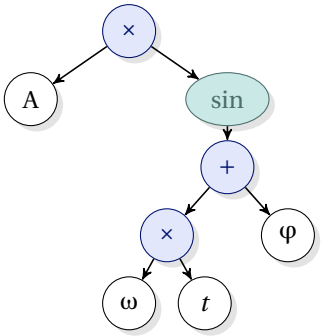
La fonction `List.map`

Si la fonction que l'on veut utiliser sur tous les éléments de la liste retourne un résultat, on peut vouloir récupérer les résultats de l'application de la fonction à tous les éléments de la liste. Pour ce faire, on dispose de la fonction `List.map` qui attend une fonction et une liste :

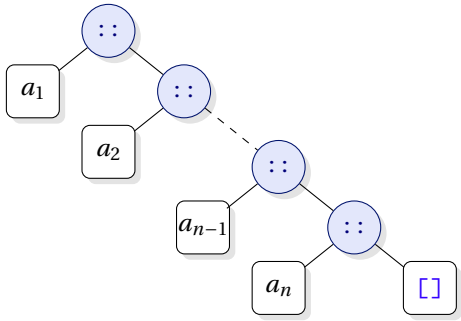
```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>

# let f n = float_of_int n ** 2.0;;
val f : int -> float = <fun>

# List.map f [ 1; 2; 3; 4; 5 ];;
- : float list = [1.; 4.; 9.; 16.; 25.]
```

À la liste $[a_1, a_2, \ldots a_n]$, on associe donc la liste $[f(a_1), f(a_2), \ldots f(a_n)]$.

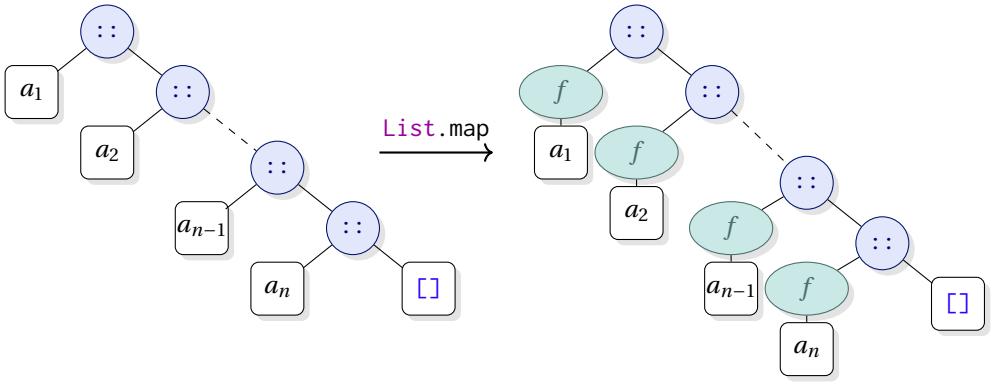
Afin de mieux comprendre le fonctionnement de `List.map`, remarquons que l'on peut considérer l'expression $A\sin(\omega t + \phi)$ sous une forme arborescente :



Ainsi, pour calculer $A\sin(\omega t + \phi)$, on calcule le produit de ω avec t, puis on ajoute ϕ ; on prend le sinus du résultat, et on multiplie enfin le tout par A. De la même façon, une liste est simplement le résultat d'utilisations successives de `::` sur une liste vide `[]`, insérant un par un les éléments par la gauche. Ainsi, une liste $[a_1, a_2, \ldots a_n]$ peut être représentée par :



L'utilisation de `List.map` correspond donc à la transformation ci-dessous, dans laquelle on a inséré la fonction `f` entre chacun des éléments de la liste et les « conse » :



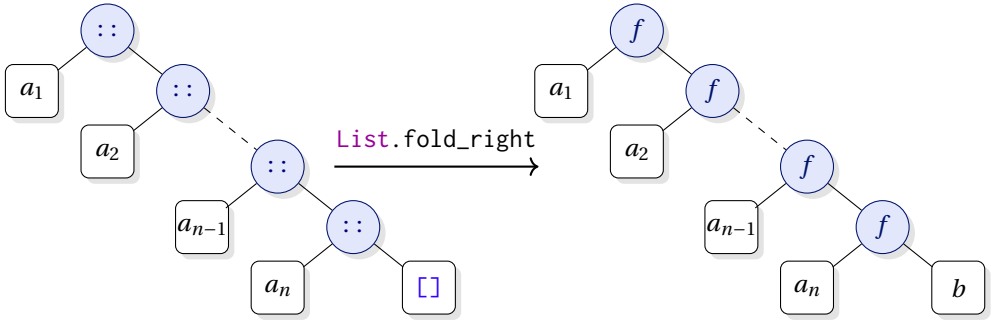
Tout se passe comme si la fonction `List.map` était définie de la sorte :

```
# let rec map f = function
    | []   -> []
    | t::q -> (f t)::(map f q);;

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

La fonction `List.fold_right`

La fonction `List.fold_right`, quant à elle, prend en argument une fonction f attendant deux arguments de types différents, une liste $[a_1, a_2, ...a_n]$ d'éléments du premier type et un élément b du second, et retourne $f\left(a_1, f\left(a2, f\left(...\left(a_n - 1, f(a_n, b)\right)...\right)\right)\right)$.

Elle effectue donc la transformation suivante :



Sa signature est la suivante (on remarquera au passage que les a_i et l'élément b ne sont pas nécessairemement de même type) :

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Cette fonction pourrait être implémentée en Caml de la façon suivante :

```
let rec fold_right f lst b =
    match lst with
      | []   -> b
      | t::q -> f t (fold_right f q b);;

val plie_droite : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

La fonction `List.fold_right`, *utilisée à bon escient*, permet de simplifier l'écriture de nombreux algorithmes opérant sur des listes. On évite ainsi la nécessité d'écrire explicitement une récursion (ou, on le verra, une boucle).

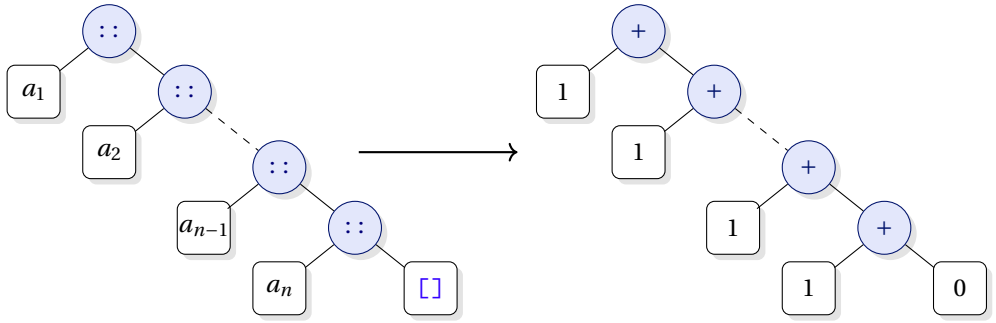
On pourra, par exemple, obtenir la somme des éléments d'une liste `L` d'entiers avec :

```
# let somme a b = a+b
    in List.fold_right somme [ 1; 2; 3; 4; 5 ] 0;;
- : int = 15
```

Pour déterminer la longueur d'une liste `L`, on peut écrire :

```
# let compte a b = b+1
    in List.fold_right compte [ 1; 2; 3; 4; 5 ] 0;;
- : int = 5
```

Pourquoi « `compte a b = b+1` »? Simplement parce que, pour chaque élément a_i de la liste, sa longueur (initialisée à 0 pour `[]`) augmente de 1, et cela quelle que soit la valeur de a_i. En d'autres termes, la transformation réalisée ici peut être résumée par le schéma suivant :



On peut s'en servir pour définir des fonctions, par exemple `somme_liste` :

```
# let somme_liste lst =
    List.fold_right (fun a b -> a+b) lst 0;;

val somme_liste : int list -> int = <fun>
```

Ou bien encore `longueur` :

```
# let longueur lst =
    List.fold_right (fun a b -> b+1) lst 0;;

val longueur : 'a list -> int = <fun>
```

Si l'on veut obtenir le plus grand élément d'une liste, le choix du troisième argument de `List.fold_right` est un peu plus délicat.

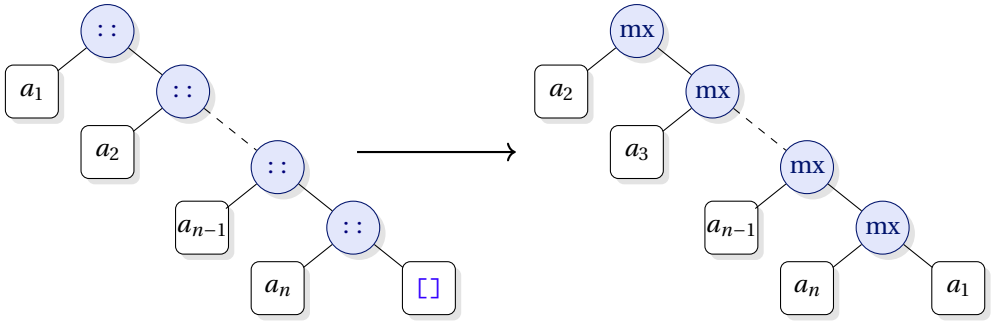
Habituellement, en programmation impérative, pour déterminer le plus grand élément d'une liste, il est d'usage de partir d'un élément quelconque de la liste. On peut de la même façon prendre ici la tête de la liste comme « élément b » (et n'appliquer `List.fold_right` qu'à la queue de la liste puisque le premier élément a déjà été pris en compte).

On définira donc `maximum_liste` ainsi :

```
# let maximum_liste lst =
    List.fold_right max (List.tl lst) (List.hd lst);;

val maximum_liste : 'a list -> 'a = <fun>
```

Débuter le « repliement » par un élément de la liste permettra par ailleurs de conserver le caractère polymorphe de la fonction `max`, et il est possible d'obtenir le plus grand élément de listes contenant n'importe quel type d'éléments (entiers, flottants, caractères, chaînes de caractères, etc.).

La transformation que l'on a effectué est donc :



Signalons enfin que si le type de `b` est une liste, on peut parfaitement utiliser `List.fold_right` pour obtenir une liste. Par exemple, il est possible de redéfinir `List.map` :

```
# let rec map f lst =
    List.fold_right (fun a b -> (f a)::b) lst [];;

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

La fonction `List.fold_left`

La fonction `List.fold_right` a un pendant, `List.fold_left`, qui réalise une transformation très similaire, mais associe à la liste $[a_1, a_2, ...a_n]$ et un élément b le résultat de $f\left(f\left(...\left(f(b, a_1),\right), a_2, ...\right), a_n\right)$. Elle est équivalente à la fonction suivante :

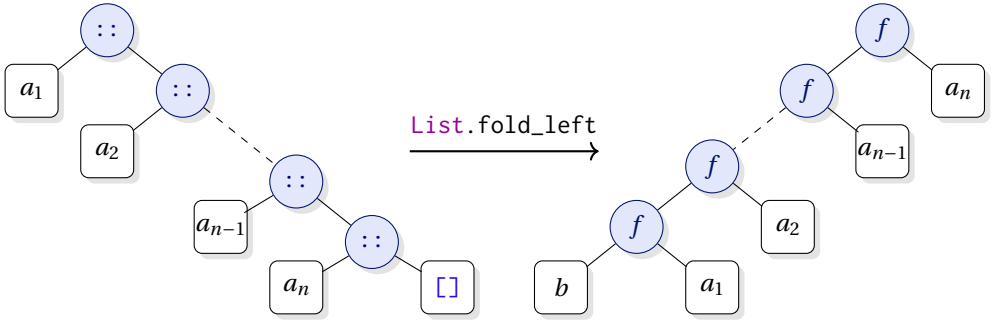
```
let rec fold_left f b = function
    | []   -> b
    | t::q -> fold_left f (f b t) q;;

val plie_gauche : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Attention, l'ordre des paramètres n'est pas le même que `List.fold_right`, la liste vient cette fois en troisième et dernier paramètre. De la même façon, la fonction passée en paramètre doit prendre en *second* argument les éléments a_i de la liste (et en premier argument, des éléments du type de `b`).

On prendra également garde au fait que, du fait de cette inversion, dans la signature, les a_i sont donc de type `'b` et b est de type `'a` :

```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Cette fonction effectue donc la transformation suivante :



Le fait que la liste se trouve en dernier paramètre permet de définir encore plus simplement une fonction sommant les éléments :

```
# let somme_liste =
    List.fold_left (fun b a -> b+a) 0;;

val somme_liste : int list -> int = <fun>
```

Ou calculant la longueur de la liste [18] :

```
# let longueur =
    List.fold_left (fun b a -> b+1) 0;;

val longueur : '_a list -> int = <fun>
```

Nous verrons un peu plus tard que la fonction `List.fold_left` est un peu plus performante que `List.fold_right`.

Les deux fonctions existent car elles ne répondent pas tout à fait aux mêmes besoins. Il est un peu délicat d'écrire une fonction `List.map` à partir de `List.fold_left` alors que la chose était facile avec `List.fold_right`. La raison en est que l'opérateur `::` ajoute les éléments à gauche, or `List.fold_left` tend à retourner la liste.

`List.fold_left` permet en revanche aisément de retourner une liste :

```
# let retourne =
    List.fold_left (fun lst e -> e::lst) [];;

val retourne : '_a list -> '_a list = <fun>
```

Ce retournement serait plus difficile à obtenir avec `List.fold_right`.

À propos des langages fonctionnels

Les fonctions `List.map`, `List.fold_left` et `List.fold_right` sont présentes dans la quasi-totalité des langages fonctionnels. Parfois, seul un équivalent de `List.fold_left` est disponible (appelé `reduce` en Clojure, Ruby, D, `fold` en F#, etc.) et on se sert d'un renversement de liste pour obtenir l'alternative.

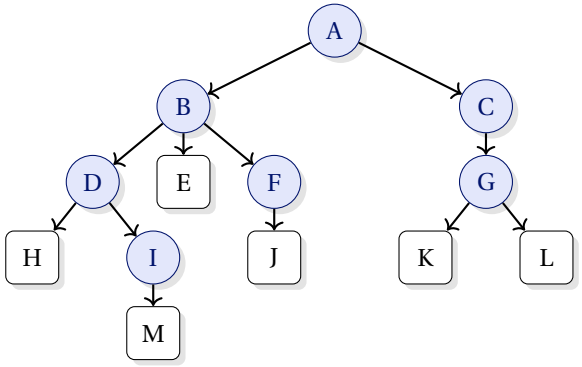
C'est d'ailleurs le cas en Python, qui, bien que n'étant pas à l'origine un langage fonctionnel, dispose néanmoins d'une fonction `map` (quoi qu'elle fasse double emploi avec le mécanisme de compréhension de liste, plus puissant) et d'une fonction `functools.reduce` qui se comporte comme `List.fold_left`.

18. On remarquera une bizarrerie dans la signature (parfois, on verra également `'_weak1` en lieu et place de `'_a`, ce qui est équivalent), sur laquelle nous reviendrons quelque peu ultérieurement : la fonction obtenue n'est pas totalement polymorphe, elle accepte des listes contenant un type quelconque, mais la première utilisation « fixera » ce type. Par exemple, après avoir calculé `longueur [ 1; 2 ]`, `longueur` sera de type `int list -> int`. Il n'est pas possible de définir une fonction polymorphe à partir d'une application partielle. Pour éviter ce problème, on fera explicitement apparaître le troisième argument dans la définition de la fonction. Les raisons de cette subtilité dépassent grandement le cadre de ce cours.

5 Les arbres

5.1 Présentation

Outre les listes, une autre structure de données est très utile en informatique : les arbres. Un arbre est formellement défini comme un graphe connexe, acyclique et orienté (la définition précise de ces termes sera abordée en seconde année). Sans entrer dans ces termes techniques pour le moment, un arbre comme une structure telle que la suivante, similaire à celles que l'on a déjà manipulées :



Une partie du vocabulaire pour décrire ces structures est emprunté aux arbres généalogiques : chacun des éléments de l'arbre, à l'exception de l'un d'entre eux, a un unique *parent* (ou *père*, *antécédent*). Ils peuvent avoir un ou plusieurs *enfants* (ou *fils*). C'est l'orientation des arêtes reliant les éléments qui compte ici, indépendamment du sens de représentation de l'arbre, même si généralement les arêtes sont toutes dirigées dans le même sens, vers le bas. Enfin, on appelle *descendants* d'un élément l'ensemble de ses fils, des fils de ses fils, et ainsi de suite. Et *ascendants* l'ensemble de son père, du père de son père, et ainsi de suite.

D'autres termes sont empruntés à la botanique. L'élément orphelin à partir duquel on peut accéder à tous les autres (ici A) est appelé *racine*, quand bien même il se trouve souvent en haut au mépris de toutes les règles de la botanique. Les éléments dépourvus d'enfants (ici H, M, E, J, K et L) sont appelés *feuilles* (ou parfois *nœuds extérieurs*). Les autres éléments sont qualifiés de *nœuds* (ou *nœuds internes*). Enfin, un chemin de proche en proche dans un arbre en suivant les arêtes orienté est appelé une *branche*.

Le nombre de descendants d'un nœud est appelé *arité* du nœud. On remarquera enfin que n'importe quel nœud est la racine d'un arbre constitué de lui-même et de ses descendants. On qualifiera cet arbre de *sous-arbre* de l'arbre initial.

Les arbres sont utiles car ils permettent de décrire de très nombreux objets. Par exemple, comme nous l'avons déjà vu, des expressions mathématiques telles que $A\sin(\omega t + \phi)$, ou une liste `[ 1; 3; 5; 4; 2 ]` (qui correspond en fait à `1::3::5::4::2::[]`).

5.2 Représenter un arbre

Compte tenu des structures très différentes que peuvent avoir les arbres, par exemple en terme d'arité des nœuds, que l'on aura à manipuler, il n'existe pas de structure toute prête en Caml pour représenter un arbre. Il n'est cependant pas bien difficile de créer de telles structures.

Comme une liste chaînée est décrite comme un élément auquel est éventuellement accroché la suite de la liste, un arbre peut être considéré comme un élément auquel est associé un ensemble, éventuellement vide, de sous-arbres.

On peut par exemple définir le type suivant pour représenter un arbre :

```
type 'a arbre = { element : 'a ;
                  fils : 'a arbre list };;
```

On remarquera que tous les éléments de l'arbre, dans le cas présent, sont nécessairement de même type.

L'arbre pris précédemment en exemple peut alors être décrit par :

```
# let ex = { element="A"; fils=[
              { element="B"; fils=[
                  { element="D"; fils=[
                      { element="H"; fils=[] } ;
                      { element="I"; fils=[
                          { element="M"; fils=[] } ]} ]} ;
                  { element="E"; fils=[] };
                  { element="F"; fils=[
                      { element="J"; fils=[] } ]} ]} ;
              { element="C"; fils=[
                  { element="G"; fils=[
                      { element="K"; fils=[] } ;
                      { element="L"; fils=[] } ]} ]} ];;
```

Comme pour les listes chaînées, on préférera rapidement écrire des fonctions permettant de construire, étape par étape, un arbre (en créant d'abord des sous-arbres et en les regroupant, nœud par nœud, jusque la racine), plutot que le définir de la sorte!

Souvent, on préférera utiliser un couple plutôt qu'un type produit pour associer l'élément et ses enfants (car cela raccourcit quelque peu les écritures, quand bien même il convient alors de se souvenir de ce à quoi correspondent chacun des éléments du couple), en écrivant par exemple :

```
# type 'a arbre = Noeud of 'a * 'a arbre list;;
```

La déclaration de l'arbre devient alors :

```
# let ex = Noeud ("A", [
             Noeud ("B", [
                 Noeud ("D", [
                     Noeud ("H", [] ) ;
                     Noeud ("I", [
                         Noeud ("M", [] ) ] ) ] ) ;
                     Noeud ("E", [] ) ;
                 Noeud ("F", [
                     Noeud ("J", [] ) ] ) ] ) ;
             Noeud ( "C", [
                 Noeud ("G", [
                     Noeud ("K", [] ) ;
                     Noeud ("L", [] ) ]) ]) ]);;
```

5.3 Propriétés d'un arbre

Définition. La *taille* d'un arbre est le nombre d'éléments qu'il contient (nœuds internes et feuilles). La *hauteur* d'un arbre est la longueur de sa plus grande branche, c'est-à-dire le nombre de *liens* de filiation que celle-ci contient.

En combinant beaucoup de choses vues dans le présent chapitre, on peut facilement calculer la taille d'un arbre, en s'efforçant de rester lisible :

```
# let rec taille = function
    Noeud(_, lst)
      -> let somme_liste = List.fold_left (fun a b -> a+b) 0
         in 1 + somme_liste (List.map taille lst);;

val taille : 'a arbre -> int = <fun>
```

De même que sa hauteur :

```
# let rec hauteur = function
    | Noeud(_, []) -> 0
    | Noeud(_, lst)
      -> let max_liste l = List.fold_left max (List.hd l) (List.tl l)
         in 1 + max_liste (List.map hauteur lst);;

val hauteur : 'a arbre -> int = <fun>
```

5.4 Arbres binaires stricts

Définitions

Dans la suite, on se limitera à certains arbres bien particuliers, les arbres *binaires*, où chaque nœud (interne) a, au plus, deux fils.

On peut utiliser les types précédents pour représenter de tels arbres, mais il est plus utile de créer un type spécifique, à la fois pour éviter de construire par erreur des arbres qui ne sont pas binaires, et pour faciliter l'accès aux enfants d'un nœud.

On s'intéresse dans un premier temps aux arbres binaires *stricts*, où chaque nœud (interne) a *exactement* deux nœuds. On peut donc remplacer la liste des enfants précédente par un couple [19] d'enfants, en définissant le type par

```
# type 'a arbre =
    | Noeud of 'a * 'a arbre * 'a arbre
    | Feuille of 'a;;
```

Comme tous les nœuds ont deux fils (deux sous-arbres), ils ne peuvent convenir pour les feuilles, aussi est-il indispensable d'ajouter spécifiquement un cas pour les feuilles.

Il est à noter que, comme les feuilles sont traitées différemment des nœuds, on peut choisir des types différents pour les objets stockés dans les feuilles et ceux stockés dans les nœuds :

```
# type ('a, 'b) arbre =
    | Noeud of 'a * ('a, 'b) arbre * ('a, 'b) arbre
    | Feuille of 'b;;
```

C'est très utile pour un arbre représentant une expression mathématique, par exemple, où les nœuds contiennent des opérateurs, et les feuilles des valeurs.

Attention, il n'est pas rare de voir les arbres binaires stricts définis, de façon équivalente, par

```
# type ('a, 'b) arbre =
    | Noeud of ('a, 'b) arbre * 'a * ('a, 'b) arbre
    | Feuille of 'b;;
```

les éléments constituant les données d'un nœud ayant été réordonnées pour mieux faire apparaître la position de chacun des sous-arbres par rapport au nœud.

19. Un enregistrement, tel que `Noeud of \{ element: 'a ; filsg: 'a arbre ; filsd: 'a arbre\}`, conviendrait également.

Utilisation

Pour manipuler les arbres binaires, on utilise à nouveau le filtrage. La fonction `Hauteur` déterminant la hauteur de l'arbre pourra s'écrire, pour un arbre binaire strict, de la façon suivante :

```
# let rec hauteur = function
    | Noeud (_, filsg, filsd) -> 1 + max (hauteur filsg) (hauteur filsd)
    | Feuille _               -> 0;;

val hauteur : ('a, 'b) arbre -> int = <fun>
```

Pour obtenir la taille de l'arbre, on aurait :

```
# let rec taille = function
    | Noeud (_, filsg, filsd) -> 1 + taille filsg + taille filsd
    | Feuille _               -> 1;;

val taille : ('a, 'b) arbre -> int = <fun>
```

Pour obtenir le nombre de feuilles, on aurait :

```
# let rec nbFeuilles = function
    | Noeud (_, filsg, filsd) -> nbFeuilles filsg + nbFeuilles filsd
    | Feuille _               -> 1;;

val nbFeuilles : ('a, 'b) arbre -> int = <fun>
```

Pour le nombre de nœuds internes :

```
# let rec nbNoeuds = function
   | Noeud (_, filsg, filsd) -> 1 + (nbNoeuds filsg) + (nbNoeuds filsd)
   | Feuille _               -> 0;;

val nbNoeuds : ('a, 'b) arbre -> int = <fun>
```

5.5 Arbres binaires

Définitions

Dans un arbre binaire, les nœuds ont *au plus* deux fils, et non nécessairement exactement deux. Il est souvent utile de conserver la distinction entre « fils gauche » et « fils droit »,

et les types que nous définirons dans la suite le feront, mais ce n'est pas toujours le cas. Auquel cas, les types seront quelque peu différents.

On l'aura compris, il y a de *très* nombreuses façons de représenter un arbre, selon le problème rencontré. On pourrait envisager d'ajouter deux cas supplémentaires :

```
# type ('a, 'b) arbre =
    | Noeud   of 'a * ('a, 'b) arbre * ('a, 'b) arbre
    | Noeud_G of 'a * ('a, 'b) arbre    (* Seulement un fils à gauche *)
    | Noeud_D of 'a * ('a, 'b) arbre    (* Seulement un fils à droite *)
    | Feuille of 'b;;
```

Une autre possibilité consiste à conserver uniquement des nœuds avec deux fils, et de définir une étiquette « `Nil` » qui, comme le « `Fin` » de notre liste chaînée, indiquerait « il n'y a plus rien par là [20] », et on pourrait vouloir écrire :

```
# type ('a, 'b) arbre =      (* mauvaise définition *)
    | Noeud of 'a * ('a, 'b) arbre * ('a, 'b) arbre
    | Feuille of 'b
    | Nil;;
```

Un nœud avec seulement un fils à droite, par exemple, correspondrait donc à un élément de type `Noeud (val, Nil, filsd)`.

Seulement, il y a cependant un souci avec la définition précédente. Les Shadoks disaient qu'il y avaient trois types de casseroles : celles avec un manche à gauche, celles avec un manche à droite, et celles sans manche, qu'on appelle communément des autobus.

Dans la définition de notre type, on a des nœuds avec uniquement un fils gauche, des nœuds avec uniquement un fils droit, des nœuds avec deux fils... et des nœuds sans aucun fils, qui se trouvent donc être des feuilles !

Comme il est difficile d'empêcher ce dernier cas, c'est le cas `Feuille` que nous allons supprimer, une feuille devenant simplement un `Noeud` dont les deux fils sont `Nil`.

```
# type 'a arbre =
    | Noeud of 'a * 'a arbre * 'a arbre
    | Nil;;
```

Principal inconvénient de cette représentation, on perd la possibilité d'avoir un type différent pour les feuilles et les nœuds.

Dans la suite de ce cours, c'est cette représentation que nous utiliserons.

20. Bien que Caml ne lui donne pas de sens particulier, `Nil` est, en informatique, plus ou moins le terme consacré pour indiquer la terminaison d'une branche ou d'une liste.

Utilisation

Un tel arbre s'utilise avec un filtrage comme ci-dessous, en prenant garde à l'ordre des motifs : le cas de la feuille doit se trouver en premier, et celui du nœud avec deux fils en dernier, car celui-ci accepte n'importe quel objet `Noeud` pour n'importe quels fils, `Nil` compris [21].

```
let foo = function
  | Noeud (val, Nil,   Nil)    -> ... (* Cas d'une feuille *)
  | Noeud (val, filsg, Nil)    -> ... (* Seulement un fils à gauche *)
  | Noeud (val, Nil,   filsd)  -> ... (* Seulement un fils à droite *)
  | Noeud (val, filsg, filsd)  -> ... (* Noeud avec deux fils *)
  | Nil                        -> ... (* Arbre "vide" *)
```

Il n'est généralement pas besoin de traiter les nœuds avec un seul fils à part :

```
# let rec nbFeuilles = function
    | Nil                      -> 0
    | Noeud (_, Nil,   Nil)    -> 1          (* Cas d'une feuille *)
    | Noeud (_, filsg, filsd) -> (nbFeuilles filsd)
                                  + (nbFeuilles filsg);;

val nbFeuilles : 'a arbre -> int = <fun>
```

```
# let rec nbNoeudsInternes = function
    | Nil                      -> 0
    | Noeud (_, Nil,   Nil)    -> 0          (* Cas d'une feuille *)
    | Noeud (_, filsg, filsd) -> 1 + (nbNoeudsInternes filsd)
                                     + (nbNoeudsInternes filsg);;

val nbNoeuds : 'a arbre -> int = <fun>
```

```
# let rec taille = function
    | Nil                      -> 0
    | Noeud (_, Nil,   Nil)    -> 1          (* Cas d'une feuille *)
    | Noeud (_, filsg, filsd) -> 1 + (taille filsd) + (taille filsg);;

val taille : 'a arbre -> int = <fun>
```

Dans certaines situations, et notamment ce dernier cas, il n'est même pas besoin de distinguer le cas des feuilles et des nœuds. Le second motif de filtrage de la fonction `taille` peut tout à fait être supprimé !

21. Sauf évidemment à préciser `filsg <> Nil` et `filsd <> Nil`.

Dans le cas du calcul de la hauteur d'un arbre, la hauteur d'un arbre vide (`Nil`) n'est pas aisée à définir. Mais il est fréquent que l'on choisisse -1 comme hauteur d'un arbre vide, car cela permet d'écrire très simplement :

```
# let rec hauteur = function
    | Nil                      -> -1
    | Noeud (_, filsg, filsd) -> 1 + max (hauteur filsd)
                                         (hauteur filsg);;

val hauteur : 'a arbre -> int = <fun>
```

Quelques propriétés des arbres binaires

Théorème 1.
- *si un arbre binaire a une hauteur h, alors il possède au plus 2^h feuilles, et au plus $2^h - 1$ nœuds (internes);*
- *si un arbre binaire strict possède n nœuds et f feuilles, alors $f = n + 1$.*

De telles propriétés peuvent être démontrées par exemple en utilisant le principe de récurrence sur la hauteur de l'arbre. Pour ce qui est de la première propriété :

Démonstration. Supposons qu'elle soit vraie pour toute hauteur comprise entre 0 et h (nous aurons besoin ici d'une récurrence forte).

Un arbre de taille $h+1$ est constitué d'un nœud, d'un sous-arbre de hauteur égale à h et éventuellement d'un sous-arbre de hauteur h' inférieure ou égale à h.

L'arbre de taille $h+1$ compte donc, au plus, $\left(2^h\right) + \left(2^{h'}\right)$ feuilles, soit au plus 2^{h+1} feuilles puisque $h' \leqslant h$.

De même, il compte au plus $1 + \left(2^h - 1\right) + \left(2^{h'} - 1\right)$ nœuds, soit au plus $2^{h+1} - 1$ nœuds puisque $h' \leqslant h$.

Par ailleurs, la propriété est vraie pour un arbre de hauteur $h = 0$, ne contenant aucun nœud et une seule feuille. D'après le principe de récurrence forte, la propriété est donc vraie pour un arbre de hauteur quelconque. □

Le raisonnement est similaire pour la seconde propriété, qui n'est cependant vérifiée que pour des arbres binaires *stricts* (il est aisé de trouver un contre-exemple pour un arbre binaire qui ne l'est pas, par exemple un arbre avec un seul nœud et une seule feuille).

Démonstration. Supposons qu'elle soit correcte pour n'importe quel arbre binaire *strict* de hauteur comprise entre 0 et h, et considérons un arbre binaire *strict* de taille $h+1$.

Le sous-arbre de gauche, de taille au plus h, contient f_g feuilles et n_g nœuds, avec $f_g = n_g + 1$. Le sous-arbre de droite, de taille également au plus h, contient f_d feuilles et n_d nœuds, avec $f_d = n_d + 1$. On a ici besoin que l'arbre binaire soit *strict* car cela garantit que l'on ait bien un arbre, éventuellement réduit à une simple feuille, de *chaque* côté.

Notre arbre de hauteur $h+1$ contient donc $1 + n_g + n_d$ nœuds, et $f_g + f_d$ feuilles, avec $f_g + f_d = (n_g + 1) + (n_d + 1) = (n_g + n_d + 1) + 1$, ce qui correspond à la propriété pour une hauteur $h+1$.

Un arbre de hauteur 0 contient une feuille et aucun nœud, donc par application du principe de récurrence forte, la propriété est vraie pour toute hauteur h ! □

5.6 Parcours possibles d'un arbre

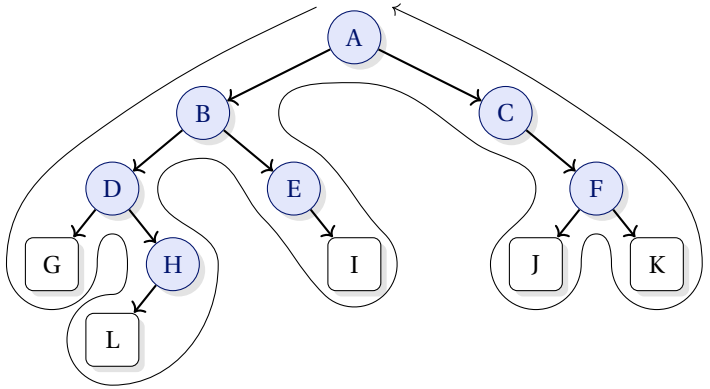
Parcours en profondeur

Terminons ce tour d'horizon des arbres binaires en parlant un peu de parcours des arbres : pour une fonction explorant l'arbre dans son ensemble, comme les fonctions précédentes, dans quel ordre les nœuds sont-ils visités ?

Si l'on y regarde de plus près, dans les cas présentés dans ce chapitre, on a un parcours de l'arbre dit *en profondeur*[22]. C'est-à-dire que l'on explore complètement une branche de l'arbre, jusqu'à la feuille, avant de remonter et d'explorer une autre branche.

En effet, dans le cas de la fonction `hauteur`, par exemple, on évalue tout d'abord la taille du sous-arbre de gauche, ce qui nécessite de le parcourir intégralement, avant de s'intéresser, dans un second temps, au sous-arbre de droite (et enfin de calculer le maximum et d'ajouter un).

Dans un tel parcours en profondeur, tout se passe comme si l'on explorait l'arbre en suivant ses « contours », comme ci-dessous :



Pour tout traitement des nœuds de l'arbre en $O(1)$, l'exploration d'un arbre nécessite naturellement un temps proportionnel à sa taille. Les fonctions récursives sur les arbres

22. On parle de « depth-first » en anglais

s'appellant sur chacun des fils auront en général une complexité correspondant à la complexité du traitement d'un nœud multipliée par la taille de l'arbre.

On peut aisément voir que, dans un parcours en profondeur, chaque feuille est visitée une unique fois. Les nœuds internes, en revanche, sont généralement visités plusieurs fois.

Dans le cas d'un arbre binaire strict, par exemple, ces nœuds internes sont visités trois fois : une première fois avant d'explorer le sous-arbre de gauche, une seconde fois entre l'exploration des deux sous-arbres, et une troisième fois après l'exploration du sous-arbre droit.

Variantes des parcours en profondeur

Dans le cas d'un arbre binaire strict, quand on s'intéresse précisément au moment où sont traités les nœuds internes, on distingue parfois plusieurs variantes de parcours en profondeur :

- le parcours en profondeur dit *suffixe*, où l'on traite d'abord les deux fils, puis le nœud lui-même ;
- le parcours en profondeur dit *préfixe*, où l'on traite d'abord le nœud, puis chacun des deux fils ;
- le parcours en profondeur dit *infixe*, où l'on traite d'abord le fils gauche, puis le nœud, puis le fils droit.

La plupart des fonctions que l'on écrira sur les arbres se trouvent être des parcours suffixes (pour la taille de l'arbre, par exemple, la somme est nécessairement effectuée après l'estimation de la taille des deux sous-arbres ! Il peut cependant arriver que l'on effectue des traitements lors de deux ou lors des trois visites des nœuds internes.

Cette distinction est particulièrement importante si la visite des nœuds a un effet immédiat. C'est par exemple le cas lorsque l'on écrit une fonction affichant l'ensemble des étiquettes de l'arbre.

Dans le cas d'un parcours suffixe, les deux appels affichent toutes les étiquettes des deux sous-arbres, et l'affichage de la racine est effectué en dernier :

```
# let rec parcoursSuffixe = function
    | Nil -> ()
    | Noeud (v, filsg, filsd) -> parcoursSuffixe filsg;
                                 parcoursSuffixe filsd;
                                 print_string v;;

val parcoursSuffixe : string arbre -> unit = <fun>

# parcoursSuffixe arbre
GLHDIEBJKFCA- : unit = ()
```

Dans le cas d'un parcours préffixe, la racine est traitée en premier :

```
# let rec parcoursPréfixe = function
    | Nil -> ()
    | Noeud (v, filsg, filsd) -> print_string v;
                                 parcoursPréfixe filsg;
                                 parcoursPréfixe filsd;;

val parcoursPréfixe : string arbre -> unit = <fun>

# parcoursPrefixe arbre
ABDGHLEICFJK- : unit = ()
```

Dans le cas d'un parcours infixe, tout se passe comme si on avait « compressé » l'arbre verticalement :

```
let rec parcoursInfixe = function
    | Nil -> ()
    | Noeud (v, filsg, filsd) -> parcoursInfixe filsg;
                                 print_string v;
                                 parcoursInfixe filsd;;

val parcoursInfixe : string arbre -> unit = <fun>

# parcoursInfixe arbre
GDLHBEIACJFK- : unit = ()
```

Ces situations ne sont pas mutuellement exclusives, il n'est pas inhabituel que l'on ait, pour un nœud interne, à la fois un traitement préfixe et un traitement suffixe.

Parcours en largeur

Un autre parcours possible d'un arbre est le parcours *hiérarchique* (ou *en largeur*) [23], qui consiste à traiter les éléments de l'arbre « étage par étage ». Le plus souvent de la racine vers les feuilles, autrement dit dans l'ordre ABCDEFGHIJKL.

Ce parcours peut être programmé, mais il nécessite l'utilisation d'une structure de donnée appelée *file* que nous introduirons un peu plus tard. Nous reviendrons à cette occasion sur le principe du parcours hiérarchique d'un arbre.

23. « broadth-first » en anglais.

Exercices

Ex. 2.1 – Gérer les éléments d'une liste

1. Sur le modèle de la fonction `dernier`, écrire une fonction `avantDernier` prenant en argument une liste Caml et retournant son avant-dernier élément. On déclenchera une erreur (avec `failwith`) si la liste contient moins de deux éléments.

2. Écrire une fonction `retire` prenant en argument une liste et un entier n, et retourne une liste identique à la liste fournie, mais privée du n^e élément. On déclenchera une erreur si la liste est trop courte.

3. Écrire une fonction `insère` prenant en argument une liste, un entier n et un élément, et retourne une liste identique à la liste fournie, mais dans laquelle on a inséré l'élément fourni juste avant le n^e élément de la liste passée en argument.

4. Déterminer la complexité (en temps) de chacune de ces fonctions. Comparer ces complexités avec les équivalents de ces fonctions en Python.

Ex. 2.2 – Réordonner les éléments d'une liste

1. Écrire une fonction `rotG` de complexité linéaire prenant en argument une liste et retourne une liste dans laquelle est éléments ont subi une permutation circulaire vers la gauche. Par exemple, `rotG [ 1; 2; 3; 4 ]` doit donner `[ 2; 3; 4; 1 ]`.

2. Écrire une fonction `rotD` de complexité linéaire prenant en argument une liste et retourne une liste dans laquelle est éléments ont subi une permutation circulaire vers la droite. Par exemple, `rotD [ 1; 2; 3; 4 ]` doit donner `[ 4; 1; 2; 3 ]`.

Ex. 2.3 – Préfixes et suffixes

1. Proposer une fonction `suffixes` prenant en argument une liste d'éléments (de type quelconque) et retournant la liste de ses suffixes. Par exemple, `suffixes [ 1; 2; 3 ]` doit retourner par exemple `[ [ 1; 2; 3 ] ; [ 2; 3 ] ; [ 3 ] ]` (l'ordre des listes dans la liste fourni comme résultat n'a pas d'importance).

2. Un peu plus difficile, proposer une fonction `prefixes` prenant en argument une liste d'éléments (de type quelconque) et retournant la liste de ses préfixes. Par exemple, `prefixes [ 1; 2; 3 ]` doit retourner par exemple `[ [ 1 ] ; [ 1; 2 ] ; [ 1; 2; 3 ] ]` (l'ordre des listes dans la liste fourni comme résultat n'a pas d'importance).

3. Quelle est la complexité de ces deux fonctions ?

Ex. 2.4 – Suppression de doublons

1. Quelle est la signature de la fonction suivante, et que fait-elle ?

```
let rec foo f = function
  | t::q when f t -> t::(foo f q)
  | t::q          -> foo f q
  | _             -> [];;
```

2. En déduire une fonction prenant en argument une liste et retournant une liste dans laquelle on a retiré les éléments n'apparaissant pas pour la dernière fois. Le résultat de la fonction sur la liste `[ 1; 2; 3; 2; 4; 5; 4 ]` devra être la liste `[ 1; 3; 2; 5; 4 ]`. Quelle est sa complexité ?

Ex. 2.5 – Factorielle

1. Proposer une fonction `multiplie` qui prend en argument une liste d'entiers et retourne le produit de ses éléments (ou 1 si la liste est vide). On pourra réfléchir à une version utilisant le filtrage, et une version utilisant `List.fold_left`.

2. Écrire une fonction prenant un entier n et retournant la liste des entiers de n à 2 (inclus, rangés par ordre décroissants) si $n \geqslant 2$ et une liste vide sinon.

3. En déduire une fonction `fact` prenant en argument un entier n et retournant sa factorielle.

Ex. 2.6 – Booléens et listes

Caml fournit deux fonctions de signature `('a -> bool) -> 'a list -> bool`, nommées `List.exists` et `List.for_all`, prenant donc en argument une fonction et une liste, et retournant un booléen indiquant

- si la fonction retourne `true` pour au moins un élément de la liste dans le cas de la fonction `List.exists`[24] ;
- si la fonction retourne `true` pour tous les éléments de la liste dans le cas de la fonction `List.for_all`[25].

1. Proposer des fonctions Caml réalisant ces deux fonctions, en procédant dans un premier temps par filtrage, puis en utilisant `List.fold_left`.

2. Écrire une fonction de signature `('a -> bool) -> 'a list -> 'a` retournant le premier élément de la liste vérifiant la propriété (définie par la fonction utilisée comme premier paramètre).

3. Écrire une fonction de signature `('a -> bool) -> 'a list -> int -> 'a` retournant le n^e élément de la liste vérifiant la propriété.

4. Écrire une fonction de signature `('a -> bool) -> 'a list -> 'a` retournant le dernier élément de la liste vérifiant la propriété.

24. L'équivalent en Python serait `any(fun(x) for x in liste)`.
25. L'équivalent en Python serait `all(fun(x) for x in liste)`.

Ex. 2.7 – Produit cartésien

Proposer une fonction de signature `'a list -> 'b list -> 'a * 'b list` prenant en argument deux listes et retournant une liste de couples résultat du produit cartésien des deux listes fournies en argument (les couples se trouvant dans un ordre quelconque). `produit [ 'a'; 'b'; 'c' ] [ 1; 2; 3 ]` retournera ainsi par exemple :

```
- : (int * char) list = [(1, 'a'); (1, 'b'); (1, 'c');
(2, 'a'); (2, 'b'); (2, 'c'); (3, 'a'); (3, 'b'); (3, 'c')]
```

Ex. 2.8 – Fonctionelle `List.iter`

Pour écrire une fonction réalisant la même tâche que `List.iter`, est-il plus simple d'utiliser `List.fold_left` ou `List.fold_right` ? Proposer une telle fonction.

Ex. 2.9 – Maxima dans un arbre

On considère un arbre binaire strict, dont les noeuds et feuilles contiennent des éléments `'a` pouvant être comparés, et dont le type est

```
type 'a arbre =
  | Feuille of 'a
  | Noeud of 'a * 'a arbre * 'a arbre
```

1. Proposer une fonction `maxFeuille` de signature `'a arbre -> 'a` prenant un arbre en paramètre et retournant le plus grande feuille de l'arbre.

2. Créer de même une fonction `maxArbre` de signature `'a arbre -> 'a` prenant un arbre en paramètre et retournant le plus grand élément de l'arbre.

Ex. 2.10 – Élagage d'un arbre

On considère des arbres binaire sans étiquettes définis par le type

```
type arbre = Nil | Noeud of arbre * arbre
```

Écrire une fonction prenant en argument un tel arbre et un entier n, et retournant un arbre dont les branches ont été coupées à la longueur n (les nœuds à la profondeur n devenant des feuilles, les éléments situés plus loin de la racine étant ignorés).

Ex. 2.11 – Symétrie et arbres

On considère des arbre binaire sans étiquettes, définis par le type

```
type arbre = Nil | Noeud of arbre * arbre
```

1. Écrire une fonction `identiques` prenant en argument deux arbres et retournant `true` si les deux arbres sont identiques (on s'intéresse ici à leur forme).

2. Écrire une fonction `miroirs` prenant en argument deux arbres et retournant `true` si les deux arbres sont image l'un de l'autre par une symétrie verticale.

3. Écrire une fonction `symetrique` prenant en argument un arbres et indiquant si l'arbre admet une symétrie verticale.

Ex. 2.12 – Numérotation d'un arbre

La numérotation des nœuds d'un arbre binaire strict de Sosa-Stradonitz vise à associer à chaque nœud de l'arbre un entier unique strictement positif, de façon à pouvoir identifier sans ambiguité ledit nœud. Elle fonctionne de la façon suivante :

- on associe 1 à la racine de l'arbre ;
- si un nœud est numéroté n, alors son fils gauche est numéroté $2n$ et son fils droit $2n+1$.

On suppose le type `arbre` défini de la façon suivante :

```
type 'a arbre = Feuille of 'a | Noeud of 'a * 'a arbre * 'a arbre
```

1. Proposer une fonction `numeroteArbre` dont la signature Caml serait `'a arbre -> (int * 'a) arbre` qui crée un nouvel arbre dans lequel, à chaque élément (nœud ou feuille) de l'arbre fourni en paramètre, on adjoint son numéro.

2. Justifier que n'importe quel entier positif correspond potentiellement à un nœu dans un arbre, et écrire une fonction `chemin` prenant en argument un entier et retournant une liste de caractères indiquant le chemin de la racine au nœud correspondant, en utilisant `'g'` pour indiquer un passage au fils gauche, et `'d'` pour le fils droit. Par exemple, `chemin 5` retournera `[ 'g'; 'd' ]`.

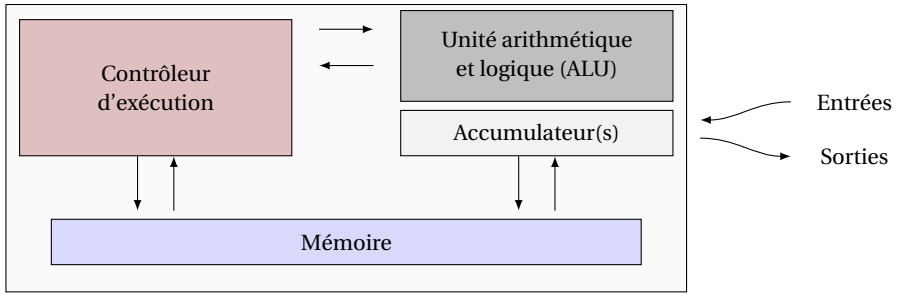
3 Programmation impérative

Durant les deux premiers chapitres, nous nous sommes intéressés à des exemples de programmation dite *fonctionnelle*, mettant en avant la définition et l'évaluation de fonctions. C'est le cœur des langages de la famille ML, donc Caml fait partie.

Toutefois, au contraire de langages fonctionnels purs comme Haskell, Caml (et les langages ML) ne sont pas limités à la seule programmetion fonctionnelle, et il est possible d'utiliser également une programmation impérative, ce qui sera le sujet de ce chapitre.

1 Programmation impérative

La programmation *impérative* repose sur l'idée que Von Neumann se faisait de l'architecture d'un ordinateur : une mémoire contenant les données et les instructions constituant le programme, les instructions étant exécutées tour à tour, sous la direction d'un *contrôleur d'exécution* par une unité de calcul, et ont pour effet des modifications du contenu de la mémoire [1]. On adjoint généralement à cette architecture des *entrées-sorties* qui permettent à l'ordinateur de communiquer avec l'extérieur.



Dans la suite de ce cours, nous introduirons les principales structures couramment rencontrées en programmation impérative (séquences d'instructions, boucles, boucles conditionnelles) ainsi que des objets *mutables* qui seront indispensable, nous le verrons, pour rendre compte du caractère « évolutif » du contenu de la mémoire dans ce type de programmation.

1. Modifications des données ou bien du programme lui-même !

2 Outils de programmation impérative

2.1 Séquences d'instructions

Dans une programmation de style impératif, les instructions se succèdent les unes aux autres. Il est donc nécessaire de pouvoir exécuter plusieurs instructions à la suite.

En Caml, les instructions sont séparées par des points-virgules (même si un retour à la ligne sépare les deux instructions).

```
# let foo x =
    print_string "Le carré de "; print_int x;
    print_string " est "; print_int (x * x);
    print_string ".";
    print_newline ();;

val foo : int -> unit = <fun>
```

On remarquera qu'il n'est pas besoin de ne mettre qu'une instruction par ligne, tant qu'elles sont séparées par des points virgules, même si l'on tend à le faire pour faciliter la lecture des fonctions [2].

Le résultat de la dernière expression est celui qui sera retourné par la fonction :

```
# let foo x =
    print_string "Calcul du carré de ";
    print_int x;
    print_newline ();
    x * x;;

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

Dans cette dernière fonction, on procède d'abord à un affichage, puis la fonction calcule et retourne le carré de son argument (entier).

```
# foo 2;;
Calcul du carré de 2
- : int = 4
```

Bien évidemment, comme une fonction ne retourne qu'un seul argument, chacune des instructions excepté la dernière devrait retourner l'élément `()` de type `unit` !

2. Il en est d'ailleurs de même en Python, il est théoriquement possible de mettre plusieurs instructions sur une même ligne en les séparant par des point-virgules, mais cette pratique est très forcement déconseillée.

Ce n'est pas rigoureusement requis (si ce n'est pas le cas, les calculs intermédiaires sont simplement perdus), quoique Caml fournira un avertissement :

```
# let foo x =
    x * x * x;
    x * x;;

Characters 18-27:
      x * x * x;
      ^^^^^^^^^
Warning 10: this expression should have type unit.
val foo : int -> int = <fun>
```

C'est un simple avertissement, la fonction est définie quand même, mais elle ne renvoie que le carré de l'argument (la première instruction, ici, ne sert à rien!)

Pour que les instructions, la dernière excepté, aient un intérêt, il faut impérativement qu'elles aient un effet (affichage à l'écran, modification de la mémoire...). C'est la raison pour laquelle on n'a guère eu besoin d'utiliser des séquences d'instructions jusqu'à présent [3].

2.2 Instructions conditionnelles

Une *instruction conditionnelle* est une instruction dont le comportement dépend d'une condition, généralement le résultat d'une expression booléenne. Elle s'écrit en Caml avec la structure

```
if condition then expression_1 else expression_2
```

Si `condition` est vraie, alors `expression_1` est évaluée et retournée, sinon c'est `expression_2` qui le sera.

On peut ainsi réécrire notre fonction `fact` calculant la factorielle d'un entier positif :

```
# let rec fact n =
    if n <= 1 then 1 else n * fact (n-1);;

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

Il n'y a pas de différence pratique entre la forme précédente de la fonction `fact` et celle que l'on a vu dans le premier chapitre utilisant un filtrage : selon la valeur de `n`, on retourne l'une ou l'autre des expressions (`1` ou `n * fact (n-1)`).

3. Excepté dans le précédent chapitre pour afficher les différents parcours d'un arbre, où l'on faisait précisément des affichages à l'écran.

En Caml, il est possible d'utiliser cette structure conditionnelle partout où on attend une expression, y compris à l'intérieur d'une expression. Par exemple, dans cette fonction :

```
# let foo n =
    n + (if n mod 2 = 1 then 1 else -1);;

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

Attention, une expression en Caml doit toujours renvoyer un objet de même type, aussi le **else** n'est, la très grande majorité du temps, pas facultatif :

```
# let foo n =
    if n mod 2 = 1 then n+1;;

Characters 42-45:
      if n mod 2 = 1 then n+1;;
                          ^^^
Error: This expression has type int
       but an expression was expected of type unit
```

Il n'existe qu'une exception à cette règle, une expression de type `unit`, auquel cas Caml ajoutera un « **else** `()` » automatiquement, ce qui permet donc d'écrire :

```
let foo n =
    if n mod 2 = 1 then print_int n;;

# val foo : int -> unit = <fun>
```

2.3 Blocs

Il n'est pas possible de mettre une séquence d'instructions entre le **then** et le **else** :

```
# let foo n =
    if n mod 2 = 1 then
        print_string "impair"; print_newline ()
    else
        print_string "pair"; rint_newline ();;

Characters 95-99:
      else
      ^^^^
Error: Syntax error
```

Il est nécessaire de « grouper » la séquence d'instructions dans un « bloc » qui se comportera comme une expression unique, grâce aux mots-clés **begin** et **end** :

```
# let foo n =
    if n mod 2 = 1 then
        begin
            print_string "impair"; print_newline ();
        end
    else
        begin
            print_string "pair"; print_newline ();
        end;;

val foo : int -> unit = <fun>
```

C'est également indispensable pour l'expression associée au **else** ici, car sinon le second `print_newline` ne ferait pas partie de la conséquence de l'échec de la condition `n mod 2 = 1`, mais serait exécuté quelle que soit la valeur de `n`, puisque le **else** ne prendra que l'expression qui le suit immédiatement.

Il est possible d'utiliser des parenthèses à la place de **begin** ... **end** dans la plupart des situations, même si en terme de lisibilité, le bloc apparaît de façon moins évidente :

```
# let foo n =
    if n mod 2 = 1 then
        (print_string "impair"; print_newline ())
    else
        (print_string "pair"; print_newline ())

val foo : int -> unit = <fun>
```

Ce type de bloc est également utile lorsque l'on a des filtrages imbriqués, afin de préciser à quel filtrage appartient chacun des motifs !

Considérons par exemple le cas suivant, où les `m1`...`m4` seraient des motifs :

```
match expr1 with
  | m1 -> match expr2 with
            | m2 -> ...
            | m3 -> ...
  | m4 -> ...
```

Contrairement à ce que l'on a pu vouloir écrire (ou que l'on peut comprendre en première lecture), le motif `m4` est un motif pour le *second* filtrage (on rappelle que Caml n'est pas sensible à l'indentation).

Si `m4` est bien un motif pour le *premier* filtrage, comme l'indentation le laisse supposer, il conviendrait d'écrire :

```
match expr1 with
  | m1 -> begin
            match expr2 with
              | m2 -> ...
              | m3 -> ...
          end
  | m4 -> ...
```

ou bien :

```
match expr1 with
  | m1 -> (match expr2 with
             | m2 -> ...
             | m3 -> ... )
  | m4 -> ...
```

2.4 Boucles inconditionnelles (`for`)

Pour effectuer un nombre déterminé de fois une série d'instructions, on écrira

```
for nom = expression_1 to expression_2 do sequence done
```

`expression_1` et `expression_2` doivent donner un résultat entier.

Le `nom` est alors associé successivement à tous les entiers entre `expression_1` et `expression_2` (inclus), et la séquence d'instructions `sequence` est évaluée pour chacun de ces entiers (notons que par la présence de **do** et **done**, il n'est pas besoin de définir nous-même un bloc d'instructions ici s'il faut plus d'une instruction dans la boucle.

Ainsi, l'expression

```
for i = 1 to 3 do expression done
```

est équivalente à la séquence d'instructions suivante :

```
let i = 1 in expression;
let i = 2 in expression;
let i = 3 in expression;
```

Précisons que, dans le cas où `expression_1` donne un résultat strictement supérieur à `expression_2`, il ne se passera rien.

On peut ainsi, par exemple, avec une boucle inconditionnelle, écrire une fonction qui imprime une table de multiplication [4] :

```
# let table n =
    for i = 1 to 10 do
      print_int n;
      print_string " fois ";
      print_int i;
      print_string " égale ";
      print_int (n*i);
      print_newline ();
    done;;

val table : int -> unit = <fun>
```

Les expressions dans la boucle devraient retourner un `()` de type `unit`, car le résultat de ces expressions sera « jeté » par Caml après chaque itération [5]. Ainsi, la fonction

```
# let foo n =
    for i = 1 to 10 do
      i * n;
    done;;

Characters 40-46:
      i * n;
      ^^^^^^
Warning 10: this expression should have type unit.
val foo : int -> unit = <fun>
```

est acceptée, mais ne renvoie rien, ce qui n'est probablement pas ce que l'on souhaite !

Cette fois encore, si l'expression dans la boucle n'a pas d'effet (affichage, modification de la mémoire...), cette structure de contrôle n'a guère d'intérêt.

Il n'existe pas de structure permettant de choisir le pas lors de l'itération, ou d'itérer sur autre chose que des entiers. On dispose cependant du mot-clé **downto** afin de *décompter* au lieu de compter :

```
for nom = expression_1 downto expression_2 do sequence done
```

4. Bien évidemment, rien n'empêche de le faire avec une écriture purement fonctionnelle et une récursion, nous y reviendrons. On dispose simplement ici d'une *autre* manière d'exprimer une telle opération.
5. Y compris la dernière, une boucle **for** retourne bien toujours `()` et non le résultat de l'expression de la dernière itération !

2.5 Boucles conditionnelles (`while`)

Parfois, le nombre d'itérations à effectuer n'est pas connu à l'avance, et l'on souhaite effectuer une tâche tant qu'une expression est vraie. Pour ce faire, on dispose de la structure de contrôle suivante :

```
while expression do sequence done
```

Avec cette structure, `sequence` sera évaluée autant de fois que nécessaire, tant que `expression` sera vraie. Par exemple :

```
while read_line () <> "Au revoir" do
    print_string "Dites m'en plus !";
    print_newline ();
done;;
```

Encore plus que dans les exemples précédents, on a besoin ici que quelque chose se passe pour que `expression` change après un certain nombre d'itérations, sinon on sera bloqués dans une boucle infinie ! Il devient vraiment indispensable que `sequence` puisse agir sur le contenu de la mémoire.

Il n'est en effet pas possible d'utiliser une définition **let** `... = ...` à l'intérieur de la boucle. Tout au plus peut on utiliser une définition locale **let** `... = ... in ...`, mais cette définition sera oubliée dès la fin de l'itération dans laquelle elle apparaît.

3 Références

3.1 Les références

On l'a vu, la programmation impérative n'a de sens que si l'on est capable d'agir sur le contenu de la mémoire, ce qui ne peut être fait avec des définitions. Plutôt que d'associer un nom à un objet (valeur, chaîne, arbre...), il est possible, grâce au mot-clé `ref`, de créer une *référence* vers un objet.

En écrivant par exemple

```
# let a = ref 2.2;;
val a : float ref = {contents = 2.2}
```

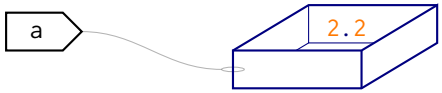
on demande à Caml d'associer le nom « a » à une référence vers un flottant (ainsi que l'indique le `float` du type « **float** `ref` »), ce flottant étant initialement 2.2.

Dans un premier temps [6], on peut imaginer le nom associé à une « boîte » contenant

6. Nous verrons un peu plus tard que cette image peut poser quelques soucis dans certains cas.

l'entier. D'ailleurs, la réponse de Caml suggère bien que l'on manipule une « boîte » avec un contenu. Les noms sont ainsi associés aux « boîtes » plutôt qu'à leur contenu.

La définition `let a = ref 2.2` conduit donc à une situation de la sorte, où l'on voit que le nom a est bien associé (de façon permanente) à la boîte et non au flottant qu'elle contient :



On ne peut utiliser directement une référence comme s'il s'agissait de l'objet qu'elle contient :

```
# a *. 2.0;;

Characters 2-3:
  a *. 2.0;;
  ^
Error: This expression has type float ref
       but an expression was expected of type float
```

Il faut donc préalablement extraire le flottant. Pour ce faire, on fait précéder le nom d'un point d'exclamation :

```
# !a;;
- : float = 2.2

# !a *. 2.0;;
- : float = 4.4
```

Pour modifier le contenu de la case mémoire, on utilise l'opérateur « `:=` » :

```
# a := 3.7;;
- : unit = ()

# !a;;
- : float = 3.7

# a;;
- : float ref = {contents = 3.7}
```

Le contenu de la case mémoire a bien été changé, mais pas la définition de a.

3.2 Utilisation

Les références permettent, entre autres choses, de créer des accumulateurs et des compteurs, des objets que l'on retrouve trèès fréquemment dans la programmation impérative. Il est par exemple très simple d'écrire, en style impératif, une fonction factorielle :

```
# let fact n =
    let res = ref 1 in
      for i=2 to n do
        res := !res * i
      done;
      !res;;

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

Pour incrémenter l'entier désigné par une référence x, il suffit en principe d'écrire

```
x := !x + 1
```

Toutefois, comme il s'agit d'une opération courante en programmation impérative, on dispose d'une fonction `incr` qui prend en argument une référence vers un entier et réalise la même opération d'incrémentation. « `incr x` » est donc équivalent à « `x := !x + 1` ». De la même façon, on dispose de la fonction `decr` qui décrémente d'une unité le contenu d'une référence vers un entier.

Il est possible de créer des références vers des objets de n'importe quel type. Ainsi, dans la fonction suivante, qui compte le nombre de zéros d'une liste d'entiers, on utilise ainsi deux références, l'une, `reste`, recueillant la liste des données restant à traiter, la seconde, `nombre`, le nombre de zéros déjà identifiés dans la liste :

```
let compte_zeros lst =
    let nombre = ref 0          (* un compteur de zéros          *)
    and reste = ref lst in      (* éléments restant à examiner   *)
      while !reste <> [] do
        if List.hd !reste = 0
          then incr nombre;     (* else () implicite             *)
        reste := List.tl !reste (* le premier élément est traité *)
      done;
      !nombre;;                 (* on retourne le contenu        *)
```

La fonction précédente illustre la façon dont on traite généralement les listes dans un style impératif. Rappelons que `List.tl` ne crée pas une copie de la liste, et est bien une opération en $O(1)$, donc cette fonction n'est pas inefficace.

De même, on peut très bien avoir des références de fonctions, par exemple ici des

fonctions des entiers vers les entiers :

```
# let funct = ref abs;;
val foo : (int -> int) ref = {contents = <fun>}

# !funct (-37);;
- : int = 37
```

On peut alors y associer tout objet de type[7] `int -> int` :

```
# funct := fun x -> x*x*x;;
- : unit = ()

# funct := min 0;;
- : unit = ()
```

Une référence peut même contenir une référence :

```
# let b = ref (ref 37);;
val b : int ref ref = {contents = {contents = 0}}
```

Pour accéder à l'entier, il faut alors utiliser deux fois un déréférencement :

```
# ! !b;;
- : int = 37
```

Les références sont en revanche toujours associées à un type bien particulier, et il n'est pas possible de leur associer un objet d'un autre type :

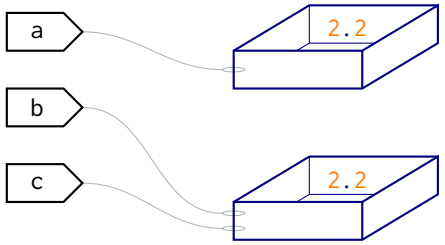
```
# let a = ref 2;;
val a : int ref = {contents = 2}

# a := 4.0;;

Characters 6-9:
  a := 4.0;;
       ^^^
Error: This expression has type float
       but an expression was expected of type int
```

7. En fait, il est même possible d'y associer des fonctions de signature `int -> 'a`, `'a -> 'a`, etc. Cependant, l'objet qui se retrouvera référencé sera un objet de type `int -> int`, qui le restera une fois déréférencé, et qui donc ne correspondra plus à l'objet « original ».

3.3 Égalité, identité

Deux noms peuvent aussi bien désigner la même « boîte », comme b et c dans l'exemple ci-dessous :

```
# let a = ref 2.2 and b = ref 2.2;;
val a : float ref = {contents = 2.2}
val b : float ref = {contents = 2.2}

# let c = b;;
val c : float ref = {contents = 2.2}
```



Modifier le contenu de la « boîte » désignée par b aura donc des conséquences sur c mais pas sur a :

```
# b := 3.7;;
- : unit = ()

# a;;
- : float ref = {contents = 2.2}

# c;;
- : float ref = {contents = 3.7}
```

Il est dès lors naturel de se poser, dans ce genre de situation, la question du fonctionnement de l'opérateur d'égalité[8] `=`. En Caml, celui-ci teste une *égalité de valeurs* (parfois qualifiée d'*égalité structurelle*), autrement dit l'opérateur regarde si les *contenus* sont égaux :

```
# a = b;;
- : bool = true

# b = c;;
- : bool = true
```

8. qui n'est pas sans rappeler des difficultés similaires concernant l'égalité/l'identité de deux listes en Python

Notons qu'il n'est, naturellement, possible que de comparer deux éléments de même type, et qu'une référence vers un flottant n'est pas comparable à un flottant :

```
# a = 3.7;;
Characters 6-9:
  a = 3.7;;
      ^^^
Error: This expression has type float
       but an expression was expected of type float ref
```

Pour tester l'*identité* (ou *égalité physique*), on utilisera l'opérateur `==` :

```
# a == b;;
- : bool = false

# b == c;;
- : bool = true
```

L'opérateur `!=` teste lui la « *non-identité* »[9].

```
# a != b;;
- : bool = true

# b != c;;
- : bool = false
```

Les opérateurs `==` et `!=` font référence à la manière dont les objets sont rangés en mémoire, et **leur usage est à réserver aux objets mutables**. Leur comportement sur des objets immutables peut être imprévisible (et varier d'un compilateur à l'autre) :

```
# 2.5 == 2.5;;
- : bool = false
```

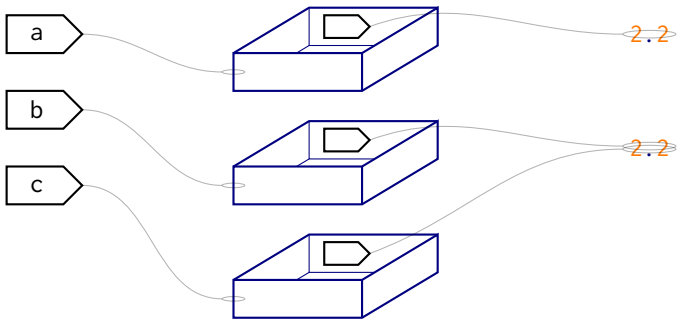
3.4 Une dernière remarque pour clore

Précisons pour terminer que si cette image de « boîte » est généralement suffisante, elle peut montrer ses limites. En effet, il est possible de créer des références distinctes vers un même objet (qui se trouverait alors simultanément dans deux « boîtes »!), comme ci-dessous :

```
# let a = ref 2.2 and b = ref 2.2
# let c = ref !b
```

9. On rappelle que c'est l'opérateur `<>` qui teste si deux valeurs ne sont pas égales.

Une meilleure image serait de considérer que l'on place dans chaque boîte non pas les objets eux-même, mais une étiquette permettant de les retrouver, comme illustré ci-dessous :



Toutefois, tant que les objets auxquels on fait référence sont immutables, et ce sera le cas dans la très grande majorité des situations, cette distinction n'a pas d'importance.

4 Objets Caml avec mutabilité

4.1 Cas du type « enregistrement »

Supposons que l'on souhaite manipuler un annuaire en Caml, regroupant le nom et le numéro de téléphone de différentes personnes.

On peut par exemple écrire un type « enregistrement » associant un nom et un numéro de téléphone [10] :

```
type coord = { nom: string ; numéro: string };;
```

On peut ensuite définir un annuaire comme une liste de tels éléments [11] :

```
let annuaire = [ { nom = "Dupont" ; numéro = "0123456789" } ;
                 { nom = "Durand" ; numéro = "0246813579" } ;
                 { nom = "Martin" ; numéro = "0918273645" } ];;
```

L'ennui, c'est qu'il n'est pas possible de modifier un numéro si la personne en change, sans modifier la liste de façon à retirer l'élément devenu incorrect pour le remplacer par un nouveau.

10. On aurait pu mémoriser le numéro sous la forme d'un entier, mais si la version de Caml utilise des entiers 32bits, ils ne permettront pas de mémoriser n'importe quel numéro à dix chiffres, et on perdrait les 0 de tête.
11. Nous verrons dans le chapitre suivant une meilleure solution pour définir un annuaire.

Ce que l'on pourrait écrire, dans un style fonctionnel, par [12] :

```
# let rec modifie nom nouv_numero = function
    | (t::q) when t.nom = nom
      -> { nom = nom ; numéro = nouv_numero }
           :: modifie nom nouv_numero q
    | (t::q) -> t :: modifie nom nouv_numero q
    | []     -> [];;

val modifie : string -> string -> coord list -> coord list = <fun>
```

Cela crée un *nouvel* annuaire, intégrant la correction. L'argument n'est lui pas modifié.

On peut donc par exemple utiliser la fonction `modifie` de la sorte :

```
# let nouvel_annuaire = modifie "Durand" "0000012345" annuaire ;;
val nouvel_annuaire : coord list =
  [{nom = "Dupont"; numéro = "0123456789"};
   {nom = "Durand"; numéro = "0000012345"};
   {nom = "Martin"; numéro = "0918273645"}]
```

On peut préférer *modifier* un annuaire existant, et pour ce faire utiliser des références pour le numéro, au prix d'un changement dans la déclaration de l'annuaire :

```
type coord = { nom: string ; numéro: string ref };;

let annuaire = [ { nom = "Dupont" ; numéro = ref "0123456789" } ;
                 { nom = "Durand" ; numéro = ref "0246813579" } ;
                 { nom = "Martin" ; numéro = ref "0918273645" } ];;
```

La fonction de modification peut alors s'écrire :

```
# let rec modifie nom nouv_numero = function
    | (t::q) when t.nom = nom
      -> t.numéro := nouv_numero;
         modifie nom nouv_numero q
    | (t::q) -> modifie nom nouv_numero q
    | [] -> ();;

val modifie : string -> string -> coord list -> unit = <fun>
```

12. Notons que si l'on trouve le nom recherché dans l'annuaire, on poursuit la recherche, et si le nom apparaît plusieurs fois dans l'annuaire, *tous* les numéros seront mis à jour.

Le résultat de la fonction est à présent de type `unit`, car on ne construit plus un nouvel annuaire, on se contente de modifier l'existant, en écrivant :

```
# modifie "Durand" "0000056789" annuaire;;
- : unit = ()

# annuaire;;
- : coord list =
[{nom = "Dupont"; numéro = {contents = "0123456789"}};
 {nom = "Durand"; numéro = {contents = "0000056789"}};
 {nom = "Martin"; numéro = {contents = "0918273645"}}]
```

On peut aussi adopter un style plus impératif :

```
# let modifie nom nouv_numero annuaire =
    let reste = ref annuaire in
      while !reste <> [] do
        let coord = List.hd !reste in
          if coord.nom = nom
            then coord.numero := nouv_numero;
          reste := List.tl !reste
      done;;

val modifie : string -> string -> coord list -> unit = <fun>
```

L'inconvénient de cette approche est que cela change la façon de déclarer l'annuaire (avec des `ref`) et de l'utiliser (avec des `!`), ce qui peut être parfois génant. Il existe cependant une autre façon de procéder : Caml nous offre la possibilité de déclarer un champ du type enregistrement comme étant *mutable* :

```
type coord = { nom: string ; mutable numéro: string };;

let annuaire = [ { nom = "Dupont" ; numéro = "0123456789" } ;
                 { nom = "Durand" ; numéro = "0246813579" } ;
                 { nom = "Martin" ; numéro = "0918273645" } ];;
```

On peut alors modifier l'élément mutable avec l'opérateur `<-` :

```
# (List.hd annuaire).numéro <- "9876543210";;
- : unit = ()

# List.hd annuaire;;
- : coord = {nom = "Dupont"; numéro = "9876543210"}
```

On écrit alors notre fonction de modification par exemple de la façon suivante :

```
# let rec modifie nom nouv_numero = function
    | (t::q) when t.nom = nom
      -> t.numéro <- nouv_numero;
         modifie nom nouv_numero q
    | (t::q) -> modifie nom nouv_numero q
    | [] -> ();;

val modifie : string -> string -> coord list -> unit = <fun>
```

Pour ceux qui se demanderaient pourquoi l'on a créé un opérateur supplémentaire `<-` au lieu d'utiliser `:=`, dans le cas où l'on définit un objet de la sorte

```
type foo = { mutable elem = int ref };;
```

il fallait bien pouvoir distinguer les deux opérations qui sont toutes les deux possibles ici !

4.2 Un autre objet mutable : les tableaux (`'a array`)

Certains types proposés par Caml sont « naturellement » mutables. Les chaînes de caractères l'ont été (il était possible de « muter » un caractère d'une chaîne), mais ne le sont plus [13] dans les dernières versions de OCaml.

Les listes sont immutables, ce qui rend, on l'a vu, leur utilisation dans un style impératif délicat. On dispse donc d'un autre conteneur, mutable, que l'on utilisera souvent dans un style impératif : les *tableaux* (de type `'a array`). Ce sont des objets qui peuvent contenir un nombre *prédéterminé* d'éléments *de même type*. On peut les définir explicitement en plaçant différents éléments (impérativement tous de même type), séparés par des points virgules, entre `[|` et `|]` :

```
# let tableau = [| 1.2; 2.3; 3.4 |];;
val tableau : float array = [|1.2; 2.3; 3.4|]
```

On peut également créer un tableau grâce à la fonction `Array.make`, en précisant la taille et l'élément à placer dans chaque case :

```
# Array.make 6 0.0;;
- : float array = [|0.; 0.; 0.; 0.; 0.; 0.|]

# Array.make 3 "Hello";;
- : string array = [|"Hello"; "Hello"; "Hello"|]
```

13. Il existe un type `bytes`, très semblable aux chaînes de caractères, qui lui est mutable.

On peut obtenir la taille d'un tableau avec `Array.length`, et accéder à un élément en indiquant, entre parenthèses précédées d'un point, l'indice de l'élément souhaité :

```
# Array.length tableau;;
- : int = 3

# tableau.(1);;
- : float = 2.3
```

Au contraire de ce qui se passe avec les listes, ces opérations sont toutes deux effectuées en temps constant ($O(1)$).

Par ailleurs, les tableaux étant des objets mutables, il est possible de modifier un élément du tableau avec `<-` :

```
# tableau.(1) <- 10.23;;
- : unit = ()

# tableau;;
- : float array = [|1.2; 10.23; 3.4|]
```

Il existe de nombreuses fonctions destinées à la manipulation de tableaux (dont `Array.copy`, `Array.sub`, `Array.iter`, `Array.map`, `Array.mem`, `Array.to_list`, `Array.of_list`, `Array.sort`...) dont la liste et le fonctionnement sont résumés dans la documentation du langage, et que nous découvrirons en fonction de nos besoins.

Les listes et les tableaux répondent à des besoins différents, comme nous le verrons. Il est très facile d'obtenir une liste résultant de l'ajout ou de la suppression d'un élément en tête d'une liste existante (en $O(1)$), mais le coût pour accéder à un élément au milieu de la liste est élevé (en $O(n)$). Par ailleurs, le contenu d'une liste est immutable. À l'inverse, les tableaux sont des objets mutables, mais ont une taille fixe (la changer nécessite de recopier le tableau, avec un coût en $O(n)$). Ainsi, en fonction des besoins de l'algorithme, on préférera donc l'une ou l'autre de ces structures.

4.3 Tableaux bidimensionnels

Pour représenter un tableau en deux dimensions, il n'existe pas de type particulier, mais comme on peut définir des tableaux de n'importe quel type, on peut définir des tableaux de tableaux. Attention toutefois, si l'on souhaite construire une matrice nulle de trois lignes et deux colonnes, **on ne peut écrire** :

```
# let matrice = Array.make 3 (Array.make 2 0.0);; (* incorrect *)
- : float array array = [|[|0.; 0.|]; [|0.; 0.|]; [|0.; 0.|]|]
```

Même si le résultat semble satisfaisant, on a créé ici un tableau *qui contient trois fois la même ligne*[14] ! Modifier un élément sur une ligne quelconque aurait un effet sur toutes les autres, ce qui n'est a priori pas ce que l'on cherche...

Pour définir une matrice nulle de trois lignes et deux colonnes, on pourra en revanche écrire :

```
# let matrice = Array.make 3 [| |];;
val matrice : '_a array array = [|[||]; [||]; [||]|]

# for i=0 to 2 do matrice.(i) <- Array.make 2 0.0 done;;
- : unit = ()

# matrice;;
- : float array array = [|[|0.; 0.|]; [|0.; 0.|]; [|0.; 0.|]|]
```

Caml fournit cependant fort obligeamment un raccourci pour effectuer cette construction, `Array.make_matrix` :

```
# Array.make_matrix 3 2 0.0;;
- : float array array = [|[|0.; 0.|]; [|0.; 0.|]; [|0.; 0.|]|]
```

On fait référence la ligne d'indice i par :

```
# matrice.(1);;
- : float array = [|0.; 0.|]
```

Et donc à l'élément situé sur la ligne d'indice i dans la colonne d'indice j par :

```
# matrice.(1).(0);;
- : float = 0.
```

Remarquons enfin que rien ne garantit, lorsque l'on a un `'a array array`, que chacune des « lignes » ait la même taille, et qu'il peut très bien ne pas s'agir d'un tableau bidimensionnel dans le sens usuel du terme !

14. Il s'agit exactement du même problème que lorsque l'on écrit `[ [0.0] * 2 ] * 3` en Python.

Exercices

Ex. 3.1 – Fonction mystérieuse

Déterminer ce que fait la fonction suivante :

```
let foo x y =
  x := !x + !y ;
  y := !x - !y ;
  x := !x - !y ;;
```

Ex. 3.2 – Parité d'un coefficient binomial

On peut montrer (théorème de Lucas) que le coefficient binomial $\binom{n}{k}$ est impair si et seulement si, à tout 0 dans l'écriture binaire de n correspond un 0 dans l'écriture de k (à la même position).

Proposer une fonction `estImpair` de signature `int -> int -> bool` prenant en argument les entiers n et k et retournant la parité de $\binom{n}{k}$.

Ex. 3.3 – Liste de couples

Proposer une fonction `couples` de signature `int -> (int * int) list` prenant en argument un entier $n > 0$ et retournant la liste de tous les couples d'entiers (x, y) vérifiant $1 \leqslant x \leqslant y \leqslant n$.

Ex. 3.4 – Permutations

Proposer une fonction `estPermutation` de signature `int array -> bool` indiquant si un tableau de longueur n, passé en argument, est une permutation de l'ensemble $[\![0\,..\,n-1]\!]$.

Ex. 3.5 – Indicatrice d'Euler

1. Proposer une fonction `pgcd` non-récursive de signature `int -> int -> int` déterminant le PGCD de deux entiers a et b.

2. En déduire une fonction non-récursive `phi` de signature `int -> int` correspondant à la fonction indicatrice d'Euler φ ($\phi(n)$ correspond au nombre d'entiers inférieurs ou égaux à n premiers avec n).

Ex. 3.6 – Codage

1. Proposer une fonction `char_cesar` de signature `char -> int -> char` prenant en argument un caractère et un entier n et retournant :

- le même caractère s'il ne s'agit pas d'une minuscule ;
- la minuscule encodée grâce au code de César dans le cas contraire, à savoir la minuscule se trouvant n rangs plus loin dans l'alphabet (en revenant au début si l'on dépasse la fin de l'alphabet) ; par exemple, pour $n = 4$, on décale les lettres de 4 rangs : `a` devient `e`, `b` devient `f`... et `z` devient `d`.

On rappelle que l'on dispose des fonctions `int_of_char` et `char_of_int` permettant de transformer un caractère en son code ASCII et inversement, et que les minuscules ont des codes ASCII consécutifs.

2. En déduire une fonction `cesar` de signature `string -> int -> string` prenant en argument une chaîne de caractères et un entier n et retournant la chaîne résultat de l'utilisation du code de César sur chacun de ses caractères.

Pour éviter d'utiliser de nombreuses concaténations de chaînes qui pourraient conduire à une complexité quadratique de la fonction, on pourra utiliser la fonction `String.concat` de signature `string -> string list -> string` qui prend en argument une chaîne et une liste de chaîne et retourne, en un temps linéaire vis-à-vis de la taille de la chaîne retournée, la concaténation de toutes les chaînes de la liste, en insérant entre chacune la chaîne passée en premier argument.

Plus simplement, `String.concat "" lst` retourne la concaténation de toutes les chaînes dans la liste `lst`.

3. Sur le même modèle, construire une fonction `vigenere` de signature `string -> string -> string` prenant en argument une chaîne de caractères et un seconde chaîne de caractères (ne contenant que des minuscules entre `a` et `z`) et encode les minuscules de la première chaîne de caractère grâce au code de Vigenere.

Le code de Vigenere est une variante du code de César, dans lequel le décalage est différent pour chaque caractère, et défini par la seconde chaîne (la *clé*) : si la clé contient p caractères, le i^{e} caractère de la première chaîne est encodé avec un décalage tel que le caractère `a` serait transformé en le $i \mod p^{\text{e}}$ caractère de la clé.

Par exemple, la chaîne `"hello"`, codée avec la clé `"abz"`, donnera la chaine `"hfklp"`.

4 Piles, files, dictionnaires

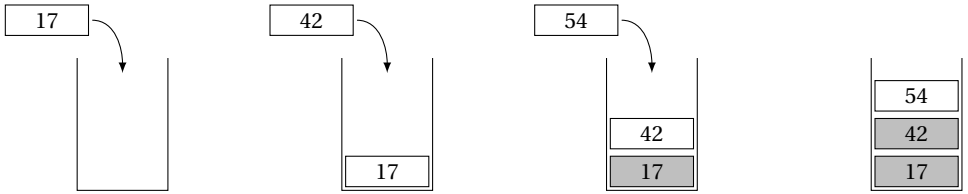
Dans ce quatrième chapitre, nous allons étudier quelques conteneurs très utiles pour l'écriture de nombreux algorithmes, les *piles*, les *files* et les *dictionnaires*, des structures de données mutables pouvant contenir un nombre variable de données (de même type), différant dans la façon dont les données sont introduites et extraites.

1 Les piles

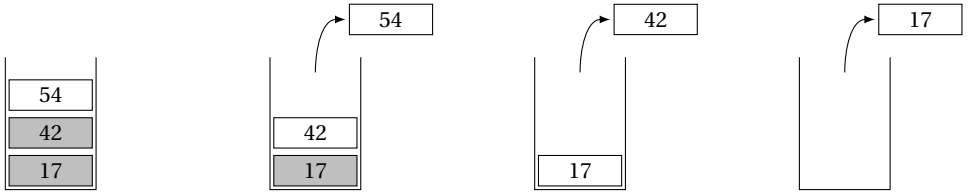
1.1 Principe

Imaginons un instant une pile de livres posée sur une table. Il est possible d'ajouter un livre à la pile en le plaçant au-dessus de la pile, ou bien de retirer le livre au sommet de la pile. Le seul livre qu'il soit possible de consulter est celui qui se trouve au sommet de la pile. Une pile, en informatique, reprend très exactement ce principe : c'est un conteneur qui regroupe un ensemble d'éléments mais ne permet d'effectuer que certaines opérations.

Tout d'abord, on peut ajouter un élément au sommet de la pile. Cette opération d'empilement est généralement appelée « *push* ».



On peut, de la même façon, reprendre l'élément au sommet de la pile. Le terme généralement associé à cette opération de dépilage est « *pop* ».



Il est fréquent qu'il soit permi de consulter l'objet situé au sommet de la pile sans avoir besoin de le retirer. En revanche, il n'est pas rare qu'il ne soit pas possible d'accéder aux éléments situés en-dessous dans la pile sans retirer chacun des éléments qui les recouvrent. Sauf mention contraire, lorsque l'on parlera de piles, on supposera que seul l'élément au sommet est directement accessible.

Pour pouvoir manipuler des piles, il nous faut également disposer d'une fonction permettant de créer une pile vide, ainsi qu'un moyen de tester si une pile est vide ou non.

1.2 Le module `Stack`

Le module OCaml « `stack` » fournit de quoi manipuler des piles. Il propose notamment trois fonctions pour les manipuler :

- `Stack.create` (`unit -> 'a Stack.t`), qui crée une nouvelle pile ;
- `Stack.push` (`'a -> 'a Stack.t -> unit`), qui ajoute un élément au sommet de la pile ;
- `Stack.pop` (`'a Stack.t -> 'a`), qui extrait et retourne l'élément situé au sommet de la pile ;
- `Stack.top` (`'a Stack.t -> 'a`), qui fournit l'élément situé en haut de la pile *sans le retirer*[1] ;
- `Stack.is_empty` (`'a Stack.t -> bool`), qui indique si la pile est vide.

On notera qu'une pile est un objet de type `'a Stack.t`. Comme pour les listes, les piles ne peuvent contenir que des objets de même type, même si ce type peut être un quelconque type que Caml puisse manipuler.

Le fonctionnement est donc très simple. On commence par créer la pile :

```
# let pile = Stack.create ();;
val pile : '_a Stack.t = <abstr>
```

Initialement, on ne connaît pas le type d'objets que contiendra la pile, d'où le `'_a`. On peut ensuite placer des objets dans la pile :

```
# Stack.push 17 pile;;
- : unit = ()

# Stack.push 42 pile;;
- : unit = ()

# Stack.push 54 pile;;
- : unit = ()
```

1. Il ne s'agit pas d'une copie de l'objet, ce que retourne la fonction désigne réellement l'objet en haut de la pile, et toute opération affectant l'objet retiré affecte également l'objet en haut de la pile.

L'ajout d'entiers dans la pile a eu pour conséquence de fixer le contenu des objets pouvant se trouver dans la pile : ce sera des entiers, comme en témoigne dorénavant le type de `pile`.

```
# pile;;
- : int Stack.t = <abstr>
```

Une tentative d'insertion d'autre chose qu'un entier va échouer :

```
# Stack.push 10.23 pile;;
Characters 19-23:
  Stack.push 10.23 pile;;
                   ^^^^
Error: This expression has type int Stack.t
       but an expression was expected of type float Stack.t
```

On peut ensuite retirer les objets placés dans la pile avec `Stack.pop` :

```
# Stack.pop pile;;
- : int = 54

# Stack.pop pile;;
- : int = 42
```

La fonction `Stack.top` permet également d'obtenir l'élément au sommet de la pile, mais sans le retirer :

```
# Stack.top pile;;
- : int = 17

# Stack.pop pile;;
- : int = 17
```

En fait, on aurait pu écrire la fonction `Stack.top` simplement avec les fonctions `Stack.pop` et `Stack.push`. Il suffit en effet de retirer l'élément au sommet de la pile, puis de le remettre avant de le retourner :

```
# let top pile =
    let elem = Stack.pop pile
      in Stack.push elem pile; elem;;

val top : 'a Stack.t -> 'a = <fun>
```

Cette fonction retournera naturellement une erreur si la pile est vide (comme le ferait également la fonction `Stack.top`).

La fonction `Stack.is_empty` enfin permet de savoir si une pile est vide ou non :

```
# Stack.is_empty pile;;
- : bool = true

# Stack.push 17 pile;;
- : unit = ()

# Stack.is_empty pile;;
- : bool = false
```

1.3 Les exceptions

Bien évidemment, lorsque la pile est vide, une tentative de retirer un élément provoque une erreur :

```
# Stack.pop pile;;
Exception: Stack.Empty.
```

En fait, il s'agit d'une *exception*. Dans beaucoup de langages, une exception est une information indiquant que quelque chose s'est mal passé. Si l'on ne fait rien, le programme va s'arrêter. Mais il est possible d'agir lorsque le programme rencontre une exception, et d'essayer de résoudre le problème (on parle de *rattraper* l'exception). Pour ce faire, on dispose d'une construction **`try ... with ...`**

Si la série d'instruction entre les mots-clés **`try`** et **`with`** déclenche une exception, celle-ci est comparée avec les exceptions pour lesquelles on dispose d'une solution, indiquées après le **`with`**. La syntaxe est très similaire à celle d'un filtrage par motif.

Ainsi, par exemple, pour savoir si une pile est vide, plutôt que d'utiliser `Stack.is_empty`, on peut procéder différemment : on retire l'élément en haut de la pile et on le remet. Si tout se passe bien, la pile n'était pas vide. Si en revanche l'exception `Stack.Empty` survient (lors de l'appel à `Stack.pop`), c'est que la pile était vide !

Ainsi, on peut écrire une fonction `is_empty` de la sorte :

```
# let isEmpty pile =
    try
      let elem = Stack.pop pile in Stack.push elem pile; false
    with
      | Stack.Empty -> true;;

val isEmpty : 'a Stack.t -> bool = <fun>
```

Ce qui suit le mot clé **with** est donc un filtrage de l'exception : ici, on ne traite que le cas de l'exception `Stack.Empty` qui, si elle est levée entre le **try** et le **with**, fait retourner à la fonction `true` au lieu du `false` retourné si les appels à pop puis push se déroulent sans problème (on remarquera ici que l'on n'attache pas d'importance à l'élément retiré, on veut seulement savoir si tout se passe bien si l'on y fait appel).

Si l'exception est identifiée par le filtrage, on dit qu'elle est *rattrapée*, et elle ne provoquera pas l'arrêt du programme. Après l'exécution de l'instruction ou de la séquence d'instruction spécifiée dans le **with**, *l'exécution se poursuit après le* **try** `...` **with**. Ce qui suit l'endroit du programme qui a déclenché l'exception n'est donc pas exécuté.

Une exception rompt donc le cours normal de l'exécution d'un programme, ce qui peut être à la fois un danger et une opportunité, lorsque c'est utilisé à bon escient.

`Stack.Empty` est une exception définie dans le module `Stack`, et levée lorsque l'on tente d'accéder à un élément d'une pile vide, mais il en existe bien d'autres, parmi lesquelles `Out_of_memory`, `Divison_by_zero`, `Match_failure`...

Ces exceptions sont levées lorsque l'on exécute une opération illégale. Mais on peut aussi lever soi-même une exception dans un programme, il suffit d'utiliser le mot-clé **raise** suivi du nom de l'exception à lever[2].

On peut définir ses propres exceptions (identifiées par un nom débutant par une majuscule) de la même façon que l'on peut définir ses types, en écrivant

```
# exception MonException;;
```

Ainsi, s'il n'existe pas de break dans une boucle **for** en Caml, on peut néanmoins avoir un comportement similaire grâce aux exceptions, comme dans cette fonction, qui retourne un booléen indiquant si un élément se trouve dans un tableau :

```
# exception Found;;

# let contient elem tab =
    try
      for i = 0 to Array.length tab - 1 do
        if tab.(i) = elem then raise Found
      done;
      false    (* Retourne false à l'issue de la boucle *)
    with
      | Found -> true;;  (* retourne true si on trouve *)

val contient : 'a -> 'a array -> bool = <fun>
```

2. L'utilisation des exceptions n'est en principe pas une compétence exigible pour les concours, aussi peut-on, en particulier en première lecture, ignorer la suite de cette section et passer directement à la suivante (interfaces de programmation).

Une exception peut par ailleurs « transporter » un élément, par exemple un entier, en la déclarant de la sorte :

```
# exception MonException of int;;
```

On écrira par exemple « `MonException 42` » pour obtenir une exception `MonException` associée à l'entier 42. L'élément associé à l'exception peut apporter des informations supplémentaires sur la situation qui a causé l'exception. C'est par exemple le cas de l'exception `Invalid_argument` que l'on rencontre souvent en Caml : elle est accompagnée d'une chaîne de caractères qui en dit davantage sur l'erreur. Ainsi, la chaîne sera par exemple `"index out of bounds"` si l'on tente d'accéder à un emplacement invalide dans une chaîne ou dans un tableau (index négatif ou trop grand).

Les éléments associés à une exception peuvent être récupérés par le filtrage. Par exemple, on peut modifier la fonction précédente pour qu'elle retourne la position de l'élément (et `-1` s'il n'est pas présent, puisque le type retourné doit toujours être le même).

```
# exception Position of int;;

# let position elem tab =
    try
      for i = 0 to Array.length tab - 1 do
        if tab.(i) = elem then raise (Position i);
      done;
      -1          (* L'élément n'est pas présent dans tab *)
    with
      | Position k -> k;;         (* Retourne la position *)

val position : 'a -> 'a array -> int = <fun>
```

Notons que ces exemples sont proposés à titre d'illustration, ce n'est pas nécessairement la façon la plus lisible de procédeer dans un tel cas, mais c'est une approche qui est régulièrement utilisée par les développeurs. Le tout est de s'en servir à bon escient (et en commentant la démarche).

Nous avons en fait déjà levé volontairement de telles exceptions grâce à la fonction `failwith` qui prend en argument une chaîne et provoque la levée d'une exception `Failure` qui contient la chaîne[3]. Si une exception a été levée par « `failwith "Message"` », elle peut donc être rattrapée par un filtrage « `Failure "Message" ->` » suivant une structure **try** `...` **with**.

Nous aurons l'occasion, dans les prochains cours, de croiser quelques autres utilisations de ce mécanisme de levée et de rattrapage d'exceptions.

3. `failwith "Hello"` a donc le même résultat que **raise** `(Failure "Hello")`.

1.4 Interfaces de programmation

Le module `Stack` nous fournit les fonctions, `Stack.create`, `Stack.push`, `Stack.pop`, `Stack.top` et `Stack.is_empty`, mais ne nous éclaire pas sur la façon dont ces fonctions sont implémentées.

C'est ce que l'on appelle une *interface*. L'utilisateur du module peut librement utiliser ces fonctions, dont le comportement est complètement détaillé (arguments attendus, résultats, effets).

En général, on précise également la complexité des fonctions. Dans le cas d'une pile, toutes les options présentées précédemment ont un coût constant (en O(1)).

L'utilisateur n'a en revanche pas à savoir comment ces fonctions sont programmées.

C'est une stratégie fréquemment utilisée en informatique. Pour le développeur du module, cela présente l'avantage de la flexibilité : il peut librement choisir la façon dont les fonctions sont implémentées, et peut même changer la façon dont les fonctions sont implémentés sans que les programmes utilisant le module ne soient impactés par le changement.

1.5 Implémentation

Il existe de nombreuses façons d'implémenter une structure de pile, nous allons en examiner quelques-unes.

La plus simple consiste à utiliser les nombreuses similarités entre les listes et les piles. On définit donc une pile comme un objet mutable contenant une liste (qui recueillera les éléments contenus dans la pile). On peut donc créer les trois fonctions élémentaires, `create`, `push` et `pop`, de la façon suivante [4] :

```
# type 'a t = { mutable contenu : 'a list };;

# let create () = { contenu = [] };;

# let push elem pile = pile.contenu <- elem :: pile.contenu;;

# exception Empty;;

# let pop pile =
    match pile.contenu with
      | t :: q -> pile.contenu <- q; t
      | []     -> raise Empty;;
```

4. On ne s'encombre pas ici avec `top` et `is_empty` pour lesquelles nous avons vu qu'il était possible de les définir à partir de `push` et `pop`.

Plutôt qu'un contenu mutable, on pourrait tout aussi bien définir le contenu comme une référence vers une liste :

```
# type 'a t = { contenu : 'a list ref };;

# let create () = { contenu = ref [] };;

# let push elem pile = pile.contenu := elem :: !(pile.contenu);;

# exception Empty;;

# let pop pile =
    match !(pile.contenu) with
      | t :: q -> pile.contenu := q; t
      | []     -> raise Empty;;
```

Puisque les ajouts en tête de liste, l'extraction de la tête et de la queue d'une liste, et la mutation sont des opérations qui sont réalisées en temps constant, chacune de ces trois fonctions a une complexité en O(1), c'est-à-dire qu'elles ne dépendent pas du nombre d'éléments dans la pile.

Ces deux approches sont très similaires, y compris en terme de rapidité à l'exécution. Il s'agit donc essentiellement ici de préférences de style de programmation de la personne écrivant le code.

Si une liste Caml est une structure de données particulièrement adaptée pour créer une structure de pile, car leurs fonctionnements sont similaires, ce n'est pas la seule solution que l'on puisse envisager.

On pourrait, par exemple, utiliser un tableau (`'a Array`) pour contenir les données [5]. La difficulté est qu'un tableau a une taille fixe, et il est hors de question, pour d'évidentes raisons d'efficacité, de créer un nouveau tableau à chaque ajout ou chaque extraction.

L'idée consiste donc à créer un tableau d'une certaine taille [6], et de mémoriser le nombre d'éléments présents dans la pile, car le tableau ne nous renseigne pas sur ce point. On supposera les éléments de la pile « tassés » dans sur la « gauche » du tableau (c'est-à-dire que l'élément d'indice 0 correspondra à l'élément le plus profond de la pile).

Il sera nécessaire de prévoir un remplacement du tableau par un tableau plus grand

5. Comme on le verra, cette approche va conduire à une solution plus complexe en terme d'implémentation, des concessions (certes modérées) en terme d'efficacité et un léger gaspillage de mémoire. On est en droit de se demander si cela présente un intérêt. En fait, les listes « dispersent » leurs éléments partout dans la mémoire au lieu de les garder dans une même zone, comme pour un tableau, ce qui a des conséquences sur la gestion de la mémoire, voire de performances (liées à l'utilisation du cache). On n'entrera pas dans les détails, mais cette solution utilisant un tableau présente des avantages qui ne semblent pas évident au premier abord.

6. Lors de l'ajout du premier élément, car avant cet ajout, on ne sait pas de quels types les objets seront.

lorsque celui-ci n'est plus assez grand pour contenir les données[7], ce qui complique quelque peu l'écriture de la fonction push.

```
# type 'a t = { mutable taille : int; mutable contenu : 'a array };;

# let create () = { taille = 0; contenu = [| |] };;

# let push elem pile =
    match pile.taille with
      | _ when Array.length pile.contenu = 0 ->
            (* La première insertion crée le tableau *)
            pile.contenu <- Array.make 4 elem;
            pile.taille <- 1
      | n when n < Array.length pile.contenu ->
            (* S'il reste de la place, on ajoute l'élément *)
            pile.contenu.(n) <- elem;
            pile.taille <- n+1
      | n -> (* S'il n'y a plus de place, on crée un nouveau tableau
                et on y recopie le contenu de l'ancien *)
            let n_contenu = Array.make (2*n) elem
              in for i = 0 to n-1 do
                n_contenu.(i) <- pile.contenu.(i)
              done;
              pile.contenu <- n_contenu;  (* n_contenu.(n)        *)
              pile.taille <- n+1;;        (* contient déjà elem ! *)

# exception Empty;;

# let pop pile =
    if pile.taille = 0 then raise Empty;
        pile.taille <- pile.taille - 1;
        pile.contenu.(pile.taille);;
```

La taille du tableau est ici doublée à chaque agrandissement, pour conserver, *en moyenne*, une complexité en O(1) lors de l'ajout d'un élément[8].

7. On pourrait également envisager de le remplacer par un tableau plus petit s'il contient beaucoup moins de données que sa capacité, afin d'économiser de la mémoire, mais on ne se souciera pas de ce problème ici.

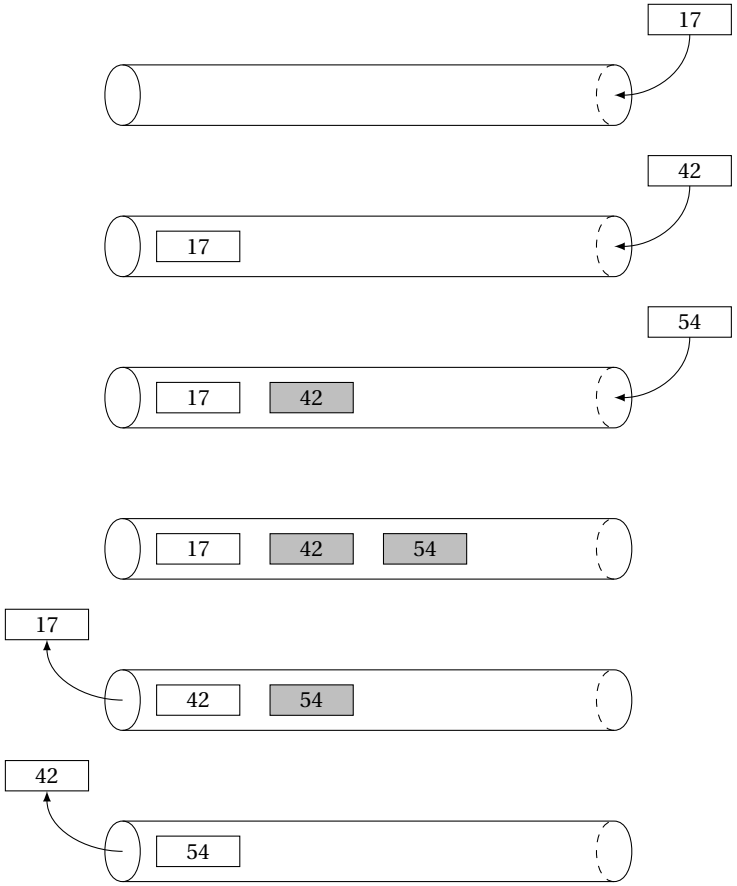
8. Une fois de temps en temps, on aura effectivement à recopier toutes les données, donc un ajout en $O(k)$, k étant la taille du tableau. Mais si on a, lors d'un ajout, à recopier k éléments, on n'aura pas à agrandir le tableau durant les $k-1$ ajouts suivants. Sans entrer trop dans les détails, le temps moyen d'un ajout est donc $(O(k)+(k-1)O(1))/k = O(1)$.

2 Les files

2.1 Principe

Une pile est un conteneur qualifié de *LIFO* (pour *Last In, First Out*), ou bien *dernier entré, premier sorti*, car les éléments sont extraits dans l'ordre inverse de leur insertion.

On peut définir un conteneur *FIFO* (pour *pour First In, First Out*) dans lequel les objets ressortent du conteneur dans l'ordre dans lequel ils ont été introduits. De tels conteneurs sont appelés *files*, ou parfois *queues*. On peut les assimiler à un tuyau, dans lequel on introduit les éléments à une extrémité et on les extrait à l'extrémité opposée.



Comme dans une pile, on ignore le nombre d'éléments présents dans la file, et on ne peut regarder qu'un seul élément, celui qui sera le prochain à sortir.

2.2 Le module Queue

OCaml fournit un module Queue qui contient quelques fonctions permettant de manipuler une file. Ces fonctions sont [9] :

- Queue.create (unit -> 'a Queue.t), qui crée une nouvelle file ;
- Queue.add ('a -> 'a Queue.t -> unit), qui insère un élément dans une file ;
- Queue.take ('a Queue.t -> 'a), qui extrait un élément de la file (ou lève l'exception Empty lorsque la file est vide) ;
- Queue.peek ('a Queue.t -> 'a), qui fournit, sans le retirer, le prochain élément à sortir de la file (ou lève l'exception Empty lorsque la file est vide) ;
- Queue.is_empty ('a Queue.t -> bool), qui indique si la file est vide ou non.

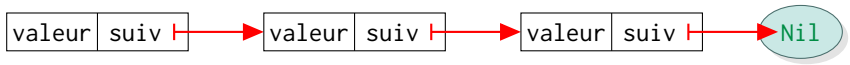
Il n'est pas possible ici de créer une fonction peek à partir de add et take : si on retire un élément de la file, il est impossible de le remettre où on l'a pris sans extraire et remettre tous les autres éléments ! En revanche, il est toujours possible de créer une fonction is_empty si elle n'existait pas, grâce aux exceptions :

```
# let isEmpty queue =
    try
      let elem = peek queue in false
    with
      | Empty -> true;;

val isEmpty : 'a Queue.t -> bool = <fun>
```

2.3 Implémentation

Implémenter une file est un peu plus difficile que d'implémenter une pile. Il n'est plus possible d'utiliser directement les listes OCaml car il faut pouvoir manipuler les deux extrémités. On peut en revanche ressortir le type « liste chaînée » que l'on avait élaboré précédemment (dans la représentation ci-dessous, le suivant d'un élément se situe à sa gauche, pour correspondre à la l'illustration précédente des files) :

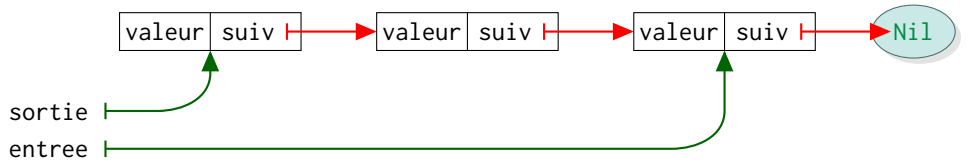
```
# type 'a cell = { valeur : 'a ; mutable suiv : 'a lst }
  and 'a lst = Nil | Cell of 'a cell;;
```



9. En fait, Queue.push, Queue.pop, Queue.top existent également, avec des comportements identiques à Queue.add, Queue.take et Queue.peek. Il en existe également quelques autres fonctions dans le module Queue sur lesquelles nous ne nous étendrons pas, qui s'écartent un peu de la structure théorique des files mais permettent de simplifier l'écriture de certains algorithmes.

Une file peut alors être représentée par une telle liste chaînée, dont on mémorise les deux extrémités, représentant l'entrée et la sortie de la file :

```
# type 'a t = { mutable entree : 'a lst ; mutable sortie : 'a lst };;
```



Une file vide est représentée par une liste chaînée vide, les deux extrémités pointant alors vers une étiquette « Nil ») :

```
# let create () = { entree = Nil ; sortie = Nil };;

val create : unit -> 'a t = <fun>
```

Insérer un élément dans la file consiste à ajouter un nouvel élément à droite de la liste chaînée.

Si la file est vide (entree et sortie désignent tous deux Nil), suite à l'ajout de l'élément, entree et sortie désignent le seul élément de la file, tout juste inséré.

Si la file n'est pas vide, l'ajout dans la liste s'effectue après l'élément désigné par le champ entree de notre file, qui n'avait pour l'instant pas de de suivant. Cet élément doit être modifié de sorte que son champ suiv désigne dorénavant l'élément nouvellement introduit :

```
# let add elem queue =
    let c = Cell { valeur = elem ; suiv = Nil }
      in match queue.entree with
        | Nil    -> queue.entree <- c;
                    queue.sortie <- c
        | Cell d -> queue.entree <- c;
                    d.suiv <- c;;

val add : 'a -> 'a t -> unit = <fun>
```

Retirer un élément dans la file consiste à retirer élément à gauche de la liste chaînée. S'il n'y avait aucun élément (la sortie désignant `Nil`), il nous faut lever une exception `Empty`. Dans le cas contraire, le champ `Sortie` de notre file devra pointer vers l'élément suivant celui qui vient d'être extrait (qui peut être `Nil`).

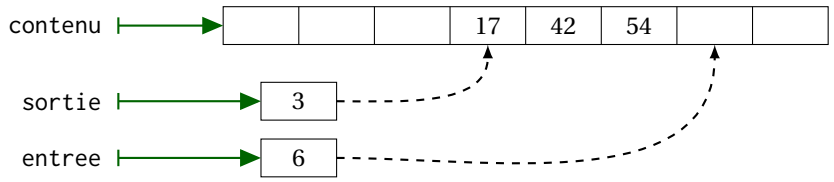
```
# exception Empty;;

# let take queue = match queue.sortie with
    | Nil    -> raise Empty
    | Cell c -> queue.sortie <- c.suiv;
                if c.suiv = Nil then queue.entree <- Nil;
                c.valeur;;

val take : 'a t -> 'a = <fun>
```

Enfin, le prochain élément à sortir est celui à gauche de la liste chaînée :

```
# let peek queue = match queue.sortie with
    | Nil    -> raise Empty
    | Cell c -> c.valeur;;

val peek : 'a t -> 'a = <fun>
```

Une autre solution utilise un tableau [10] [11] (`'a array`), ainsi que deux indices : celui de la case où ranger le prochain élément introduit, et celui de la case du prochain élément à sortir. Le contenu de la file est situé entre ces deux cases.

```
# type 'a t = { mutable contenu : 'a array ;
                mutable entree : int ;
                mutable sortie : int };;
```



10. Mutable, car il sera nécessaire de faire varier sa taille en fonction des besoins, en particulier l'agrandir s'il n'est plus assez grand.
11. Cette fois encore, l'implémentation est plus complexe, et il semble n'y avoir que des inconvénients, mais une étude plus poussée permettrait de voir que cette alternative présente, comme pour les piles implémentées avec des tableaux, des avantages en terme d'utilisation de la mémoire.

Initialement, puisque l'on ne connaît pas le type des éléments qui seront introduit dans la file, on est forcé, comme pour les listes, de créer un tableau vide [12].

```
# let create () = { contenu = [| |] ; entree = 0; sortie = 0 };;

val create : unit -> 'a t = <fun>
```

L'ajout se fait en plaçant l'élément dans la case désignée par l'indice `entree`. Lorsque l'on dépasse la fin du tableau, on reprend au début [13]. Si c'est le tout premier élément introduit dans la file (le tableau a encore une taille nulle), on en profite pour créer le tableau. Si, après l'insertion, le tableau est plein (ce que l'on détecte lorsque `entree` désigne la même case que `sortie`), on crée un tableau plus grand dans lequel on recopie les données [14] :

```
# let add elem queue =
    let taille = Array.length queue.contenu in
      if taille = 0 then   (* Insertion du premier élément,       *)
        begin              (* on crée un premier tableau non vide *)
          queue.contenu <- Array.make 4 elem;
          queue.entree <- 1
        end
      else
        begin
          queue.contenu.(queue.entree) <- elem;
          queue.entree <- (queue.entree + 1) mod taille;
          if queue.sortie = queue.entree then (* Si le tableau est    *)
            begin                             (* plein, on crée un    *)
              let n_tab = Array.make          (* nouveau tableau deux *)
                          (taille*2) elem in  (* fois plus grand dans *)
                for i = 0 to taille-1 do      (* lequel on va         *)
                  n_tab.(i) <-                (* recopier les données *)
                    queue.contenu.((queue.sortie + i) mod taille)
                done;
              queue.contenu <- n_tab;
              queue.entree <- taille;
              queue.sortie <- 0
            end
        end;;

val add : 'a -> 'a t -> unit = <fun>
```

12. Les valeurs de `entree` et `sortie` n'ont ici pas d'importance.
13. On parle parfois de tableau *circulaire*, comme si l'on avait collé l'extrémité droite à l'extrémité gauche.
14. Les données sont recopiées à partir du début du nouveau tableau, par simplicité.

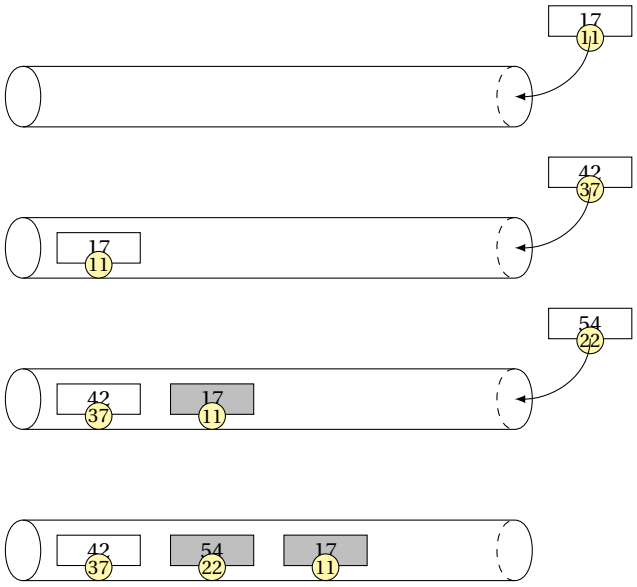
L'extraction des éléments de la file est bien plus simple :

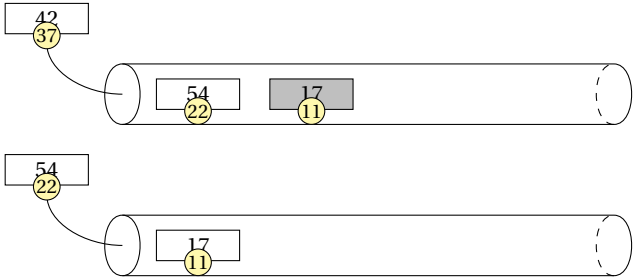
```
# exception Empty;;

# let take queue =
    if queue.entree = queue.sortie then raise Empty;
    let res = queue.qontenu.(queue.sortie) in
      queue.sortie <- (queue.sortie+1) mod (Array.length queue.contenu);
      res;;

val take : 'a t -> 'a = <fun>
```

3 Les files de priorité

3.1 Principe

Les files de priorité sont des conteneurs similaires aux files « normales » à ceci près qu'à chaque élément est associé une *priorité*, c'est-à-dire une valeur appartenant à un ensemble ordonné (généralement des entiers). L'ordre d'extraction des éléments ne dépend alors plus de l'ordre d'insertion, mais de la priorité des éléments : c'est l'élément de plus haute priorité qui est extrait en premier. Cela donne par exemple :





Les files de priorités permettent souvent d'altérer la priorité d'un élément dans la file, ce qui provoque une réorganisation des éléments dans la file (ils restent en permanence triés par ordre de priorité décroissante).

3.2 Implémentation

Il n'existe pas d'outil directement utilisable en OCaml pour manipuler une file de priorité [15]. Par ailleurs, l'implémentation d'une file de priorité est plus complexe. On en propose ici une implémentation élémentaire, à titre d'illustration, mais une solution plus efficace sera étudiée en seconde année.

Les fonctions à implémenter sont, comme pour une file, `create`, `add`, `peek` et `take` (`is_empty` pouvant, comme précédemment, être écrit à partir de `peek`). On choisit ici de représenter la file par une liste de couples, contenant un entier (la priorité) et un élément de type `'a` (le type des éléments dans la file).

```
# type 'a t = { mutable contenu : (int * 'a) list };;

# let create () = { contenu = [] };;

val create : unit -> 'a t = <fun>
```

Le début de la liste correspondra à la *sortie* de la file de priorité. Cela permettra d'accéder au prochain élément à sortir en $O(1)$:

```
# let peek queue =
    match queue.contenu with
      | (_,elem) :: q -> elem
      | [] -> raise Empty;;

val peek : 'a t -> 'a = <fun>
```

15. On trouve cependant de nombreuses bibliothèques prêtes à l'usage proposant de tels conteneurs.

Même chose pour l'extraire :

```
# let take queue =
    match queue.contenu with
      | (_,elem) :: q -> queue.contenu <- q; elem
      | [] -> raise Empty;;

val take : 'a t -> 'a = <fun>
```

L'ajout est plus délicat, car il faut placer l'élément à la bonne position dans la liste. Pour ce faire, on utilise une fonction auxiliaire `insert` qui va aller insérer dans la file le couple (priorité, élément) au bon endroit :

```
# let add priority elem queue =
    let rec insert = function
      | [] -> [ (priority,elem) ]
      | t::q when priority >= fst t (* Placé en tête si la priorité  *)
        -> (priority,elem)::t::q    (* est plus grande que l'élément *)
                                    (* présentement en tête de liste *)
      | t::q -> t::(insert q)       (* Sinon, on essaie plus loin    *)
    in queue.contenu <- insert queue.contenu;;

val add : int -> 'a -> 'a t -> unit = <fun>
```

On remarque ici que l'ajout est en $O(n)$, car si l'élément doit se retrouver à l'extrémité droite de la file, il est nécessaire de parcourir toute la liste pour le placer.

Nous verrons l'an prochain une meilleure solution pour l'implémentation, permettant d'avoir toutes les opérations en temps logarithmique ($O(\log(n))$).

4 Quelques exemples d'utilisation

4.1 Vérification du parenthésage

Supposons que l'on ait une chaîne contenant expression incluant des parenthèses, des crochets et des accolades. On souhaite vérifier que ces éléments soient bien équilibrés, c'est-à-dire que l'on trouve bien une parenthèse fermante à la suite d'une parenthèse ouvrante, et ainsi de suite [16].

Par exemple, `"a+[b-c*{(d/e+f)-[(g+h(i))]-k/{l+(m/n)}]+o(p)-q]+r}/s]"`

16. Ces règles sont à la base des langages de Dyck qui seront étudiés en seconde année.

On cherche donc à écrire une fonction de signature `string -> bool` ignorant tous les caractères qui ne sont pas des délimiteurs, et vérifiant le bon équilibrage de ces derniers. On souhaite par ailleurs avoir une complexité *linéaire* vis-à-vis de la longueur de la chaîne de caractères.

Les piles simplifient fort heureusement grandement ce type de vérification :

```
# let verifie ch =









val verifie : string -> bool = <fun>
```

4.2 Parcours hiérarchique d'un arbre

Dans le chapitre précédent, nous avions défini un arbre binaire strict par exemple de la sorte :

```
type 'a arbre = Feuille of 'a | Noeud of 'a * 'a arbre * 'a arbre
```

Nous avons vu que les parcours en profondeur s'écrivaient très simplement avec des fonctions récursives, mais nous avions laissé pour plus tard l'implémentation d'un parcours *en largeur*, ou *hiérarchique*, dans lequel on explore l'arbre « génération par génération ». On s'intéressera d'abord à la racine de l'arbre, puis à ses fils, puis aux fils de ses fils, etc. Ce parcours est rendu possible grâce aux files [17].

17. On travaille ici sur des arbres binaires stricts, mais l'idée sous-jacente s'étend de façon immédiate à des arbres quelconques.

Par exemple, on peut écrire une fonction affichant les entiers contenus dans un arbre d'entiers dans un ordre hiérarchique :

```
# let affiche_hierarchique arbre =









val affiche_hierarchique : int arbre -> unit = <fun>
```

5 Les dictionnaires

5.1 Introduction

Dans le chapitre précédent, nous avons brièvement étudié le problème de la gestion d'un annuaire. La solution proposée, une liste de couples (nom, numéro), présentait l'avantage de la simplicité, mais elle n'est guère efficace.

En effet, la recherche d'un numéro, connaissant le nom d'une personne, nécessite dans le pire des cas de parcourir l'intégralité de la liste. La consultation de l'annuaire est donc, formellement, en $O(n)$ où n correspond à la taille de l'annuaire.

On pourrait envisager, comme dans un véritable annuaire papier, de conserver les noms par ordre alphabétique. La méthode de recherche dichotomique [18] permet de réduire le temps de recherche en $O(\log(n))$, mais elle suppose que l'on puisse accéder en un temps en $O(1)$ à un quelconque élément de la liste dont on connaît la position.

Or, en OCaml, les listes ne permettent pas d'accéder aux éléments en $O(1)$. Il faut donc se retourner vers des tableaux [19] (`'a array`). Outre les éventuelles difficultés posées par la taille fixe des tableaux [20], le problème viendra de l'ajout ou de la suppression d'un élément : dans un tableau trié, cela impose de déplacer tous les éléments suivant l'élément ajouté ou retiré, ce qui donne une complexité en $O(n)$ pour l'ajout ou la suppression.

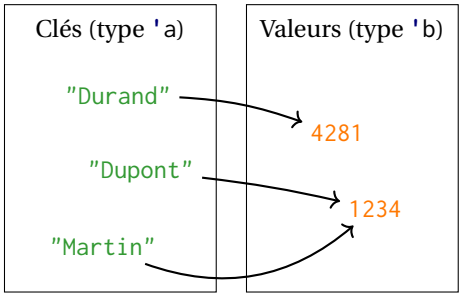
Il existe en fait une structure de données bien plus adaptée à cet usage, les *dictionnaires*.

5.2 Principe

Un *dictionnaire*, ou *table d'association*, est une structure de données qui associe des *clés* à des *valeurs*. Clés et valeurs peuvent être des objets de n'importe quel type (qui seront désignés par `'a` et `'b` dans la suite). On peut donc représenter un dictionnaire par un ensemble de couples (clé, valeur) où *toutes les clés sont distinctes* [21].

Cela convient donc tout à fait pour un annuaire, par exemple un annuaire d'entreprise, dans lequel le nom de chaque employé est associé à son numéro de poste. Un employé n'apparaît qu'une seule fois [22], mais certains employés peuvent partager un même poste.

La table d'association correspondante ressemblera par exemple à cela :



où les noms des employés (`'a` ≡ `string`) font office de clés tandis que les numéros de postes (`'b` ≡ `int` ou `string`) sont les valeurs qui leurs sont associées.

Dans un dictionnaire, il doit être aisé [23] de vérifier l'existence d'une clé, d'obtenir la valeur associée à une clé, et d'ajouter (ou de retirer) une clé (et sa valeur associée). Une telle structure de données permet de faire beaucoup de choses très efficacement dans les algorithmes [24].

18. Méthode étudiée en tronc commun, mais sur laquelle nous reviendrons ultérieurement.

19. Les « listes » en Python sont en fait bien plus proches de tableaux, à ceci près qu'il est possible de les redimensionner.

20. Difficultés que l'on a évoquées précédemment lorsque l'on a voulu implémenter une pile avec un tableau.

21. Dans le cas de clés mutables, distinctes pour l'égalité, pas pour l'identité, ce qui on le comprend peut occasionner des difficultés.

22. On supposera qu'il existe un moyen de différencier les éventuels homonymes.

23. Notamment en terme de rapidité pour exécuter ces opérations. Nous verrons qu'elles sont toutes $O(1)$ en moyenne dans le cas d'une implémentation normale d'un dictionnaire.

24. Et, pour une raison obscure, est très peu utilisée dans le cadre des concours. Il vaut très probablement mieux, d'ailleurs, éviter de l'utiliser sauf si le sujet vous y autorise explicitement.

5.3 Le module `Hashtbl`

Le module `Hashtbl` de la bibliothèque standard fournit un ensemble de fonctions permettant de manipuler des dictionnaires.

Un dictionnaire dont les clés sont de type `'a` et les valeurs associées de type `'b` aura pour type `('_a, '_b) Hashtbl.t`.

Le module fournit :

- une fonction `Hashtbl.create` de type[25] `int -> ('a, 'b) Hashtbl.t` qui prend en argument un entier[26] et retourne un dictionnaire vide ;
- une fonction `Hashtbl.add` de type `('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` qui prend en argument un dictionnaire, une clé et une valeur, et ajoute dans le dictionnaire le couple (clé, valeur) correspondant[27] ;
- une fonction `Hashtbl.mem` de type `('a, 'b) Hashtbl.t -> 'a -> bool` qui prend en argument un dictionnaire et une clé, et retourne un booléen indiquant si la clé est présente dans le dictionnaire ;
- une fonction `Hashtbl.find` de type `('a, 'b) Hashtbl.t -> 'a -> 'b` qui prend en argument un dictionnaire et une clé, et retourne la valeur associée à cette clé, ou lève l'exception `Not_found` si la clé n'est pas présente dans le dictionnaire ;
- une fonction `Hashtbl.remove` de type `('a, 'b) Hashtbl.t -> 'a -> unit` qui prend en argument un dictionnaire et une clé, et retire le couple (clé, valeur) correspondant du dictionnaire s'il existe (et ne fait rien si la clé n'est pas présente) ;
- une fonction `Hashtbl.clear` de type `('a, 'b) Hashtbl.t -> unit` qui prend en argument un dictionnaire et supprime tous les couples (clé, valeur) qu'il contient[28].

On remarquera qu'il est possible, avec un dictionnaire, de trouver une valeur connaissant la clé, mais pas de retrouver aisément une clé connaissant une valeur. Plusieurs raisons à cela, la principale tenant à la façon dont les dictionnaires sont implémentés. Mais aussi, nous n'avons pas imposé l'unicité des valeurs (et de fait, dans l'exemple précédent, deux clés conduisent à la même valeur), aussi la recherche en sens inverse serait, en l'état, ambiguë[29].

25. Les curieux qui iraient vérifier le type trouveront une signature un peu plus curieuse, car la fonction accepte un paramètre booléen *facultatif* supplémentaire, que nous passerons sous silence ici par souci de simplicité.
26. Qui détermine la taille initiale de la table, la documentation recommandant de choisir une valeur de l'ordre du nombre d'éléments qui seront rangés dans la table ; nous reviendrons sur ce point en étudiant l'implémentation des dictionnaires.
27. En fait, si la clé est déjà présente, la nouvelle valeur « cache » la valeur précédemment associée à la clé, un appel à `Hashtbl.remove` fait réapparaitre le couple précédent. Il existe une fonction `Hashtbl.replace` qui *remplace* la valeur associé à la clé (et crée un nouveau couple (clé, valeur) si la clé n'était pas présente).
28. En conservant la zone mémoire utilisée pour le stockage en mémoire du dictionnaire, dont la taille a pu croître avec le temps. Une fonction `Hashtbl.reset` permet, outre de vider le contenu, de réduire la mémoire utilisée à celle utilisée lors de la création du dictionnaire.
29. Dans l'hypothèse d'unicité des valeurs, on peut assez facilement d'obtenir une structure de données utilisable dans les deux sens, en maintenant à jour *deux* dictionnaires, un dans chaque sens !

5.4 Utilisation

L'utilisation du module `hashtbl` est simple. On crée d'abord un dictionnaire :

```
# let annuaire = Hashtbl.create 997;;
val annuaire : ('_a, '_b) Hashtbl.t = <abstr>
```

On remarquera que le type des associations est `('_a, '_b)`. C'est le signe que l'introduction du premier couple (clé, valeur) dans la table fixera de manière définitive les types des clés et des valeurs dans le dictionnaire.

Puis on ajoute les couples (clés, valeur) avec la fonction `add` :

```
# Hashtbl.add annuaire "Durand" 8241;;
- : unit = ()
```

On peut vérifier que l'instruction précédente a bien fixé les types des clés et valeurs (les premières comme des chaînes de caractères, les secondes comme des entiers) :

```
# annuaire;;
- : (string, int) Hashtbl.t = <abstr>
```

D'autres appels à la fonction `add` permettent de compléter le dictionnaire en ajoutant d'autres couples (clé, valeur) :

```
# Hashtbl.add annuaire "Dupont" 1029;;
- : unit = ()

# Hashtbl.add annuaire "Martin" 4281;;
- : unit = ()
```

On peut ensuite librement interroger le dictionnaire :

```
# Hashtbl.mem annuaire "Durand";;
- : bool = true

# Hashtbl.mem annuaire "Lechêne";;
- : bool = false
```

Et d'obtenir les valeurs associées aux clés :

```
# Hashtbl.find annuaire "Durand";;
- : int = 8241
```

Lorsque l'on tente d'accéder à la valeur associée à une clé qui n'est pas présente dans le dictionnaire, le module lève une exception `Not_found` indiquant qu'il n'a pu trouver la clé demandée :

```
# Hashtbl.find annuaire "Lechêne";;
Exception: Not_found.
```

On peut supprimer un couple (clé, valeur) avec la commande `Hashtbl.remove` :

```
# Hashtbl.find annuaire "Dupont";;
- : int = 1029

# Hashtbl.remove annuaire "Dupont";;
- : unit = ()

# Hashtbl.find annuaire "Dupont";;
Exception: Not_found.
```

On peut enfin vérifier que la fonction `Hashtbl.add`, lorsque la clé existe déjà, a pour effet de *remplacer*[30] l'ancien couple (clé, valeur) par le nouveau couple :

```
# Hashtbl.find annuaire "Durand";;
- : int = 8241

# Hashtbl.add annuaire "Durand" 6809;;
- : unit = ()

# Hashtbl.find annuaire "Durand";;
- : int = 6809
```

5.5 Implémentation naïve d'un dictionnaire

Bien que nous ayons déjà suggéré que cette solution serait inefficace, il est possible d'envisager, dans un premier temps, d'implémenter un tel dictionnaire en maintenant une liste des couples (clé, valeur).

Pour ce faire, on commence par définir notre dictionnaire comme un type contenant une liste (nécessairement mutable) de couples `'a * 'b` :

```
# type ('a, 'b) t = { mutable table : ('a * 'b) list };;
```

La création d'un nouveau dictionnaire revenant simplement à créer un objet de ce type, associé à une liste vide :

```
# let create () = { table = [] };;

val create : unit -> ('a, 'b) t = <fun>
```

La recherche d'une clé dans la table peut être effectuée récursivement :

```
# let find dict cle =
    let rec auxFind = function
      | []                  -> raise Not_found
      | (k, v)::q when k=cle -> v
      | _::q                -> auxFind q
    in auxFind dict.table;;

val find : ('a, 'b) t -> 'a -> 'b = <fun>
```

La fonction `mem` s'écrirait exactement de la même manière, mais retournerait `true` plutôt que `v` et `false` plutôt que de lever l'exception `Not_found`.

L'ajout d'un couple (ou la modification de la valeur associée à une clé déjà présente[31] [32]) est également implémentée via une fonction récursive :

```
# let add dict cle valeur =
    let rec auxAdd = function
      | []                  -> [ (cle, valeur) ]
      | (k, v)::q when k=cle -> (cle, valeur)::q
      | t::q                -> t::(auxAdd q)
    in dict.table <- (auxAdd dict.table);;

val add : ('a, 'b) t -> 'a -> 'b -> unit = <fun>
```

30. En fait, c'est un peu plus compliqué que cela... Comme on l'a dit, la nouvelle valeur « cache » l'ancienne, mais les anciennes valeurs ne sont pas supprimées pour autant. On peut retrouver toutes les valeurs grâce à la commande `Hashtbl.find_all` qui retourne une liste de valeurs, rangées par ancienneté décroissante. La fonction `Hashtbl.remove` a pour effet de retirer la valeur présentement associée à une clé, ce faisant rétablissant la valeur précédente. Si le but est de supprimer la clé du dictionnaire, il faudra donc effectuer autant d'appels `Hashtbl.remove` que l'on a effectué d'appels à `Hashtbl.add`, ou bien faire précéder les `Hashtbl.add` visant à modifier une clé d'un `Hashtbl.remove` à chaque fois !

31. On supposera qu'une clé ne peut apparaître qu'une seule fois dans la liste, ce qui sera bien le cas ici si l'on n'utilise que la fonction `Add` pour ajouter des couples (clé, valeur). La présente implémentation diffère quelque peu du vrai module `hashtbl` en ce sens que l'on n'a pas conservé la valeur antérieure associée à une clé, si elle existait, pour la rétablir lors d'un appel ultérieur à la fonction `remove`.

32. En fait, pour obtenir le même comportement que le dictionnaire fourni par Caml, il suffirait simplement d'ajouter le couple (clé, valeur) en tête de liste, sans se préoccuper de sa présence ou non de la clé plus loin dans la liste.

La suppression d'une clé pourrait quant à elle s'écrire :

```
# let remove dict cle =
    let rec auxRemove = function
      | []                    -> []
      | (k, _)::q when k=cle -> q
      | t::q                  -> t::(auxRemove q)
    in dict.table <- (auxRemove dict.table);;

val remove : ('a, 'b) t -> 'a -> unit = <fun>
```

Avec cette approche,

- la recherche d'une clé (ou de la valeur associée) est en temps linéaire $O(n)$ en fonction du nombre n de couples (clé, valeur) ;
- l'ajout d'une clé pourrait être en $O(1)$, mais comme il nous faut vérifier que la clé n'existe pas encore, on a un ajout en $O(n)$ également ;
- enfin, la suppression d'une clé serait également en $O(n)$.

On pourrait envisager améliorer les choses en utilisant un tableau que l'on maintiendrait trié selon l'ordre des clés[33], ce qui permettrait de retrouver une clé et la valeur qui lui est associée en $O(\log(n))$ par une recherche dichotomique. Mais dans ce cas, l'ajout d'un couple est en $O(n)$ car l'insertion du nouveau couple nécessite de décaler les éléments dans le tableau[34].

On peut en fait obtenir bien mieux que cela. Une solution possible est d'utiliser, en lieu et place de tableaux, des arbres, dits *arbres binaires de recherche*, pour implémenter cette idée de recherche dichotomique dans un ensemble trié.

5.6 Arbres binaires de recherche

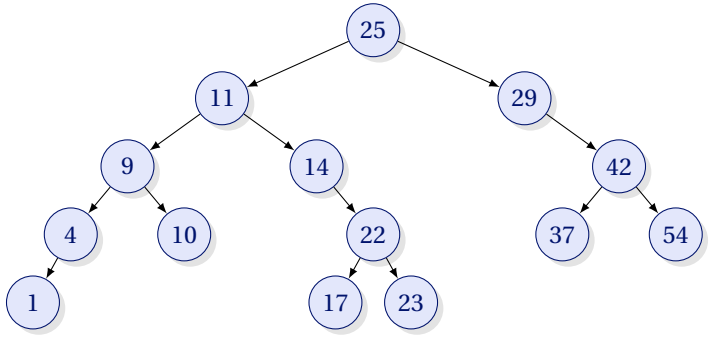
Structure d'un arbre binaire de recherche

Définition. On considère un ensemble $\mathcal{E}$ muni d'un ordre total $\preceq$.
Un arbre binaire étiqueté par $\mathcal{E}$ est qualifié d'*arbre binaire de recherche* lorsqu'il est soit vide, soit de la forme (F_g, x, F_d) et tel que

- F_g et F_d sont des arbres binaires de recherche ;
- tout étiquette y d'un nœud de F_g vérifie $y \preceq x$;
- tout étiquette y d'un nœud de F_d vérifie $y \succeq x$.

33. En supposant qu'il existe un ordre total sur l'ensemble des clés.
34. On n'a pas ce problème avec une liste, mais il n'est pas possible d'implémenter une recherche dichotomique en temps logarithmique dans une liste chaînée, car il est impossible d'accéder au milieu de la liste en $O(1)$!

L'arbre binaire ci-dessous a par exemple la structure d'un arbre binaire de recherche pour la comparaison usuelle $\leqslant$:



Recherche d'un élément

Nous nous servirons principalement des arbres de recherche pour représenter un ensemble, les étiquettes (généralement supposées toutes distinctes) représentant les éléments de l'ensemble. Dans la suite, nous dirons qu'un élément de $\mathcal{E}$ se trouve « dans l'arbre » s'il est égal à une des étiquettes de l'arbre.

Pour tester la présence d'un élément y dans un arbre binaire de recherche, on peut écrire une fonction récursive, utilisant les propriétés de tels arbres :

- l'élément n'est pas présent dans un arbre vide ;
- pour un arbre (F_g, x, F_d), si l'élément $x \neq y$, alors il suffit de chercher y dans le sous-arbre F_g si $y \preceq x$ et dans le sous-arbre F_d dans le cas contraire.

La recherche de la présence d'un élément dans l'arbre a, de façon évidente, une complexité majorée par la hauteur de l'arbre, soit en $O(h(A))$. On ne vérifie donc qu'un nombre d'éléments qui peut être très petit devant $|A|$ (de l'ordre de $\log(|A|)$ si, à chaque étape de l'algorithme, les tailles du sous-arbre gauche et du sous-arbre droit sont similaires), ce qui fait tout l'intérêt de la structure d'arbre binaire de recherche par rapport à un conteneur linéaire tel qu'une liste.

La traduction en OCaml est immédiate ;

```
# let rec contient y = function
    | Nil                      -> false
    | Noeud (_, x, _)  when y=x -> true
    | Noeud (fg, x, _) when y<x -> contient y fg
    | Noeud (_, _, fd)          -> contient y fd;;

val contient : 'a -> 'a arbre -> bool = <fun>
```

Plus petit/plus grand élément

De part la structure d'un arbre de recherche, pour trouver le plus petit élément pour la relation $\preceq$, il suffit de suivre la branche de gauche tant que l'on n'atteind pas « `Nil` » (notons que le nœud que l'on recherche n'est pas nécessairement une feuille, mais simplement n'a pas de fils gauche). Cette recherche aura une complexité temporelle en $O(\mathrm{h}(\mathrm{A}))$.

Un arbre vide n'ayant pas de plus petit élément, il paraît naturel de provoquer une erreur si l'on tente de trouver un plus petit élément dans un tel arbre. Beaucoup de fonctions ne pouvant retourner un résultat pour un arbre vide, il est utile de créer une exception :

```
# exception Empty
```

La fonction retournant le plus petit élément s'écrira donc ainsi :

```
# let rec plusPetit = function
    | Nil               -> raise Empty
    | Noeud (Nil, x, _) -> x
    | Noeud (fg,  _, _) -> plusPetit fg;;

val plusPetit : 'a arbre -> 'a = <fun>
```

Alternativement, on pourrait, de façon équivalente, systématiquement chercher le plus petit (plus grand) élément dans le fils gauche (droit) et sans chercher à savoir s'il est vide, et si cette recherche lève l'exception `Empty`, retourner l'étiquette de la racine :

```
# let rec plusPetit = function
    | Nil               -> raise Empty
    | Noeud (fg, x, _) -> try plusPetit fg with Empty -> x;;

val plusPetit : 'a arbre -> 'a = <fun>
```

Pour obtenir le plus grand élément, on parcourt de même la branche la plus à droite.

Prédécesseur et successeur

Définition. On considère un ensemble $\mathcal{E}$ muni d'un ordre total $\preceq$, et un arbre binaire de recherche A étiqueté par $\mathcal{E}$. Soit un élément $y \in \mathcal{E}$.
Le *prédécesseur* de y dans l'arbre A est la plus grande (pour $\preceq$) des étiquettes [a] de A strictement inférieure à y. Le *successeur* de y dans l'arbre A est la plus petite (pour $\preceq$) des étiquettes de A strictement supérieure à y.

a. Les termes prédécesseurs et successeurs peuvent également désigner le nœud portant l'étiquette correspondante (il n'y a alors unicité que si les étiquettes sont toutes distinctes).

Pour l'arbre proposé en exemple, le prédécesseur de 31 est 29, le successeur de 31 est 37.

Pour trouver le prédécesseur d'un élément y dans un arbre binaire de recherche A, on utilisera une démarche récursive :

- si A est l'arbre vide, un tel prédécesseur n'existe pas ;
- si A est de la forme $(\mathrm{F}_g, x, \mathrm{F}_d)$ avec $y \preceq x$, on recherche le prédécesseur dans le sous-arbre gauche F_g ;
- sinon, A est de la forme $(\mathrm{F}_g, x, \mathrm{F}_d)$ avec $x \prec y$, et deux cas peuvent se présenter : soit on trouve un prédécesseur dans le sous-arbre droit (qui sera plus grand que l'étiquette située à la racine, et le prédécesseur recherché), soit il n'y a pas de prédécesseur dans l'arbre droit, et c'est l'étiquette de la racine qui convient.

Pour traduire cet algorithme en OCaml, on utilisera l'exception `Not_found` pour indiquer qu'il n'y a aucun prédécesseur dans un arbre binaire de recherche fourni en paramètre, ce qui nous permettra, avec un `try ... with`, d'implémenter le test du dernier cas. Cela donne donc (la fonction retournant le successeur s'écrirait de façon similaire) :

```
# let rec prec y = function
    | Nil                          -> raise Not_found
    | Noeud(fg, x, _) when y <= x -> prec y fg
    | Noeud(_, x, fd)              -> try prec y fd
                                      with Not_found -> x;;

val prec : 'a -> 'a arbre -> 'a = <fun>
```

Cette fois encore, la fonction a une complexité en $O(\mathrm{h}(\mathrm{A}))$.

Parcours infixe d'un ABR

Le parcours en profondeur infixe d'un arbre binaire de recherche présente un intérêt particulier. En effet, un tel parcours traite les descendants gauches d'un nœud avant celui-ci, et ses descendants droits après. Or, les descendants gauches du nœud portent tous des étiquettes plus petites pour $\preceq$ que l'étiquette du nœud considéré, et les descendants droits des étiquettes plus grandes. Les nœuds sont donc traités dans un ordre croissant pour $\preceq$.

Cela nous donne un moyen simple de vérifier qu'un arbre binaire respecte les conditions d'un arbre binaire de recherche : il faut et suffit que les étiquettes, dans un parcours en profondeur infixe, soient rangées par ordre croissant.

On peut par exemple écrire une fonction qui visite les nœuds dans un parcours en profondeur infixe, et vérifie que chaque étiquette est plus grande que la précédente considérée. On peut conserver la dernière étiquette dans une référence.

La première étiquette visitée est un cas un peu particulier, car il est inutile de la comparer à quoi que ce soit. Pour la traiter comme les autres, toutefois, on initialisera la référence

avec la première étiquette, à savoir celle retournée par la fonction `plusPetit`[35].

```
# let verifie arbre =
    let dernier = ref (plusPetit arbre) in
      let rec dfs = function
        | Nil -> true
        | Noeud (fg, x, fd) -> (dfs fg) &&
          (if !dernier <= x then (dernier := x; true) else false) &&
          (dfs fd)
      in dfs arbre;;

val verifie : 'a arbre -> bool = <fun>
```

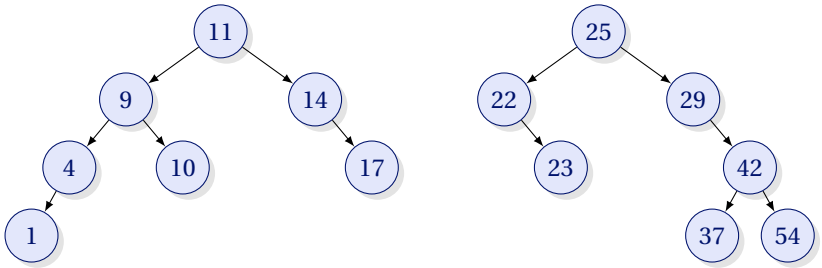
Partition

Une partition d'un arbre binaire de recherche par rapport à un élément y consiste en deux arbres binaires de recherche[36] contenant les nœuds de l'arbre original, l'un n'ayant que des éléments x vérifiant $x \preceq y$, l'autre des éléments vérifiant $y \preceq x$.

On peut obtenir une partition de façon récursive en écrivant :

```
# let rec partition y = function
    | Nil -> Nil, Nil
    | Noeud (fg, x, fd) when x <= y
      -> let a1, a2 = partition y fd in Noeud (fg, x, a1), a2
    | Noeud (fg, x, fd)
      -> let a1, a2 = partition y fg in a1, Noeud (a2, x, fd);;

val partition : 'a -> 'a arbre -> 'a arbre * 'a arbre = <fun>
```

La fonction précédente, appliquée à l'arbre utilisé en exemple pour une valeur $y = 18$, donne les deux arbres binaires de recherche suivants :



35. Si l'arbre n'est pas un arbre binaire de recherche, l'élément retourné par la fonction n'est pas nécessairement le plus petit, mais cela reste l'étiquette la plus à gauche, la première visitée par le parcours en profondeur.
36. Même si toutes les étiquettes sont distinctes, il n'y a pas unicité pour la structure des deux arbres.

La structure des arbres binaires de recherche permet d'obtenir cette partition, une fois encore, avec une complexité temporelle $O(\mathrm{h(A)})$. Notons par ailleurs que la hauteur des deux arbres obtenus est inférieure ou égale à la hauteur de l'arbre original.

Démonstration. Cette propriété est vraie pour un arbre vide.
Supposons cette propriété vraie pour tout arbre de hauteur inférieure ou égale à h ; pour un arbre de hauteur $h+1$, on a $\mathrm{h}(\mathrm{F}_d) \leqslant h$ et $\mathrm{h}(\mathrm{F}_g) \leqslant h$, donc les arbres `a1` et `a2` issus de la partition de F_g ou F_d auront également une hauteur majorée par h, et l'arbre `Noeud(fg, x, a1)` (respectivement l'arbre `Noeud(a2, f, fd)`) aura une hauteur majorée par $h+1$, donc la propriété est vraie pour un arbre de hauteur $h+1$. □

Insertion (adjonction) d'un élément dans un ABR

Il existe deux stratégies pour insérer un élément dans un arbre binaire de recherche : au niveau de la racine, et au niveau des feuilles.

Profitons de l'occasion pour rappeler que l'on travaille avec des arbres immutables, donc une fonction insérant un élément dans un arbre ne *modifie* pas cet arbre, mais crée un *nouvel* arbre contenant l'élément inséré[37].

Insertion au niveau des feuilles Pour insérer un élément y dans un arbre binaire de recherche au niveau des feuilles, on procède de façon récursive :

- insérer un élément y dans un arbre vide consiste simplement à retourner une feuille (soit `Noeud(Nil, y, Nil)`) ;
- insérer un élément y dans un arbre $(\mathrm{F}_g, x, \mathrm{F}_d)$ lorsque $y \preceq x$ revient à insérer l'élément y dans le sous-arbre gauche[38] : on crée donc un nouvel arbre, dont la racine porte toujours l'étiquette x, avec le même sous-arbre droit F_d, et avec pour sous-arbre gauche le résultat de l'insertion de y dans F_g ;
- sinon, lorsque $x \prec y$, il en est de même, mais avec une insertion de y dans le sous-arbre droit.

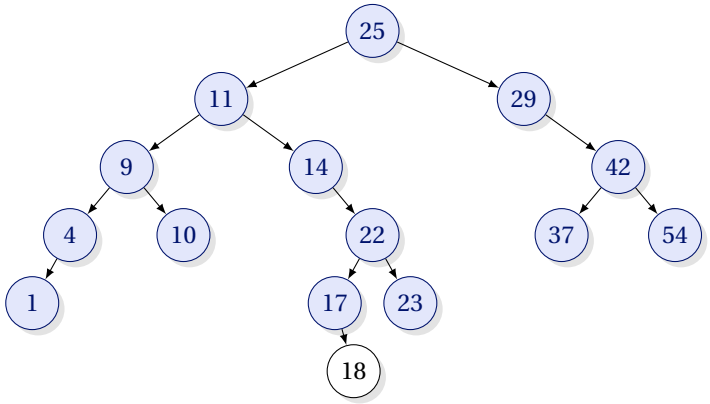
En OCaml, cela donne :

```
# let rec insere y = function
    | Nil                          -> Noeud(Nil, y, Nil)
    | Noeud(fg, x, fd) when y <= x -> Noeud(insere y fg, x, fd)
    | Noeud(fg, x, fd)             -> Noeud(fg, x, insere y fd);;

val insere : 'a -> 'a arbre -> 'a arbre = <fun>
```

La complexité de cette insertion est $O(\mathrm{h(A)})$. Dans le pire des cas, l'ajout d'une feuille supplémentaire augmente de 1 la hauteur de l'arbre.

37. Attention, la plupart du temps, les deux arbres ne sont pas indépendants, et peuvent avoir des branches en commun. Mais tant que l'on reste dans le cadre d'objets immutables, cela n'a pas d'importance.
38. Dans le cas où $y = x$, on pourrait tout aussi bien l'insérer dans le sous-arbre droit.

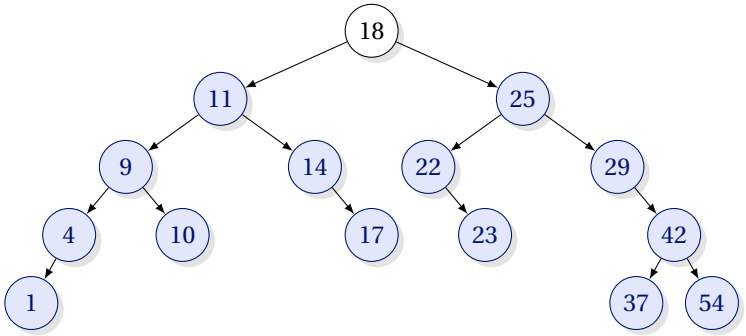
L'insertion d'un nœud avec une étiquette 18 dans l'arbre d'exemple donnerait :



Insertion à la racine L'insertion d'un élément y au niveau de la racine est un peu plus délicate. En effet, cela a des conséquences plus importantes sur le forme de l'arbre. En effet, si l'élément inséré y et l'étiquette de la racine x vérifient par exemple $y \preceq x$, une partie des nœuds de la branche gauche de l'arbre vont devoir migrer dans la branche droite. Cependant, nous avons déjà écrit une fonction capable de partitionner un arbre binaire de recherche en deux arbres binaires de recherche, selon que les étiquettes soient inférieures ou supérieures à un élément y. On peut donc s'en servir pour écrire notre insertion :

```
# let insere y arbre =
    let fg, fd = partition y arbre in Noeud(fg, y, fd);;

val insere : 'a -> 'a arbre -> 'a arbre = <fun>
```

L'insertion d'un nœud portant l'étiquette 18 au niveau de la racine de notre exemple d'arbre binaire de recherche conduirait au résultat suivant :



La complexité de cette insertion est celle de `partition`, $O(\mathrm{h}(\mathrm{A}))$. Dans le pire des cas, l'insertion augmente également de 1 la hauteur de l'arbre.

Suppression d'un élément dans un ABR

La suppression d'un nœud portant une étiquette donnée dans un arbre binaire de recherche (ou le nœud de profondeur minimale vérifiant cette propriété si plusieurs nœuds pourraient convenir) est une opération un peu plus délicate car, comme dans le cas d'une insertion au niveau de la racine, elle peut nécessiter de nombreux changements dans la structure de l'arbre.

Pour préparer cette tâche de suppression, nous allons tout d'abord écrire quelques fonctions auxiliaires qui nous seront utiles. Tout d'abord, une fonction permettant d'extraire la plus petite étiquette, et de retourner cette étiquette, et un arbre binaire de recherche où le nœud portant cette étiquette a été retiré :

```
# let rec retireMin = function
    | Nil               -> raise Empty
    | Noeud(Nil, x, fd) -> x, fd
    | Noeud(fg, x, fd)  -> let minimum, arbre = retireMin fg
                           in minimum, Noeud(arbre, x, fd);;

val retireMin : 'a arbre -> 'a * 'a arbre = <fun>
```

Cette fonction permet d'écrire une fonction fusionnant deux arbres binaires de recherche tels que tout éléments x et x' issus respectivement du premier et du second arbre vérifient $x \preceq x'$, en créant un arbre avec pour racine le plus petit élément du second arbre, pour fils gauche le premier arbre, et pour fils droit le second arbre privé de son plus petit élément [39] :

```
# let rec fusion a1 = function
    | Nil -> a1
    | a2  -> let minimum, arbre = retireMin a2
             in Noeud(a1, minimum, arbre);;

val fusion : 'a arbre -> 'a arbre -> 'a arbre = <fun>
```

On peut enfin écrire notre fonction supprimant un élément y d'un arbre binaire de recherche :

- il est impossible de le supprimer d'un arbre vide ;
- si l'étiquette de la racine est l'élément à supprimer, alors il suffit de retourner la fusion des deux fils ;
- si l'étiquette x de la racine vérifie $y \prec x$, alors on cherche à supprimer y dans le sous-arbre gauche ;
- si, au contraire $x \prec y$, on s'intéresse au sous-arbre droit.

39. On aurait évidemment pu tout aussi bien prendre pour racine le plus grand élément du premier arbre, et pour fils le reliquat du premier arbre et l'intégralité du second.

Cela donne, en OCaml :

```
# let rec supprime y = function
    | Nil                          -> raise Not_found
    | Noeud(fg, x, fd) when y = x -> fusion fg fd
    | Noeud(fg, x, fd) when y < x -> Noeud(supprime y fg, x, fd)
    | Noeud(fg, x, fd)            -> Noeud(fg, x, supprime y fd);;

val supprime : 'a -> 'a arbre -> 'a arbre = <fun>
```

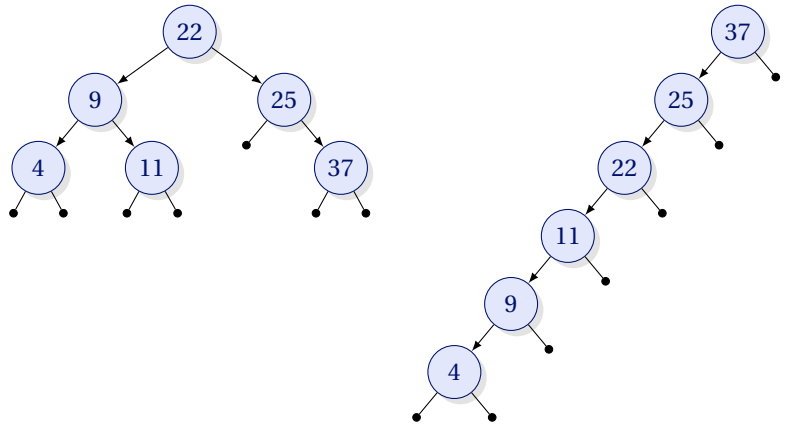
Un peu de complexité

Parmi toutes les opérations que l'on effectue sur un arbre de recherche, lesquelles étant implémentées par des algorithmes récursifs, on peut distinguer deux situations :

- les algorithmes dont la récursion se fait uniquement sur l'un des deux fils (gauche ou droit), tels que la recherche d'un élément, et dont la complexité temporelle sera la plupart du temps [40] en $O(\mathrm{h}(\mathrm{A}))$;
- les algorithmes dont la récursion porte (ou peut porter) sur les *deux* fils, tels qu'un parcours de l'arbre, et dont la complexité temporelle atteindra souvent [41] $O(|\mathrm{A}|)$.

Rappelons, pour un arbre binaire, l'encadrement $\lfloor \log_2(|\mathrm{A}|) \rfloor \leqslant \mathrm{h}(\mathrm{A}) \leqslant |\mathrm{A}| - 1$. Pour des raisons d'efficacité pour les algorithmes dont la complexité est en $O(\mathrm{h}(\mathrm{A}))$, il est préférable que la hauteur $\mathrm{h}(\mathrm{A})$ d'un arbre soit aussi proche que possible de la limite inférieure, $\lfloor \log_2(|\mathrm{A}|) \rfloor$.

Ainsi, pour un même ensemble de six éléments, on préférera pour des raisons d'efficacité travailler avec l'arbre binaire de recherche de gauche plutôt que celui de droite (que l'on qualifie parfois d'« arbre-peigne » :



40. Tant que les opérations sur chaque nœud sont en $O(1)$.
41. Toujours pour des opérations en $O(1)$ sur les nœuds.

Pour s'assurer que l'on ne travaille qu'avec des arbres binaires de recherche dont la forme est « favorable », on peut ajouter des contraites sur les arbres que l'on manipule.

Définition. On dira travailler avec un ensemble $\mathcal{A}_\mathrm{B}$ d'arbres binaires *équilibrés* si, pour tout arbre binaire $\mathrm{A} \in \mathcal{A}_\mathrm{B}$, sa hauteur vérifie la relation $\mathrm{h}(\mathrm{A}) = O\big(\log(|\mathrm{A}|)\big)$.

Lorsque l'on travaille sur un ensemble $\mathcal{A}_\mathrm{B}$ d'arbres binaires équilibrés, les nombreuses fonctions que nous avons présentées, de complexité temporelle $O(\mathrm{h}(\mathrm{A}))$, ont en fait un coût logarithmique $O\big(\log(|\mathrm{A}|)\big)$, ce qui les rend très efficaces. il existe de nombreuses familles d'arbres binaires de recherches équilibrés (AVL, arbres rouges-noirs, etc.), et il est possible d'appliquer toutes les opérations précédentes sur de tels arbres. Leur étude dépasse le cadre de ce cours.

Utilisation pour implémenter un dictionnaire

Pour implémenter un dictionnaire à l'aide d'un arbre binaire de recherche, on place au niveau des nœuds des étiquettes contenant des couples (clé, valeur). Dans ce but, on définit un type :

```
# type ('a,'b) enregistrement = { cle: 'a; vlr: 'b };;
```

On suppose disposer ici d'une relation d'ordre total $\preceq$ sur l'ensemble des clés car, dans l'arbre binaire de recherche, *les éléments seront ordonnées en fonction de leur clé.*

Les opérations courantes sur un dictionnaire sont :

- la recherche d'une clé donnée dans le dictionnaire, et le renvoi de la valeur associée (on lèvera une exception `Not_found` si la clé n'est pas présente) ;
- la modification d'une valeur associée à une clé ;
- l'ajout d'un nouveau couple (clé, valeur) ;
- la suppression d'une clé et de la valeur qui lui est associée.

Toutes ces opérations peuvent aisément être réalisées à partir des fonctions que l'on a déjà écrites sur les arbres binaires de recherche, au prix de quelques changements dûs au typage des étiquettes.

La recherche peut être écrite ainsi :

```
# let rec recherche k = function
    | Nil                          -> raise Not_found
    | Noeud(_,  x, _) when k=x.cle -> x.vlr
    | Noeud(fg, x, _) when k<x.cle -> recherche k fg
    | Noeud(_,  _, fd)             -> recherche k fd;;

val recherche : 'a -> ('a, 'b) enregistrement arbre -> 'b = <fun>
```

L'ajout ou la modification (si la clé existe, on modifie la valeur, et si elle n'existe pas, on crée un nouveau nœud dans l'arbre étiqueté par le couple (clé, valeur)) s'écrira [42] :

```
# let rec modifie k v = function
    | Nil                          -> Noeud(Nil, { cle=k; vlr=v }, Nil)
    | Noeud(fg, x, fd) when k=x.cle -> Noeud(fg, { cle=k; vlr=v }, fd)
    | Noeud(fg, x, fd) when k<x.cle -> Noeud(modifie k v fg, x, fd)
    | Noeud(fg, x, fd)              -> Noeud(fg, x, modifie k v fd);;

val modifie : 'a -> 'b -> ('a, 'b) enregistrement arbre
                -> ('a, 'b) enregistrement arbre = <fun>
```

La suppression d'une clé reste l'opération la plus délicate, mais les changements sont très limités par rapport aux fonctions précédemment étudiées :

```
# let rec supprime y arbre =
    let rec retireMin = function
      | Nil              -> raise Empty
      | Noeud(Nil, x, fd) -> x, fd
      | Noeud(fg, x, fd)  -> let minimum, arbre = retireMin fg
                             in minimum, Noeud(arbre, x, fd)

    and fusion a1 = function
      | Nil -> a1
      | a2  -> let minimum, arbre = retireMin a2
               in Noeud(a1, minimum, arbre)

    in match arbre with
      | Nil                          -> raise Not_found
      | Noeud(fg, x, fd) when y=x.cle -> fusion fg fd
      | Noeud(fg, x, fd) when y<x.cle -> Noeud(supprime y fg, x, fd)
      | Noeud(fg, x, fd)              -> Noeud(fg, x, supprime y fd);;

val supprime : 'a -> ('a, 'b) enregistrement arbre
                -> ('a, 'b) enregistrement arbre = <fun>
```

Bien évidemment, ces opérations sont en $O(\mathrm{h(A)})$ et non en $O(1)$ comme c'est le cas pour une implémentation à l'aide d'une table de hachage. Toutefois, l'implémentation nécessite généralement moins de mémoire (il n'est pas nécessaire d'avoir une table avec de nombreuses cases vides) et plus simple (pas besoin non plus d'augmenter la taille de la table de hachage lorsque les collisions deviennent nombreuses).

Par ailleurs, si l'on parvient à garder un arbre raisonnablement équilibré, la hauteur de l'arbre sera de l'ordre de $\log(n)$ pour un dictionnaire contenant n éléments, et les opérations courantes seront donc en $O\big(\log(n)\big)$. La différence entre $O(1)$ et $O\big(\log(n)\big)$ n'est pas suffisamment marquée pour qu'elle emporte un choix d'implémentation. Rappelons en effet que pour $n = 10^6$ par exemple, $\log_2\big(10^6\big) \simeq 20$. Les opérations sur un dictionnaire à un million de clés ne nécessiteront donc généralement, tant que l'arbre reste équilibré, pas plus de deux douzaines de comparaisons, ce qui reste très efficace.

On y gagne également la possibilité d'implémenter d'autres opérations, telles que la recherche efficace d'un successeur/prédécesseur dans l'ensemble des clés.

Les arbres binaires de recherche sont donc une excellente façon d'implémenter un dictionnaire, à condition toutefois que l'on y inclue un mécanisme garantissant que l'arbre binaire de recherche reste équilibré (arbres AVL, arbres rouge-noir, etc.)

5.7 Tables de hachage

Principe

Une autre solution très efficace pour implémenter un dictionnaire consiste à utiliser une *table de hachage*, qui permettra, sous certaines conditions, d'obtenir une complexité en temps en O(1) pour les opérations de recherche, d'ajout, de modification et de suppression d'une clé.

Le principe est relativement simple : la structure qui nous permettait d'accéder directement aux données en temps constant est celle d'un tableau. On crée donc un tableau de m cases [43], mais il nous faut un moyen d'associer une quelconque clé à un entier entre 0 et $m-1$, correspondant à l'une des cases. Cela se fait en deux temps :

- on utilise une *fonction de hachage* qui transforme un objet de n'importe quel type en un entier (la fonction `Hashtbl.hash` du module `hashtbl`, de signature `'a -> int`, peut par exemple être utilisée à cet effet) ;
- on restreint le résultat dans l'intervale $[\![0\,..\,m-1]\!]$ par exemple en utilisant le reste d'une division entière.

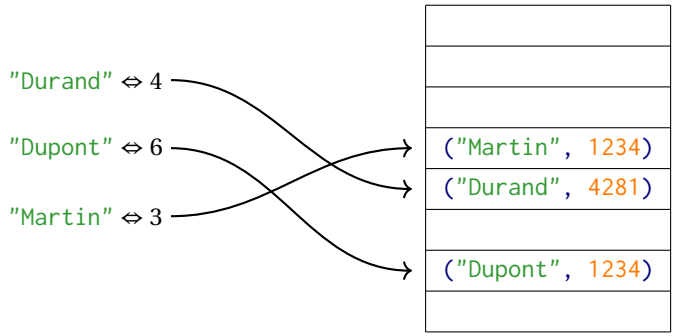
Par exemple, si l'on souhaite ranger le couple `(Durand, 8241)` dans un tableau contenant 8 cases, on commence par déterminer dans quelle case il convient de le ranger, en utilisant une fonction de hachage :

```
# Hashtbl.hash "Durand";;
- : int = 679905404

# Hashtbl.hash "Durand" mod 8;;
- : int = 4
```

42. On a choisi un ajout au niveau des feuilles, on pourrait envisager un ajout au niveau de la racine.

43. On appelle généralement ces cases « *buckets* », ou « *seaux* » en français.

Puis le couple (Durand, 8241) est alors glissé dans la case correspondant au résultat fourni par la fonction de hachage ramené, par un modulo, au nombre de cases du tableau, et de même pour les autres couples (clé, valeur) :



Dès lors, tout est plus rapide : pour savoir si une clé est présente dans le dictionnaire, et pour récupérer sa valeur, on utilise la fonction de hachage pour déterminer dans quelle case la clé devrait se trouver, et on n'a alors qu'une seule et unique case à examiner. Il en est de même pour les autres opérations.

Il est important de conserver dans les cases non seulement les valeurs, mais également les clés. En effet, le nombre de clés possibles excède très largement le nombre `m` de cases dans le tableau. Par exemple, dans un tableau avec 8 cases, `"Dubois"` conduirait également à la case `3`, et une recherche avec la clé `"Dubois"` ne doit pas retourner la valeur associée à la clé `"Martin"` simplement parce que les deux clés correspondent à la même case! Une fois la case identifiée, il faut donc vérifier la clé. Bien évidemment, si le dictionnaire doit contenir à la fois les clés `"Martin"` et `"Dubois"`, on voit venir un problème, nous reviendrons sur ce point un peu plus loin.

Idéalement, la fonction de hachage devrait pouvoir :

- fournir un résultat rapidement (pour que l'on puisse accéder rapidement aux éléments du dictionnaire) ;
- donner une distribution aussi uniforme que possible sur l'ensemble $[\![0\,..\,N-1]\!]$ des valeurs retournées.

La seconde condition vise à limiter les *collisions*, c'est-à-dire les clés qui se retrouvent dans une même case du tableau, telles que `"Martin"` et `"Dubois"`. En effet, même avec un nombre de cases m supérieur au nombre n de clés, on risque de se retrouver avec plusieurs clés dans la même case. On peut montrer que, quelle que soit la fonction de hachage choisie, la probabilité d'avoir une collision est d'au moins

$$1 - \frac{m!}{m^n(m-n)!}$$

ce qui donne, pour un tableau de 1000 cases, une probabilité de plus de 71% de chances d'avoir au moins une collision avec seulement 50 clés [44].

Ces collisions devront ensuite être prises en compte. Une solution [45] consiste à dire que chaque case peut contenir une liste de couples (clé, valeur).

Tant que le nombre de clés dans la table d'association est, au plus, de l'ordre du nombre de cases, et sous réserve que la fonction de hachage ait des propriétés satisfaisantes en terme de répartition, on aura au plus quelques couples dans chacune des cases, et l'accès aux clés et aux valeurs pourra rester en un temps constant. Tout se passe comme si le dictionnaire contenait un grand nombre de dictionnaires distincts, un pour chacune des valeurs de l'intervalle $[\![0\,..\,m-1]\!]$.

Une implémentation possible

Un type définissant un dictionnaire implémenté avec une table de hachage serait donc, par exemple :

```
# type ('a, 'b) t = { table : ('a * 'b) list array };;
```

On construit un nouveau dictionnaire en créant un tableau à n cases (n étant passé en argument), contenant des listes vides destinées à recueillir des couples (clé, valeur) :

```
# let create n = { table = Array.make n [] };;
val create : int -> ('a, 'b) t = <fun>
```

La recherche [46] de la valeur associée à une clé se passe comme précédemment (en particulier, il s'agit de la *même* fonction auxilliaire `auxFind`), mais dans une liste, normalement courte, prise dans la case du tableau désignée par le résultat `hsh` du hachage de la clé :

```
# let find dict cle =
  let rec auxFind = function
      | []                  -> raise Not_found
      | (k, v)::q when k=cle -> v
      | _::q                -> auxFind q
    in let hsh = Hashtbl.hash cle mod (Array.length dict.table)
      in auxFind dict.table.(hsh);;

val find : ('a, 'b) t -> 'a -> 'b = <fun>
```

44. Situation connue sous le nom de « paradoxe des anniversaires », puisque cette même formule indique que, à supposer que les naissances soient réparties uniformément sur l'année, il y a presque 95% de chances d'avoir deux personnes nées le même jour de l'année dans une classe de 45 étudiants.

45. La plus courante, même si ce n'est pas la plus efficace en terme de mémoire. Il en existe de nombreuses autres, par exemple la solution consistant, lorsque la case désignée est déjà occupée par une autre clé, à utiliser itérativement une autre fonction de hachage jusqu'à trouver une case libre. On peut montrer que sous certaines conditions, sur la fonction de hachage et la taille du tableau par rapport au nombre de clés, cette approche présente des avantages.

46. La fonction `mem`, déterminant simplement la présence d'une clé dans la table, s'écrirait de la même manière.

Même chose pour la fonction add, la liste de couples (clé, valeur) potentiellement allongée étant replacée dans la case idoine :

```
# let add dict cle valeur =
    let rec auxAdd = function
      | []                  -> [ (cle, valeur) ]
      | (k, v)::q when k=cle -> (cle, valeur)::q
      | t::q                -> t::(auxAdd q)
    in let hsh = Hashtbl.hash cle mod (Array.length dict.table)
      in dict.table.(hsh) <- (auxAdd dict.table.(hsh));;

val add : ('a, 'b) t -> 'a -> 'b -> unit = <fun>
```

Et enfin, la fonction remove :

```
# let remove dict cle =
    let rec auxRemove = function
      | []                  -> []
      | (k, _)::q when k=cle -> q
      | t::q                -> t::(auxRemove q)
    in let hsh = Hashtbl.hash cle mod (Array.length dict.table)
      in dict.table.(hsh) <- (auxRemove dict.table.(hsh));;

val remove : ('a, 'b) t -> 'a -> unit = <fun>
```

Le lecteur ne manquera pas de remarquer que nous avons utilisé la fonction `Hashtbl.hash` plutôt que de la reprogrammer. En effet, cette fonction est particulièrement difficile à écrire, tant d'un point de vue théorique (pour qu'elle respecte les critères précédemment énoncés) que pratique (il n'existe pas de manière simple d'écrire une fonction traitant un argument de type quelconque).

Il reste une chose à envisager : lorsque le nombre de couples devient trop important dans certaines cases, il nous faut agrandir le tableau. Pour ce faire, il doit être possible de remplacer le tableau, aussi nous faut-il modifier notre type dictionnaire par exemple par

```
# type ('a, 'b) t = { mutable table : ('a * 'b) list array };;
```

Ensuite, lorsque le tableau devient trop petit (trop de collisions, c'est-à-dire des listes qui deviennent longues dans certaines cases), on crée un tableau plus grand, et on hashe à nouveau toutes les clés pour replacer les couples dans les bonnes cases. Plutôt que de doubler la taille m de la table de hachage, on peut être tenté de choisir comme nouvelle taille $2m+1$. En effet, si la nouvelle taille était un multiple de la précédente, vu la façon dont on calcule le numéro de la case, on séparerait le contenu de chaque case isolément (un élément dans la case 3 d'une table de taille 8 ne pourrait se retrouver que dans les cases 3 et 11 d'une table de taille 16), plutôt que de procéder à une nouvelle répartition globale des couples dans la table (un élément dans la case 3 d'une table de taille 8 peut se retrouver dans n'importe quelle case d'une table de taille 17). On peut ainsi parfois obtenir un meilleur résultat[47].

On peut par exemple utiliser la fonction suivante :

```
let extend dict =
  let m = Array.length dict.table in     (* Ancienne taille m        *)
    let p = 2*n+1 in                     (* Nouvelle taille p = 2m+1 *)
      let n_table = Array.make p [] in   (* Nouveau table de hachage *)
        let addkv = function (k, v)
          -> let hsh = Hashtbl.hash k mod p
               in n_table.(hsh) <- (k, v)::n_table.(hsh)
        in for i=0 to n-1 do             (* On la remplit avec tous  *)
          List.iter addkv dict.table.(i) (* les couples (clé,valeur) *)
        done;                            (* dans l'ancienne table    *)
      dict.table <- n_table;;   (* Elle remplace la table précédente *)

val extend : ('a, 'b) t -> unit = <fun>
```

Le coût de cette inflation est $O(m+n)$ où m est la taille de la table et n le nombre de clés (coût de la création du tableau, puis coût de son remplissage). Cependant, on en vient à augmenter la taille du tableau en général lorsque les listes dans les cases s'allongent, ce qui arrivent seulement lorsque n n'est plus petit devant m, aussi la fonction est $O(n)$ en pratique. Cela coûte cher, mais l'opération ne sera plus effectuée avant un nombre d'ajouts de l'ordre de n puisque la taille de la table est doublée, donc en moyenne, le coût de l'augmentation de la table de hachage est $O(1)$ lors d'un ajout d'un couple (clé, valeur), ce qui en fait une opération raisonnable.

Il reste à détecter le besoin d'agrandir la table, ce qui peut se faire sur le nombre de couples dans la table (facile à comptabiliser) ou bien à partir de statistiques (que l'on peut mettre à jour sans surcout notable lors des ajouts et recherches). La fonction `Hashtbl.stat` permet justement d'obtenir des statistiques sur le remplissage de la table de hachage d'un dictionnalre.

Pour conclure, signalons que les clés devraient être des objets immutables[48] : si l'on mute une clé, son hachage va changer, et on cherchera possiblement la clé dans la mauvaise case du tableau ! Par exemple, avec notre implémentation du dictionnaire[49] :

47. Il peut aussi être moins bon, parfois... Cet aspect est à contrebalancer avec l'utilisation d'un m qui soit toujours une puissance de deux pour faciliter le calcul du reste de la division entière par m.
48. C'est une des raisons pour lesquelles les chaînes sont immutables en Python, qui utilise considérablement dans son fonctionnement des dictionnaires.
49. Mais il en serait de même avec l'implémentation du module `Hashtbl`.

```
# let dico = create 8;;
val dico : ('_a, '_b) t = {table = [|[]; []; []; []; []; []; []; []|]}

# let ch = "Hello";;
val ch : string = "Hello"

# add dico ch 42;;          (* On ajoute le couple ("Hello", 42) *)
- : unit = ()

# Hashtbl.hash ch mod 8;; (* Il a été rangé dans la case 5 *)
- : int = 5

# find dico ch;;           (* On retrouve le couple associé à ch *)
- : int = 42

# ch.[0] <- 'h';;          (* Mutons à présent la chaîne ch *)
- : unit = ()

# find dico ch;;           (* On ne retrouve plus la clé *)
Exception: Not_found.

# dico;;                   (* Pourtant, elle demeure dans dico *)
- : (string, int) t =
        {table = [|[]; []; []; []; []; [("hello", 42)]; []; []|]}

# Hashtbl.hash ch mod 8;; (* Mais find la cherche à présent *)
- : int = 0                (* dans la case (hash ch) = 0     *)
```

Bref, une mutation de la clé nécessiterait de vérifier si le couple (clé, valeur) correspondant ne doit pas être déplacé dans la table de hachage !

Exercices

Ex. 4.1 – Piles d'assiettes

On considère un ensemble d'assiettes bleues et rouges, numérotées empilées dans un ordre quelconque. On souhaite réordonner les assiettes, de sorte que les bleues se situent en-dessous des rouges, mais sans changer la position relatives des assiettes bleues entre elles, ou des assiettes rouges entre elles.

Les assiettes sont des objets de type

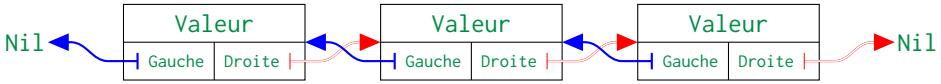
```
# type assiette = Bleue of int | Rouge of int;;
```

Proposer une fonction de signature `assiette Stack.t -> unit` prenant en argument une pile d'assiette et les réordonnant selon les critères proposés.

Ex. 4.2 – Doubles files

Il est parfois utile d'avoir des files à double sens : il s'agit de files particulières dans lesquelles on peut ajouter des éléments à la fois à l'extrémité gauche et à l'extrémité droite, et de même les retirer de chaque côté.

Pour représenter une telle double file, on peut utiliser des listes *doublement chaînées*, avec pour chaque élément dans la liste une indication de celui qui se trouve à gauche et à droite dans la file. En voici une représentation :



Le type d'une telle liste doublement chaînée serait par exemple :

```
# type 'a cell = { valeur : 'a ;
                   mutable gauche : 'a dclist ;
                   mutable droite : 'a dclist }

   and 'a dclist = Nil | Cell of 'a cell;;
```

Et celui de la double file :

```
# type 'a dequeue = { mutable extrGauche : 'a dclist;
                      mutable extrDroite : 'a dclist; }
```

Une double file vide, comme une file vide, voit ses deux extrémités pointer vers `Nil`.

Proposer des fonctions `addLeft`, `addRight`, `takeLeft` et `takeRight` qui ajoute et retirent des éléments dans la double file respectivement à gauche et à droite.

Ex. 4.3 – Nombres de Hamming

On rappelle que les nombres de Hamming sont les entiers strictement positifs dont la décomposition en facteurs premiers ne font intervenir que des 2, des 3 et des 5. Les vingt premiers nombres de Hamming sont donc

$$1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,\ldots$$

On souhaite écrire une fonction affichant les n premiers nombres de Hamming. Une méthode testant les entiers un à un avec la fonction écrite dans un exercice précédent serait inefficace (le 10000^{e} entier de Hamming est 288325195312500000 !)

On se propose donc d'utiliser la méthode suivante :

- on crée trois files f_2, f_3 et f_5, et on insère 1 dans chacune des files ;
- puis, n fois de suite :
  - on détermine les entiers n_2, n_3 et n_5 en tête de chacune des files ;
  - on détermine et affiche $n = \min(n_2, n_3, n_5)$;
  - on retire n des files f_2, f_3 et f_5 s'il s'y trouve en tête ;
  - on insère $2n$ dans f_2, $3n$ dans f_3 et $5n$ dans f_5.

1. Implémenter la fonction précédemment décrite.

2. Justifier qu'elle affiche bien les n plus petits nombres de Hamming par ordre croissant.

3. Une limite de la fonction précédente est que certains entiers peuvent se retrouver dans plusieurs files. Améliorer la fonction pour que cela n'arrive pas.

Ex. 4.4 – PGCD

On suppose que l'on dispose d'une fonction pgcd de signature `int -> int -> int` retournant le PGCD de deux entiers positifs passés en arguments.

On dispose d'une pile d'entiers non vide, et on souhaite calculer le PGCD de tous les entiers dans la pile.

On se propose d'utiliser l'algorithme suivant :

- tant que la pile contient au moins deux éléments [50], on en extrait deux entiers, on calcule leur PGCD, et on empile le résultat ;
- lorsque la pile ne contient plus qu'un élément, on le retourne.

1. Justifier que l'algorithme termine et retourne le résultat attendu.

2. Implémenter une fonction de signature `int Stack.t -> int` implémentant cet algorithme (lequel a pour effet de vider la pile passée en argument en plus de retourner le PGCD des entiers qu'elle contient).

3. Cet algorithme fonctionne-t-il également avec une file ?

50. On rappelle que l'on ne connait PAS la taille de la pile, on peut seulement savoir si elle est vide ou non.

5 Étude de la récursivité

1 Récursivité, terminaison, correction

1.1 Introduction

Une fonction *récursive*, en informatique[1] est une fonction qui, pour déterminer le résultat associé à certains paramètres, fait appel à elle-même. Nous avons eu l'occasion d'en croiser à plusieurs reprises déjà. Rappelons qu'il est nécessaire, en Caml, d'utiliser le mot-clé **rec** pour signaler au compilateur que la fonction va apparaître dans sa propre définition.

Par exemple, nous avons vu que la fonction factorielle, définie en mathématiques sur $\mathbb{N}$ à valeur dans $\mathbb{N}$ par

$$\text{Factorielle} : \begin{cases} 0 \longmapsto 1 \\ n \longmapsto n! = n \times (n-1)! \end{cases}$$

peut naturellement être écrite en Caml par la fonction récursive

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n-1);;
```

Dans ce chapitre, nous allons nous attacher, principalement, à montrer la *correction* et la *terminaison* de fonctions récursives.

Montrer la *correction* d'une fonction consiste à prouver que, si la fonction retourne un résultat, ce résultat est bien le résultat recherché. Par exemple que la fonction `fact` précédente retourne bien la factorielle de son argument.

Montrer la *terminaison* d'une fonction récursive pour un ensemble de paramètres consiste à prouver que la fonction retournera bien un résultat en un temps fini pour n'importe lequel de ces paramètres. Par exemple, on cherchera à montrer que la fonction `Fact` précédente retourne bien un résultat quel que soit l'entier positif qu'on lui fournit en paramètre.

1. En mathématiques, le terme de « fonction récursive » a un autre sens, lié à sa calculabilité.

En effet, toute fonction ne retourne pas un résultat en un temps fini. Prenons par exemple la fonction `Foo` retournant toujours 0, définie par

$$\text{foo} : \begin{cases} 0 \longmapsto 0 \\ n \longmapsto 2 \times f\left(\left\lceil \frac{n}{2} \right\rceil\right) \end{cases}$$

que l'on traduit par

```
let rec foo = function
  | 0 -> 0
  | n -> 2 * foo ((n+1)/2);;
```

Cette fonction ne s'arrête jamais[2], car foo(1) = 2 × foo(1), aussi est-il impossible de calculer foo(1) (et par conséquent, il en est de même pour tout entier positif).

Cette fonction peut néanmoins être « correcte » dans l'acception que l'on en donne dans ce cours (*chaque fois qu'elle retourne un résultat*, ce résultat est bien 0).

1.2 Lien entre terminaison et démonstration par récurrence

La preuve de la terminaison d'une fonction récursive est fortement liée aux démonstration par récurrence en mathématiques, même si dans ce dernier cas on a un raisonnement davantage « constructif ».

Par exemple, pour démontrer par récurrence qu'un ensemble de propriétés notées $\mathcal{P}(n)$, pour tout $n \in \mathbb{N}$, sont vraies, on montre généralement que la propriété est vraie pour $n = 0$ (initialisation), et que, pour tout $n \geqslant 0$, si elle est vraie pour n, alors elle est vraie également pour $n+1$ (récursion).

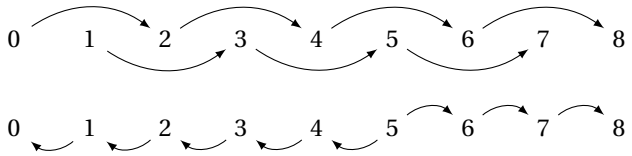
Ce n'est pas la seule possibilité. On pourrait montrer par exemple :

- qu'elle est vraie pour $n = 0$ et $n = 1$, et que, pour tout $n \geqslant 0$, si elle est vraie pour n et $n+1$, alors elle est vraie pour $n+2$;
- qu'elle est vraie pour $n = 0$, et que, pour tout $n \geqslant 0$, si elle est vraie pour tout $k \leqslant n$, alors elle est vraie pour $n+1$;
- qu'elle est vraie pour $n = 0$ et $n = 1$, et que, pour tout $n \geqslant 0$, si elle est vraie pour n, alors elle est vraie pour $n+2$;
- qu'elle est vraie pour $n = 5$, et que, pour tout $n \geqslant 5$, si elle est vraie pour n, alors elle est vraie pour $n+1$, et également que, pour $1 \leqslant n \leqslant 5$, si elle est vraie pour n, elle est vraie pour $n-1$...

On remarque qu'il y a toujours deux éléments indispensables : une initialisation, et une récurrence. Le point important est que l'on puisse atteindre n'importe quel n en partant de l'initilisation et en utilisant les récurrences.

2. Ou plutôt, ne s'arrêtera que lorsque l'ordinateur se trouvera à court de mémoire.

Par exemple, pour les deux derniers cas proposés, cela peut se comprendre aisément avec un schéma :



Sinon, il existe des valeurs de n pour lesquelles la propriété n'a pas été démontrée.

Il en est de même pour les fonctions récursives. Elles ont nécessairement deux composantes, une terminaison (des paramètres pour lesquels la fonction n'a pas à faire appel à elle-même) et une récursion.

Alors qu'une démonstration mathématique par récurrence suppose que l'on montre que toutes les propriétés peuvent être déduites à partir de l'initialisation en utilisant la récurrence, on cherchera à prouver que les appels récursifs finissent toujours par tomber dans le cas terminal (ou un des cas terminaux), quel que soit le paramètre initial fourni à la fonction.

1.3 Cas de la factorielle

Revenons sur le cas de la fonction factorielle. Il est assez simple de comprendre que `fact n` retourne un résultat en effectuant $n+1$ appels à la fonction `fact`. Le calcul de `fact n` exige le calcul de `fact (n-1)`, et ainsi de suite, jusqu'à parvenir au calcul de `fact 0` pour lequel on a un résultat sans qu'il soit nécessaire de faire un appel récursif.

Pour justifier ceci aisément, on peut simplement dire que les arguments des différents appels successifs sont des entiers positifs, constituant une suite strictement décroissante. Il ne peut y avoir de suite infinie strictement décroissante dans $\mathbb{N}$, donc on peut être assuré que la fonction terminera toujours.

Ce même argument ne fonctionne pas pour la fonction `foo` car on n'a pas toujours $\lceil \frac{n}{2} \rceil < n$, donc les arguments des appels successifs ne forment pas nécessairement une suite strictement décroissante dans $\mathbb{N}$ (et on a vu qu'effectivement, ce n'était pas le cas pour $n = 1$).

Outre la terminaison de la fonction `fact`, on peut s'interroger sur sa correction. On peut, pour cela, raisonner de façon similaire aux invariants de boucles. Pour un paramètre $n \geqslant 1$, la fonction associe `fact n` à `n * fact (n-1)`, ce qui est précisément la définition de la factorielle. Si la fonction retourne un résultat (ce qu'elle fait puisqu'elle termine), il sera correct, sous réserve que la multiplication ne déborde pas (ce qui arrive relativement vite en Caml, dès $n = 21$, car Caml utilise des entiers signés sur 63 bits).

1.4 Un autre exemple, le PGCD

Prenons un autre exemple, le calcul du PGCD de deux entiers positifs, que l'on définirait en Caml par la fonction

```
let rec pgcd a b = match a with
  | 0 -> b
  | _ -> pgcd (b mod a) a;;
```

Regardons la succession des appels qui sont effectués :

```
pgcd 42 24 -> pgcd 24 42 -> pgcd 18 24 -> pgcd 6 18 -> pgcd 0 6 -> 6
```

Comment justifier que la fonction termine toujours pour `a` et `b` positifs ? On peut remarquer que, lorsque `b` n'est pas nul, on a un appel récursif dont le premier paramètre est `b mod a`. Or, pour $b \geqslant 0$ et $a > 0$, on a `0 b mod a < a` !

Autrement dit, lors des appels successifs, le premier paramètre représente une suite strictement décroissante dans $\mathbb{N}$, donc finira toujours par atteindre 0 (et donc le cas de terminaison), ce qui permettra de terminer la fonction.

Pour prouver la correction de la fonction, il suffit de justifier que `pgcd a b` est bien égal, pour tout a et b positifs, à `pgcd (b mod a) a`, ce qui est le cas.

2 Aller un peu plus loin

2.1 Un autre exemple

Prenons un exemple qui a beaucoup fait cogiter de nombreux penseurs, et ce dès l'antiquité, un des paradoxes de Héron. Achille était réputé pour être l'un des plus grands athlètes de l'antiquité. Héron, dans une expérience de pensée, l'oppose à une tortue dans une épreuve de course à pied.

La tortue n'étant pas bien rapide, Achille a la bonté de lui laisser cent mètres d'avance. La question est de savoir si Achille rattrapera la tortue (et au bout de combien de temps), ou si cette dernière restera indéfiniment en tête.

Bien évidemment, après un certain temps, Achille aura parcouru une distance de 100 m. En supposant qu'il court à la vitesse de $v_a = 10$ m/s, il lui faudra 10 s. Cependant, pendant ces dix secondes, la tortue aura elle aussi avancé, et se trouvera un peu plus loin, devançant toujours Achille. En supposant que la tortue (très rapide) avance à une vitesse $v_t = 0{,}1$ m/s, honorable mais cent fois moindre ($v_t / v_a = 0.01$), elle aura encore un mètre d'avance. Un mètre qu'Achille comblera en un maigre dixième de seconde, mais la tortue avançant toujours, elle est toujours en tête. Et ainsi de suite.

En combien d'étapes Achille rattrapera-t-elle la tortue ? Il en faut évidemment une infinité, bien que l'on ne s'attend pas à une victoire de la tortue pour autant, ce qui en faisait un paradoxe intéressant à étudier (qui se résoud en remarquant qu'une somme infinie de temps de parcours peut très bien donner un résultat fini, ce qui n'étonnera guère le lecteur).

Revenons à nos problèmes de récursivité. On peut écrire une fonction qui calcule le temps nécessaire à Achille pour dépasser la tortue, lorsque l'on passe en paramètre l'avance d de la tortue :

```
let rec temps = function
  | d when d <= 0. -> 0.
  | d              -> d /. 10. +. temps (0.01 *. d);;
```

Pour le problème qui nous intéresse, on l'appellera avec `temps 100.0`.

Cette fonction récursive termine-t-elle quels que soient les arguments qui lui sont fournis ? Nous allons voir que la réponse est loin d'être simple.

Prenons le problème d'un point de vue mathématique : l'avance d de la tortue, tant qu'elle est positive, est multipliée par $v_t / v_a = 0.01$ à chaque appel. On a donc ici une suite de réels positifs, décroissante. Seulement, d'un point de vue *mathématique*, on n'atteindra jamais zéro [3], comme c'était le cas pour nos deux exemples précédents, même si la limite se trouve bien être zéro. Il y a donc une différence fondamentale entre des suites strictement décroissantes sur $\mathbb{N}$ et celles strictement décroissantes sur $\mathbb{R}^+$.

2.2 Formaliser le problème

Pour formaliser un peu les choses, considérons une fonction récursive $f : \mathrm{E} \mapsto \mathrm{F}$, prenant un « argument » x parmi un ensemble E d'arguments possibles [4], et retournant un résultat dans F.

Parmi les éléments $x \in \mathrm{E}$, il existe un sous-ensemble A d'arguments pour lesquels la fonction retourne un résultat immédiatement. Pour les arguments de $\mathrm{E} \setminus \mathrm{A}$, la fonction opère nombre fini d'appels récursifs [5].

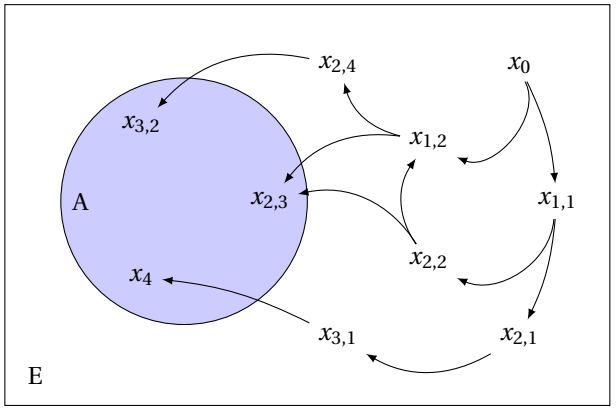
Pour que le problème soit intéressant, A et $\mathrm{E} \setminus \mathrm{A}$ ne sont pas vides (si $\mathrm{E} \setminus \mathrm{A}$ est vide, la fonction, non récursive, termine toujours, si A est vide, elle ne peut pas terminer).

3. Peut-être aurez-vous dans l'idée que le problème n'est pas si simple avec des flottants, et vous avez parfaitement raison, nous y reviendrons un peu plus tard.

4. On prendra ici « argument » dans un sens très général, x pouvant être un simple entier, une liste, un ensemble d'une demi-douzaine d'éléments de types variés, ou quelque entrée que ce soit que l'on puisse imaginer à une fonction ; ainsi f peut désigner n'importe quelle fonction.

5. Si le nombre d'appels récursifs est infini pour certaines valeurs du paramètre x, la question de la terminaison est d'ores et déjà réglée !

Prenons un élément x_0 de $\mathrm{E} \setminus \mathrm{A}$. Le calcul de $f(x_0)$ nécessite le calcul d'une ou plusieurs expressions $f(x_{1,i})$. Certains des $x_{1,i}$ seront peut-être des éléments de A, mais d'autres nécessiteront possiblement à leur tour le calcul de $f(x_{2,j})$, et ainsi de suite. Par exemple, les appels récursifs peuvent se succéder comme dans le schéma ci-dessous :



Prouver qu'une fonction récursive termine toujours consiste à montrer qu'il n'y a pas, dans cet ensemble d'appels récursifs, une séquence infinie d'appels n'aboutissant jamais dans A (une véritable séquence infinie, ou bien un cycle, ce qui revient au même), et ce quel que soit le point de départ x_0 dans E. Bref, on se pose la question de savoir si « tous les chemins mènent à A » !

Une des solutions pour parvenir à prouver la terminaison consiste à montrer qu'à chaque étape, on se « rapproche » de A, et que la « longueur » du chemin menant à A, pour chacun des éléments x diminue à chaque appel.

Cependant, le fait qu'on s'approche à chaque étape n'est pas suffisant : Achille ne rattrapera pas la tortue en un nombre fini d'étapes, même s'il s'en approche à chacune des étapes.

2.3 Principe d'induction

Définition. Considérons un ensemble $\mathcal{E}$, muni d'une relation d'ordre, notée $\leqslant$. Si $\mathcal{A}$ est une partie non-vide de $\mathcal{E}$, et $a \in \mathcal{A}$, on dit que

- a est *un élément minimal* de $\mathcal{A}$ lorsque, $\forall x \in \mathcal{A}, x \leqslant a \Rightarrow x = a$;
- a est *le plus petit élément* de $\mathcal{A}$ lorsque, $\forall x \in \mathcal{A}, a \leqslant x$.

Dans une partie, il peut y avoir plusieurs éléments minimaux (lorsque l'ordre n'est pas total), mais le plus petit élément, s'il existe, est nécessairement unique. De façon évidente, s'il existe un plus petit élément, il est nécessairement aussi un élément minimal (la réciproque n'étant vraie que pour un ordre total).

Définition. $(\mathcal{E}, \leqslant)$ est qualifié d'ensemble *bien fondé* lorsque toute partie non-vide de E possède au moins un élément minimal. $(\mathcal{E}, \leqslant)$ est qualidié d'ensemble *bien ordonné* lorsque toute partie non-vide de $\mathcal{E}$ possède un plus petit élément. Bien entendu, un ensemble bien ordonné est nécessairement bien fondé.

L'ensemble $\mathbb{N}$ muni de l'ordre usuel est bien ordonné et bien fondé. En revanche, $\mathbb{R}^+$ n'est ni bien ordonné, ni bien fondé : en effet, des sous-ensembles comme $\mathbb{R}^{+\star}$, ou $]0,1]$, n'ont ni élément minimal, ni plus petit élément. C'est cette différence qui est la clé.

Théorème 2 (Principe d'induction)**.** *Soit $(\mathcal{E}, \leqslant)$ un ensemble bien fondé, $\mathcal{M}$ l'ensemble de ses éléments minimaux. Si un prédicat $\mathcal{P}$ sur $\mathcal{E}$ vérifie*

- *pour tout $x \in \mathcal{M}$, $\mathcal{P}(x)$ est vrai;*
- *pour tout $x \in \mathcal{E} \setminus \mathcal{M}$, $(\forall y < x, \mathcal{P}(y)) \Rightarrow \mathcal{P}(x)$,*

alors $\mathcal{P}(x)$ est vrai pour tout $x \in \mathcal{E}$.

Démonstration. On peut démontrer le principe d'induction en raisonnant par l'absurde, en supposant qu'il existe un ensemble X d'éléments de $\mathcal{E}$ pour lesquels $\mathcal{P}(x)$ est faux. Cet ensemble admet un élément minimal x_0, qui est nécessairement un élément de $\mathcal{E} \setminus \mathcal{M}$. Seulement, tout $y \in \mathcal{E}$ vérifiant $y < x_0$, $\mathcal{P}(y)$ est vérifié (puisque x_0 est un élément minimal de X), ce qui implique que $\mathcal{P}(x_0)$ est vérifié, soit une contradiction avec l'hypothèse initiale, prouvant le principe. □

Le principe d'induction est une généralisation du principe de récurrence. Il va nous permettre de vérifier que notre fonction récursive $f : \mathrm{E} \mapsto \mathrm{F}$ termine pour tout $x \in \mathrm{E}$.

Théorème 3. *Soit f une fonction récursive d'un ensemble* E *vers un ensemble* F*, et ϕ une fonction de* E *vers un ensemble bien fondé $(\mathcal{E}, \leqslant)$ telle que, pour tout $x \in \mathrm{E}$,*

- *soit la fonction f retourne un résultat directement;*
- *soit la fonction calcule le résultat en utilisant un nombre fini d'appels récursifs dont les arguments y_i vérifient tous $\phi(y_i) < \phi(x)$.*

La fonction f termine pour tous les éléments de E*.*

Démonstration. Il suffit d'utiliser le principe d'induction, en considérant pour tout $e \in \mathcal{E}$ le prédictat $\mathcal{P}(e)$ « Pour tout argument $x \in \mathrm{E}$ vérifiant $\phi(x) = e$, la fonction récursive f termine ». Les propriétés de ϕ permettent de vérifier la seconde condition du principe d'induction. Pour la première condition, il suffit de voir que les éléments $x \in \mathrm{E}$ pour lesquels $\phi(x)$ est un élément minimal de $\mathcal{E}$ n'effectuent jamais aucun appel récursif[a]. L'application du principe d'induction garantit donc bien que la fonction f termine bien pour tout $x \in \mathrm{E}$. □

a. Car ils devraient utiliser des arguments y pour lesquels $\phi(y) < \phi(x)$, ce qui est en contradiction avec le fait que $\phi(x)$ est un élément minimal de $\mathcal{E}$.

2.4 Exemples

Il n'est pas absolument nécessaire de maîtriser le formalisme précédent pour démontrer la terminaison d'une fonction récursive.

On peut en effet justifier la terminaison d'une fonction en deux points :

- exhiber une suite *strictement décroissante*[6] construite à partir des paramètres;
- justifier que cette suite ne peut être infinie (par exemple en remarquant que les termes de la suite sont des éléments d'un ensemble bien fondé pour l'ordre considéré[7]).

Comme on l'a vu, le cas le plus simple est celui où on peut travailler avec des entiers naturels et la relation d'ordre usuelle. Idéalement, les arguments sont des entiers naturels et forment eux-même une suite strictement décroissante[8].

Par exemple, dans le cas de la fonction factorielle, $\mathbb{N}$ muni de l'ordre usuel est un ensemble bien fondé, et lors de l'unique appel récursif, on passe d'un paramètre n à un paramètre `n-1` strictement inférieur. La fonction `fact` termine donc pour tout entier naturel.

Pour la fonction calculant le PGCD,

```
let rec pgcd a b = match a with
  | 0 -> b
  | _ -> pgcd (b mod a) a;;
```

une solution consiste à considérer la fonction $\phi : \mathbb{N}^2 \mapsto \mathbb{N}$ qui au couple (a, b) associe a.

Comme on l'a montré tantôt, $\phi(a \mod b, a) < \phi(a, b)$, donc comme $(\mathbb{N}, \leqslant)$ est bien fondé, cela suffit à prouver la terminaison de la fonction `pgcd` pour tout couple d'entiers positifs.

Prenons de même la fonction `comb` calculant le coefficient binomial $\binom{n}{k}$ définie par

```
let rec comb k n =
  if k < 0 || k > n then 0 else
  if k = 0 || k = n then 1 else
  comb (k-1) (n-1) + comb k (n-1);;
```

et intéressons-nous à sa terminaison pour k et n entier naturels.

Cette fois-ci, pour calculer `comb k n`, on a deux appels récursifs, mais le second paramètre de ces deux appels est toujours strictement plus petit que `n`. En prenant la fonction

6. Pour un ordre quelconque, pas nécessairement l'ordre usuel.
7. Il est aisé demontrer que la non-existence de suite strictement décroissante dans un ensemble ordonné est équivalente au caractère bien fondé de cet ensemble.
8. On prend alors pour fonction ϕ la fonction identité

$\phi : \mathbb{N}^2 \mapsto \mathbb{N}$ qui au couple (a,b) associe b, on peut justifier que la fonction précédente termine pour tout couple d'entiers positifs (lorsque le paramètre `n` est nul, la fonction termine bien quelle que soit la valeur du paramètre `k`).

La correction de la fonction précédente est également facile à justifier, puisque la récursion est basée sur la relation

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Signalons cependant que, si la fonction précédente termine pour tout couple d'entiers positifs, et donne un résultat correct, elle n'en est pas moins très maladroite.

En effet, le nombre d'appels devient rapidement prohibitif : le premier appel de la fonction `Comb` provoque potentiellement deux appels, qui en provoquent à leur tour jusque quatre, et ainsi de suite. Le temps de calcul devient rapidement déraisonnable, (quoique nous verrons prochainement une façon d'y remédier en partie).

Ce n'est pas parce que l'on a réussi à justifier qu'une fonction termine et retourne le résultat correct qu'elle sera utilisable en pratique. Encore faut-il qu'elle retourne le résultat en un temps raisonnable, ce qui est un tout autre problème.

Dans l'exemple précédent, on peut aisément montrer par une récurrence que la fonction est appelée, au total, $2\binom{n}{k} - 1$ fois, ce qui est vite gigantesque même pour des valeurs assez modestes de k et n.

On préférera nettement cette version de la fonction `comb` (elle termine d'après un raisonnement similaire au précédent), qui conduit au plus à $k+1$ appels de la fonction :

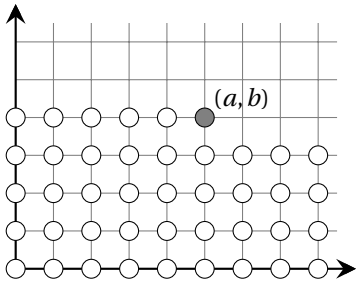
```
let rec comb k n =
  if k < 0 || k > n then 0 else
  if k = 0 || k = n then 1 else
  Comb (k-1) (n-1) * n / k;;
```

2.5 Autres ordres utiles

Plutôt que de se ramener à des éléments de $\mathbb{N}$, il est parfois plus simple de travailler directement avec un ensemble $\mathcal{E} = \mathbb{N}^p$ avec $p > 1$. Pour ce faire, on dispose de plusieurs ordres potentiellement utiles.

Ordre lexicographique $\mathbb{N}^2$, lorsqu'il est muni de l'ordre lexicographique $\preceq_\ell$ (que l'on définit par $(a,b) \preceq_\ell (a',b') \Leftrightarrow a < a'$ ou $a = a'$ et $b \leqslant b'$) est un ensemble bien ordonné (et donc également bien fondé).

Par exemple, le graphe ci-dessous montre le couple (a,b) où $a = 4$ et $b = 5$ (le premier élément du couple est placé en ordonnée), et l'ensemble des couples (a',b') qui sont inférieurs à (a,b) pour l'ordre lexicographique $\preceq_\ell$:

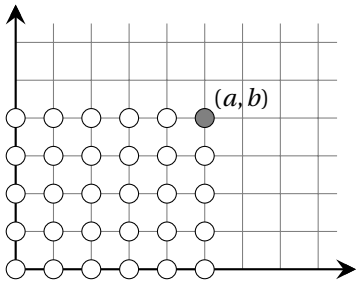


Il est intéressant de remarquer ici qu'il existe une infinité d'éléments $(a',b') \in \mathbb{N}^2$ strictement plus petits qu'un (a,b) donné. Et pourtant, il n'existe pas de suite infinie $(a_i,b_i)_{i\in\mathbb{N}}$ dans $\mathbb{N}^2$ strictement décroissante pour $\preceq_\ell$!

Cela peut se comprendre en remarquant qu'à chaque itération, soit $a_{i+1} < a_i$, soit $a_{i+1} = a_i$ et $b_{i+1} < b_i$ diminue. Le premier cas ne peut pas arriver plus de a_0 fois. Par conséquent, s'il existait une suite infinie, le second cas devrait se succéder une infinité de fois pour une valeur donnée de a_i, ce qui n'est pas possible car on exhiberait une suite de b_i infinie strictement décroissante dans $\mathbb{N}$!

L'ordre lexicographique peut aisément être étendu de $\mathbb{N}^2$ à $\mathbb{N}^p$ pour $p \in \mathbb{N}^*$ quelconque.

Ordre produit De même, l'ensemble $\mathbb{N}^2$, muni de l'ordre produit $\preceq_\times$ (que l'on défini par $(a,b) \preceq_\times (a',b') \Leftrightarrow a \leqslant a'$ et $b \leqslant b'$), n'est pas bien ordonné (l'ordre n'est pas total) mais est néanmoins bien fondé. Les éléments inférieurs à (a,b) sont représentés ci-dessous :

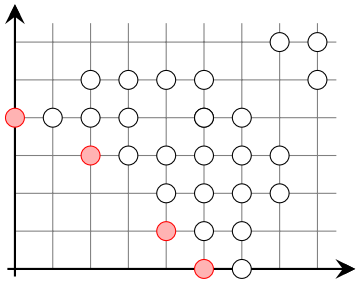


Il n'existe dont pas de suite infinie strictement décroissante dans $\mathbb{N}^2$ pour $\preceq_\times$ [9].

Notons que, l'ordre produit n'étant pas total (on n'a ni $(2,5) \preceq_\times (5,2)$, ni $(5,2) \preceq_\times (2,5)$ par exemple), il existe des parties de $\mathbb{N}^2$ sans plus petit élément, comme celle ci-dessous.

9. Comme cette fois, il existe un nombre fini – exactement $(a+1)\times(b+1)$ – d'éléments de $\mathbb{N}^2$ plus petits que (a,b) pour $\preceq_\times$, cela n'est guère surprenant.

En revanche, toute partie de $\mathbb{N}^2$ a bien des éléments minimaux (que l'on a mis en évidence en couleur ci-dessous) :



L'ordre produit peut également être étendu à $\mathbb{N}^p$ pour p quelconque.

Exemples d'utilisation Dans le cas de pgcd, on peut donc simplement choisir l'identité pour la fonction ϕ, et remarquer que $(b \mod a, a) \prec_\ell (a, b)$ pour l'ordre lexicographique. Ce qui permet directement de conclure sur la terminaison de la fonction pgcd.

L'ordre produit, en revanche, ne convient pas, car la première étape de la fonction pgcd peut échanger les arguments si initialement $a > b$.

Pour la fonction comb, $(k-1, n-1) \prec (k, n)$ et $(k, n-1) \prec (k, n)$ à la fois pour l'ordre lexicographique $\prec_\ell$ et pour l'ordre produit $\prec_\times$, donc les deux ordres conviennent.

2.6 Arguments complexes (listes, arbres, chaînes...)

Lorsque les arguments sont un peu plus complexes (listes, arbres), on peut utiliser ϕ pour les ramener à des entiers positifs en prenant une de leurs caractéristiques bien choisies (longueur de la liste, hauteur de l'arbre...).

Par exemple, dans cette fonction calculant la somme des termes d'une liste d'entiers,

```
let rec somme = function
  | []   -> 0
  | t::q -> t + somme q;;
```

la longueur de la liste passée en argument est un entier positif qui décroit strictement lors de chaque appel récursif, donc la fonction termine pour toute liste.

De même, dans la fonction calculant la hauteur d'un arbre,

```
let rec hauteur = function
  | Feuille              -> 0
  | Noeud (filsg, filsd) -> 1 + max (hauteur filsg) (hauteur filsd);;
```

ou bien dans celle calculant sa taille,

```
let rec taille = function
  | Feuille              -> 1
  | Noeud (filsg, filsd) -> 1 + taille filsg + taille filsd;;
```

les hauteurs des deux sous-arbres utilisés comme arguments des appels récursifs sont strictement inférieures à celle de l'arbre argument de la fonction, donc ces fonctions `hauteur` et `taille` terminent pour tout arbre.

Pour une chaîne de caractères, on peut utiliser sa longueur [10], comme dans cette fonction qui termine puisque la longueur de la chaîne passée en argument décroit de 2 à chaque appel récursif :

```
let rec estPalindrome chaine =
  let n = String.length chaine in
    n < 2 || chaine.[0] = chaine.[n-1]
     && estPalindrome (String.sub chaine 1 (n-2));;
```

Précisons que la fonction précédente effectue des copies coûteuses de la chaîne lors des appels à `String.sub`, et inutiles : une meilleure solution consisterait à utiliser des indices délimitant la partie de la chaîne qu'il reste à vérifier. Ces indices étant entiers, et le nombre de caractères restant à vérifier strictement décroissant, la terminaison d'une telle fonction n'est pas difficile à prouver.

2.7 Problème de la terminaison

Malheureusement, il n'est pas toujours aussi simple de montrer qu'une fonction récursive termine. Par exemple, la fonction Q de Hofstadter :

```
let rec q = function
  | 1 -> 1
  | 2 -> 1
  | n -> q (n - q (n-1)) + q (n - q (n-2));;
```

ou bien cette fonction, basée sur la suite de Syracuse :

```
let rec syracuse = function
  | 1                    -> true
  | n when (n mod 2) = 0 -> syracuse (n/2)
  | n                    -> syracuse (3*n+1);;
```

10. Précisons qu'il existe des relations d'ordre bien fondés sur l'ensemble des chaînes de caractères, dont certaines seront étudiées dans le programme de seconde année.

sont deux exemples de fonctions dont on a conjecturé qu'elles terminaient toujours (on n'a pas trouvé de n pour lesquels ce ne serait pas le cas) mais ces conjectures n'ont pas encore pu être démontrées à l'heure actuelle.

Par exemple, dans le cas de la fonction de Hofstadter, on a bien $n-1<n$, $n-2<n$, $n-q(n-1)<n$ et $n-q(n-2)<n$ puisque la fonction ne peut visiblement retourner que des éléments de $\mathbb{N}^\star$, *mais* il reste à prouver que $n-q(n-1)\geqslant 1$ et $n-q(n-2)\geqslant 1$ pour justifier que la fonction n'effectue des appels récursifs qu'avec des éléments de $\mathbb{N}^\star$.

En fait, il a été montré par A. Turing (avant même que les ordinateurs modernes existent) que la question de la terminaison d'une fonction (récursive ou non) était un problème indécidable, c'est-à-dire qu'on ne peut écrire de fonction « `termine` » qui prenne une fonction en argument et retourne `true` si cette fonction termine pour tous les arguments, et `false` dans le cas contraire. Il est aisé de s'en convaincre en considérant la fonction suivante :

```
let rec foo () = match termine foo with
  | true  -> foo ()
  | false -> true;;
```

En effet, si `foo` termine, alors `foo ()` fait un appel récursif à `foo ()`, donc elle ne termine pas. Et si elle ne termine pas, alors elle retourne `true`, donc elle termine. La seule façon de lever ce paradoxe est que `termine` ne peut pas fonctionner correctement avec `foo` comme argument.

2.8 Réels vs flottants

Revenons à notre course entre Achille et la tortue. Si l'on travaillait sur $\mathbb{R}$, le fait que $(\mathbb{R}^+,\leqslant)$ ne soit ni bien ordonné, ni bien fondé, nous empêche de conclure à la terminaison de l'algorithme. Et, de fait, dans $\mathbb{R}^+$, il ne terminerait pas.

Pourtant, la fonction retourne bien un résultat après un nombre fini d'appels (164 très exactement) :

```
# temps 100.0;;

- : float = 10.101010101
```

La raison est que l'on travaille avec des *flottants* et non des réels. La quasi-totalité des grandeurs manipulées en informatique utilisent un nombre prédéterminé de bits (souvent 64 bits pour des flottants), et il y en a donc un nombre *fini* de valeurs possibles. L'ensemble des entiers relatifs sur 32 bits ou l'ensemble des flottants 64 bits positifs sont bien fondés lorsqu'ils sont munis de la relation d'ordre habituel (puisque tout sous-ensemble fini muni d'un ordre total aura nécessairement un élément minimal), même si $\mathbb{Z}$ et $\mathbb{R}^+$ ne le sont pas.

Excepté lorsque l'on travaille avec des données dont la taille peut varier (des listes, par exemple) [11], seule la question de la relation d'ordre importe réellement.

Ce pourrait être une excellente nouvelle, mais toute médaille ayant son revers, il faut se souvenir que les calculs sur les flottants ne se comporent pas tout à fait comme ceux sur les réels. Dans le cas qui nous intéresse, pour un x flottant positif, on a bien $0.01\times x<x$.

Seulement, $x>0$ n'implique pas $0.01\times x>0$, car $0.01\times x$ peut être nul. C'est ce qui permet à notre fonction récursive de se terminer après 164 appels. Le plus petit flottant est de l'ordre de 10^{-324}, or $100\times(0.01)^{164}$ est plus petit que cela, donc assimilé à zéro [12].

À l'inverse, on n'a pas $0.6\times x<x$, car il est possible que $0.6\times x=x$ si x est suffisamment petit ($0.6\times x$ étant arrondi à x). De sorte que si la tortue, dopée au stéroïdes, a une vitesse de 0,6 fois celle d'Achille, la fonction

```
let rec temps = function
  | 0. -> 0.
  | d  -> d /. 10. +. (temps (0.6 *. d));;
```

elle, ne termine jamais [13] [14] !

Ces problèmes d'arrondis peuvent également poser des problèmes lorsqu'il s'agit de prouver la correction de la fonction.

Il est bien plus facile de travailler avec des entiers, les preuves concernant les fonctions faisant intervenir des flottants sont souvent très délicates, et encore souvent un sujet de recherche.

3 Récursion terminale

3.1 Le mécanisme d'appel de fonction

Cette dernière partie de ce chapitre vise à examiner un peu plus en détail comment un ordinateur gère les appels de fonctions. Dans l'optique des concours, il n'est nul besoin d'être un expert sur le sujet, et on cherche avant tout ici à éclairer pourquoi certaines écritures de fonctions récursives fonctionnent mieux que d'autres (et sont donc plus fréquemment utilisées).

11. C'est aussi le cas des entiers relatifs en Python, dont la représentation peut utiliser un nombre quelconque de bits.
12. Pour les règles usuelles d'arrondi, lesquelles peuvent être changées mais nous n'entrerons pas dans les détails.
13. Ici encore, pour les règles usuelles d'arrondis sur les flottants.
14. La fonction ne termine pas, mais elle finira cependant par provoquer une erreur, plus précisément un débordement de pile, car les appels récursifs consomment ici de la mémoire. Nous allons y revenir dans la section suivante.

Dans un programme, un appel de fonction « suspend » un temps l'exécution d'une séquence d'instructions, le temps d'exécuter les instructions constituant la fonction appelée.

Dans un programme compilé, par exemple, la séquence d'instructions de la fonction se trouve à un endroit différent dans la mémoire de la séquence d'instructions où survient l'appel à ladite fonction. Lors d'un appel de fonction, le processeur doit noter où il se trouvait juste avant l'appel à la fonction avant de s'intéresser au instructions constituant la fonction, histoire de pouvoir y revenir ultérieurement.

Tout se passe comme si vous lisez un livre, qu'un paragraphe fait référence à une note en fin d'ouvrage, et que vous laissez un marque-page le temps d'aller lire la note avant de reprendre votre lecture à l'emplacement du marque-page.

Seulement, dans un programme, une fonction peut appeler une seconde fonction, qui peut faire appel à une troisième fonction, et ainsi de suite. Les fonctions peuvent même s'appeler elles-mêmes, ce qui se trouve d'ailleurs être le sujet de ce chapitre. Il faut donc mémoriser plusieurs « adresses de retour ». Comme on reviendra aux différentes tâches dans l'ordre inverse où on les a laissées, il est naturel d'utiliser une pile pour cet usage.

De fait, tous les ordinateurs disposent d'une *pile d'appel* dans laquelle on mémorise les adresses de retour à chaque appel de fonction, de façon à pouvoir reprendre l'exécution normale du programme lorsque l'on termine la fonction.

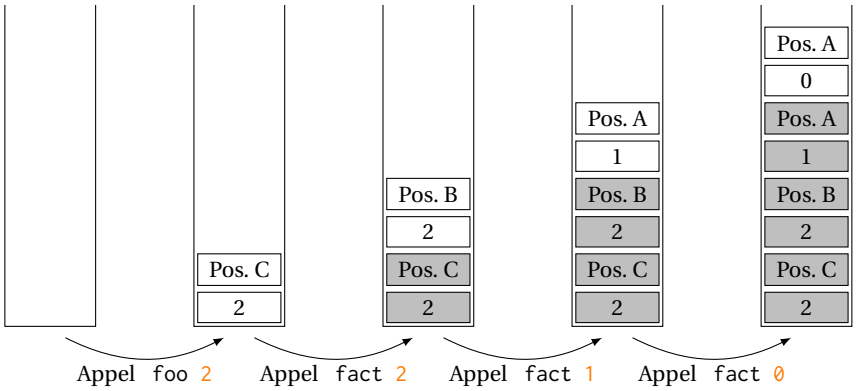
Ce n'est pas la seule chose que l'on place dans cette pile. Les arguments de la fonction y sont généralement également placés, de sorte que la fonction qui est appelée puisse les retrouver aisément. Et dans le cas d'une fonction récursive, que les différents appels ne mélangent pas leurs arguments respectifs ! Les définitions locales [15] y sont généralement également placées, afin que chaque fonction puisse accéder à ses propres définitions.

Prenons l'exemple du programme suivant, qui calcule $2n!$:

```
let rec fact = function
  | 0 -> 1
  | n ->
    fact (n-1)     (* position A *)
    * n;;

let foo n =
  fact n           (* position B *)
  * 2;;

foo 2;;            (* position C *)
```

L'instruction `foo 2` fait appel à `foo` avec 2 en paramètre. Laquelle fait appel à `fact` avec 2 en paramètre. Qui à son tour fait appel à `fact` avec 1 en paramètre. Et enfin, un dernier appel à `fact` avec 0 en paramètre.

Au niveau de la pile, les choses se passent de la façon suivante :



Comme on le voit, à chaque appel de fonction, Caml empile le paramètre de la fonction, puis juste avant de « sauter » à la fonction appelée, l'adresse [16] à laquelle il devra revenir lorsqu'il en aura terminé avec la fonction.

Chaque fonction n'a besoin d'accéder qu'à la partie de la pile qui lui correspond (en blanc dans l'exemple du dessus). On parle parfois de « trame de pile » (*stack frame* en anglais). Sous l'adresse de retour se trouvent le ou les paramètres qui lui ont été passés en arguments [17].

Le dernier appel à `fact` ne cause pas de nouvel appel, et la fonction retourne simplement le résultat 1. Le programme Caml va donc alors dépiler l'adresse de retour (de même que l'argument, qui ne sera plus utile et sera jeté), et retourne à la fonction appelante (qui ici est `fact` également). La fonction appelante récupère le résultat (ici 1) et le multiplie par son argument (1 encore), et retourne comme résultat le produit. On dépile alors à nouveau une trame de pile. Et ainsi de suite.

Le résultat retourné par la fonction pourrait être placé dans la pile [18], mais actuellement, en général, il est plutôt laissé dans un registre du processeur [19], une mémoire interne au processeur destinée aux calculs, et particulièrement rapide, ce qui permettra d'utiliser directement le résultat dans la suite du programme sans perdre de temps.

15. Ou les variables locales dans d'autres langages

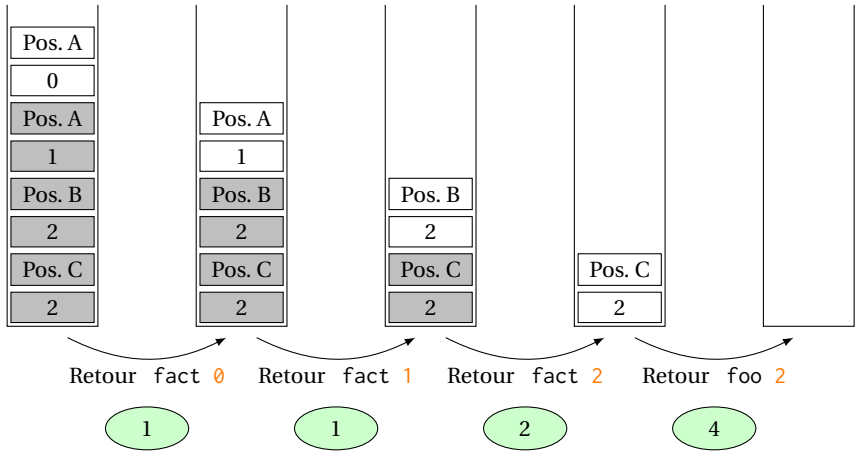
16. Désignée ici par « Pos. A », « Pos. B » et « Pos. C », mais correspondant pour le processeur en réalité à l'emplacement, dans la mémoire, de l'instruction suivant celle qui effectue l'appel.

17. Précisons que l'ordre des éléments dans la trame de pile est une convention, la seule chose qui importe est que la fonction appelante et la fonction appelée s'entendent. Il est néanmoins fréquent que les paramètres se trouvent « en-dessous » de l'adresse de retour dans la pile, car l'adresse de retour est empilée au moment du saut d'une fonction à l'autre, autrement dit au tout dernier moment.

18. C'est ici aussi un choix laissé au compilateur, qui dépend des possibilités du processeur utilisé.

19. Ou bien, s'il est trop volumineux pour y tenir, la fonction laisse dans le registre une adresse mémoire indiquant où trouver le résultat.

Au niveau de la pile d'appel et du registre utilisé pour les résultats, la suite du programme se déroule donc de la sorte :



3.2 Récursion vs boucles

Chaque fois que l'on effectue un appel de fonction, il y a donc un coût au niveau du processeur : il faut empiler les paramètres, puis l'adresse de retour, avant de « sauter » en un autre point du programme. Puis, un peu plus tard, on dépilera les données et on reviendra à la fonction appelante.

Parfois, il est nécessaire d'effectuer des opérations supplémentaires. Par exemple, beaucoup de fonctions utilisent les registres du processeur. On peut avoir besoin de mémoriser le contenu de ces registres avant de faire appel à une fonction, afin de pouvoir les restaurer lorsque l'appel sera terminé, dans le cas où la fonction appelée aurait modifié le contenu des registres. On parle alors de *sauvegarde (et de restauration) du contexte*.

Les appels récursifs ont donc un coût, modéré (quelques dizaines de cycles, soit quelques nanosecondes sur un ordinateur moderne) mais non nul. Si une écriture récursive d'une fonction est parfois plus lisible ou plus simple, elle peut être un peu plus lente qu'une version écrite au moyen d'une boucle.

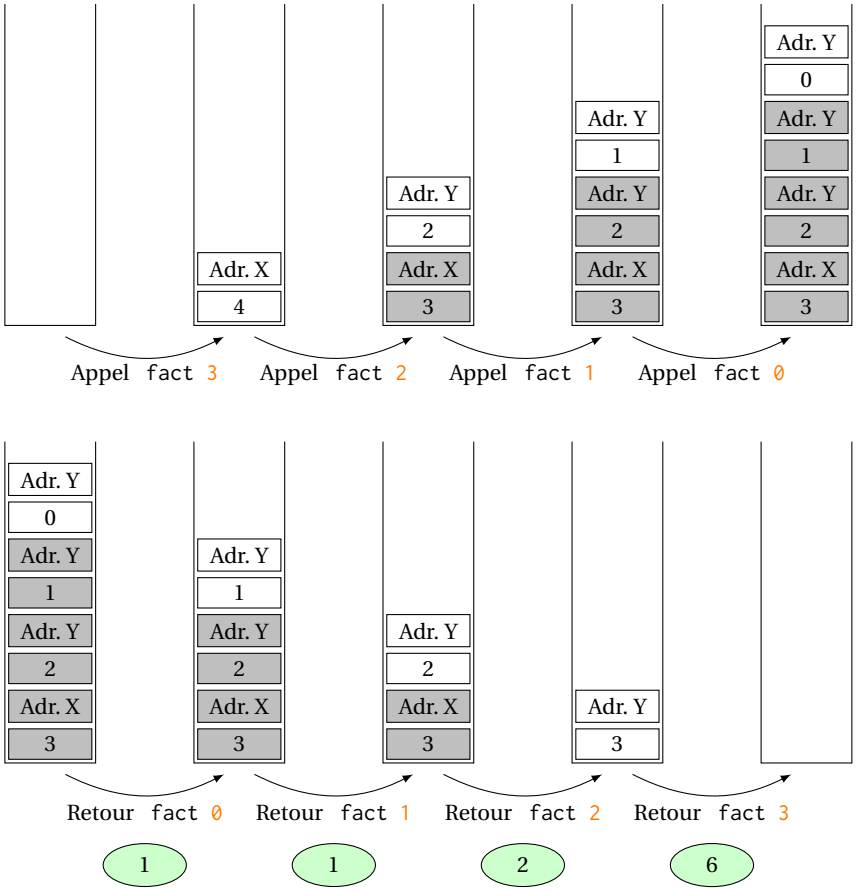
On passe cependant souvent plus de temps à écrire et modifier les programmes qu'à les utiliser, donc gagner quelques nanosecondes n'a cependant de sens que si la fonction est réellement appelée très, très souvent, surtout si c'est au prix d'une complication importante de la fonction.

L'autre difficulté que l'on peut rencontrer lorsque l'on effectue de nombreux appels de fonction est lié à la mémoire : la pile d'appels décrite ci-dessus occupe de la place en mémoire (dans une zone qui lui est réservée). Lorsque l'on effectue trop d'appels récursifs, on peut se retrouver à court de mémoire, ce qui interrompt le programme! C'est par exemple le cas sur cette fonction qui tente d'effectuer une infinité d'appels récursifs :

```
# let rec foo () = 1 + foo ();;
val foo : unit -> int = <fun>

# foo ();;
Stack overflow during evaluation (looping recursion?).
```

3.3 Récursion terminale

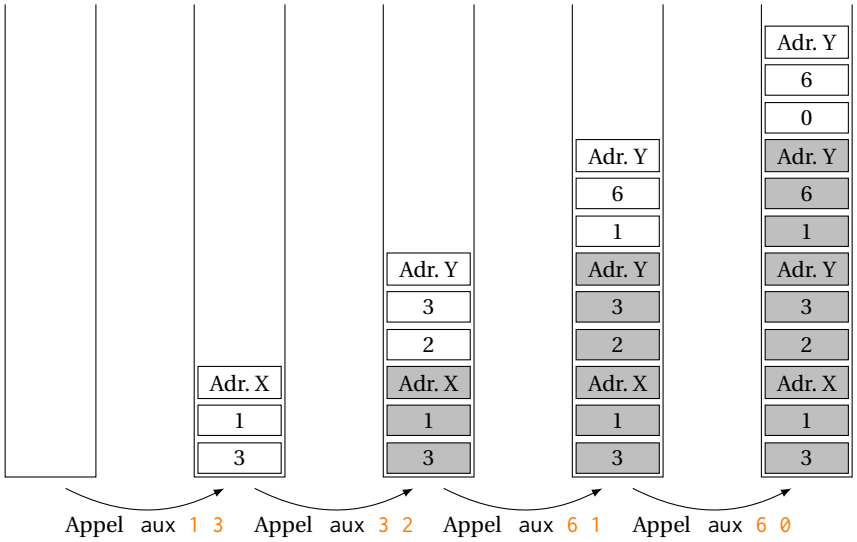
Il existe un mécanisme permettant d'économiser quelques opérations et un peu de place en mémoire : la *récursion terminale*. Regardons ce que donne le calcul de `fact 3` :



On peut écrire légèrement différemment la fonction `fact` :

```
# let fact_RT n =
    let rec aux res = function
      | 0 -> res
      | n -> aux (res*n) (n-1)
    in aux 1 n;;

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

Voyons ce qui se passe avec la fonction auxiliaire `aux` :



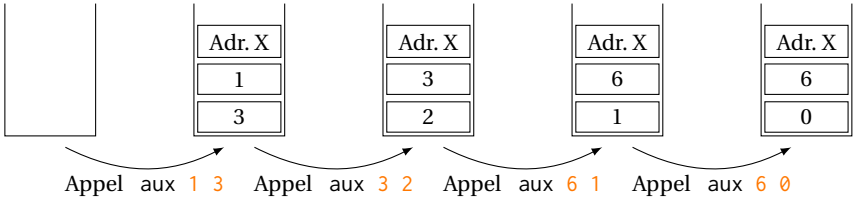
Pour l'instant, les avantages sont loin d'être évidents, on a plutôt augmenté l'occupation de la mémoire. Cependant, il se passe quelque chose d'intéressant lorsque l'on commence à s'intéresser aux retours des fonctions.

En effet, la fonction `aux 6 0` place son résultat, 6, dans un registre, et retourne. La fonction `aux 6 1` retourne directement le résultat que vient de lui retourner `aux 6 0`. Ce résultat est déjà dans le registre, elle n'a donc qu'à dépiler sa trame de pile et retourner à la fonction `aux 3 2` qui l'a appelée. `aux 3 2` se trouve dans la même situation.

Comme l'appel récursif se trouve être *la toute dernière opération* qu'effectue la fonction `Aux`, on peut alors effectuer une optimisation qui va avoir de l'importance. Plutôt que de remonter les appels un par un, on va s'arranger pour effectuer tous les retours d'un seul coup.

Pour ce faire, plutôt que d'empiler une nouvelle trame de pile lors d'un appel qui est la toute dernière instruction d'une fonction, on va modifier la trame actuellement au sommet de la pile, de façon à y substituer les nouveaux paramètres, mais en conservant l'adresse de retour.

La pile évoluera donc de la façon suivante :



On remarquera que pour tous les appels, dans l'exemple ci-dessus, les nouveaux arguments sont substitués aux anciens, mais l'adresse de retour n'est pas modifiée.

Lorsque l'on arrivera à la fin des appels récursifs, le résultat est placé dans le registre, et on saute directement à `Adr. X`, l'appelant (notre fonction `fact_RT` ici), en un seul retour. Plus le nombre d'appels récursifs sera important, plus le gain sera notable. Par ailleurs, puisque la pile ne se remplit pas, on ne risque pas d'avoir des soucis de mémoire !

Précisons cependant, encore une fois, que la lisibilité d'une fonction prime généralement sur des critères d'efficacité. Il n'est pas attendu que vous écriviez des fonctions utilisant le principe de récursion terminale [20], surtout lorsque cela complique notablement l'écriture de la fonction. Mais cela vous permettra peut-être de comprendre pour quelle raison certaines fonctions que vous rencontrerez sembleront écrites de façon un peu moins naturelles.

3.4 Quelques exemples

Pour calculer (récursivement) la somme des termes d'une liste, la solution la plus naturelle consiste à écrire :

```
# let rec somme = function
    | []   -> 0
    | t::q -> t + somme q;;

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

Si l'on souhaite en faire une fonction récursive terminale, on peut, comme dans le cas de la fonction `aux` de `fact_RT`, construire le résultat étape par étape, et passer les calculs

20. Et encore moins en Python où ce mécanisme d'optimisation de l'utilisation de la pile n'est pas utilisé, par choix.

intermédiaires en tant qu'argument supplémentaire à la fonction (le dernier appel ne faisant que retourner le paramètre correspondant au résultat, en général).

Il est alors généralement nécessaire de définir une fonction auxiliaire, travaillant avec un argument supplémentaire. Dans le cas de la fonction `Somme`, cela pourrait donner :

```
let somme liste =
  let rec aux s = function
    | []   -> s
    | t::q -> aux (s+t) q
  in aux 0 liste;;

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

Une telle fonction peut être légèrement plus rapide, mais permet aussi (et surtout) de traiter des listes plus longues sans que la pile ne « déborde », ce qui est très utile dans un langage fonctionnel qui fait la part belle aux fonctions récursives. Toutefois, le résultat n'étant pas aussi lisible, il convient d'utiliser cette possibilité avec parcimonie, lorsque c'est réellement utile.

Précisons qu'il n'est pas toujours nécessaire d'introduire une fonction auxiliaire, on aurait pu également écrire notre fonction `Somme` de la sorte :

```
# let rec somme = function
    | []        -> 0
    | [ elem ]  -> elem
    | t1::t2::q -> somme ((t1+t2)::q);;

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

L'important est que l'appel récursif soit la *dernière* opération qu'effectue la fonction.

Dans ce dernier cas, on utilise davantage le *conse*, aussi il n'est pas certain que les performances soient améliorées, c'est surtout la possibilité de traiter de longues listes qui peut faire pencher la balance en faveur d'une telle solution.

Profitons enfin de l'occasion pour éclaircir un point évoqué tantôt : comme on peut le constater ci-dessous, la fonction `List.fold_left` utilise une récursion terminale :

```
let rec fold_left f b = function
  | []   -> b
  | t::q -> fold_left f (f b t) q;;
```

En effet, l'appel récursif à `fold_left` est bien la toute dernière opération effectuée par la fonction, ce qui rend possible l'optimisation.

Ce n'est en revanche pas le cas de `List.fold_right`, dans laquelle le résultat de l'appel récursif sert ensuite d'argument à la fonction `f` :

```
let rec fold_right f lst b = match lst with
  | []   -> b
  | t::q -> f t (fold_right f q b);;
```

Aussi la fonction `fold_left` est-elle un peu plus rapide, mais surtout s'accomode de listes plus longues que sa consœur.

6 Paradigmes de programmation

Dans ce dernier chapitre, nous étudierons quelques stratégies pouvant permettre l'élaboration d'algorithmes plus efficaces en terme de complexité : division pour régner, programmation dynamique, etc. Nous en profiterons pour établir quelques résultats permettant de déterminer plus rapidement et plus simplement la complexité d'un algorithme. Nous aurons, par la même occasion, l'opportunité d'étudier quelques algorithmes intéressants pour résoudre des problèmes courants (tris, recherches, etc.)

1 Les tris

1.1 Objectif

Une des tâches que l'on rencontre fréquemment en informatique consiste à trier des données. Compte tenu de son importance, il a été apporté à ce problème un nombre conséquent de solutions, aux avantages et inconvénients variés. Il va nous permettre par ailleurs d'illustrer quelques concepts importants de programmation, mais également de revenir sur les calculs de complexité.

Dans la suite, nous chercherons donc à écrire une fonction de tri, prenant en argument une liste d'éléments (des flottants par exemple), et retournant une liste contenant les mêmes éléments, triés par ordre croissant.

1.2 Tri par sélection

Une façon naturelle de trier un ensemble d'éléments consiste à trouver, dans cet ensemble, le plus petit de ses éléments, puis le second plus petit, le troisième, et ainsi de suite jusqu'à épuisement des éléments de l'ensemble. On parle de *tri par sélection*.

L'écriture en Caml d'une telle méthode est assez simple [1]. Dans un premier temps, on commence par écrire une fonction `minReste` prenant en argument une liste, et retournant un couple constitué du plus petit élément de la liste, et de la liste des éléments restants [2].

1. On ne cherchera pas, dans ce cours, à obtenir des fonctions exhibant une récursion terminale, sauf si elle vient naturellement ; ce sont les algorithmes proprement dits qui nous intéressent ici, et non le détail leur implémentation.
2. Si l'élément le plus grand apparaît n fois dans la liste, il doit apparaître $n-1$ fois dans la liste retournée.

Comme souvent en Caml lorsque l'on manipule des listes, une solution récursive est assez naturelle. Après avoir séparé la liste en une tête et un reste, le plus petit élément est soit la tête, soit le plus petit élément du reste. La liste privée du plus petit élément contient la queue privée de son plus petit élément, à laquelle on rajoute l'élément parmi les deux précédents qui n'a pas été retenu comme plus petit élément de la liste :

```
# let rec minReste = function
    | []       -> failwith "Empty"
    | [ elem ] -> elem, []
    | t::q     -> let minimum, reste = minReste q in
                  (min t minimum), (max t minimum)::reste;;

val minReste : 'a list -> 'a * 'a list = <fun>
```

La terminaison de cet algorithme est garantie par le fait que la longueur de la liste décroit strictment à chaque appel, sa correction est immédiate par récurrence.

Trier les données consiste alors simplement à chercher successivement les plus petits éléments et à les assembler, ce qui s'écrit très simplement de façon récursive :

```
# let rec tri = function
    | []    -> []
    | liste -> let minimum, reste = minReste liste in
               minimum::(tri reste);;

val tri : 'a list -> 'a list = <fun>
```

Intéressons-nous à présent au temps nécessaire à l'exécution des fonctions `minReste` et `tri`. Bien que nos fonctions soient polymorphes, on s'intéressera ici uniquement au cas du tri de liste d'entiers (ou de flottants) de sorte que le temps de comparaison de deux éléments a et b reste borné [3]

Dans la suite, nous noterons $\mathcal{D}_n$ l'ensemble des listes comprenant n éléments, et $\mathrm{T}_{\mathrm{MR}}(d)$ le temps d'exécution de la fonction `minReste` pour une liste d.

Un appel à la fonction`minReste` sur un élément de $\mathcal{D}_n$ consiste en un appel récursif à `minReste` avec pour paramètre un élément de $\mathcal{D}_{n-1}$ et une succession d'opérations (extraction de la tête, calcul du minimum et du maximum, conse...) dont le temps d'exécution ne dépend pas de n, et que l'on peut encadrer par deux temps t_1 et t_2.

Pour la question de la complexité, cet encadrement n'a pas besoin d'être précis, « entre une femtoseconde et un milliard d'années » est suffisant ! L'important est que cet encadrement soit valable *quel que soit l'argument* de la fonction `minReste`.

3. En effet, lorsque l'on compare des chaînes de caractères ou des listes, par exemple, le temps nécessaire peut dépendre de la taille des chaînes ou des listes comparées.

On peut donc écrire, pour tout entier $n \geqslant 2$ et en notant $T_{MR}(d)$ le temps d'exécution [4] de la fonction `minReste` sur la liste $d \in \mathcal{D}_n$:

$$\forall d \in \mathcal{D}_n, \qquad t_1 + \min_{d' \in \mathcal{D}_{n-1}} (T_{MR}(d')) \leqslant T_{MR}(d) \leqslant t_2 + \max_{d' \in \mathcal{D}_{n-1}} (T_{MR}(d'))$$

Par une récurrence immédiate, cela conduit, pour tout $n \geqslant 1$, à :

$$\forall d \in \mathcal{D}_n, \qquad (n-1) \times t_1 + \min_{d' \in \mathcal{D}_1} (T_{MR}(d')) \leqslant T_{MR}(d) \leqslant (n-1) \times t_2 + \max_{d' \in \mathcal{D}_1} (T_{MR}(d'))$$

Puisque l'on peut encadrer le temps $T_{MR}(d')$ pour un quelconque élément de $\mathcal{D}_1$ par deux constantes, il existe donc des constantes réelles strictement positives α, β, γ et δ telles que, pour tout $n \geqslant 1$:

$$\forall d \in \mathcal{D}_n, \qquad \alpha n + \beta \leqslant T_{MR}(d) \leqslant \gamma n + \delta$$

Ce qui signifie que la fonction `minReste` a un coût linéaire, une complexité $\Theta(n)$.

Penchons-nous à présent sur la fonction `tri`. Pour un argument pris dans $\mathcal{D}_n$ avec $n \geqslant 1$, elle comprend un appel à la fonction `minReste` avec le même argument, un appel récursif à la fonction `tri` avec pour argument un élément de $\mathcal{D}_{n-1}$, et un ensemble d'opérations dont le temps peut être encadré par des temps t'_1 et t'_2. Aussi peut-on écrire, pour tout entier $n \geqslant 1$ et en notant $T_{tri}(d)$ le temps d'exécution de la fonction `tri` pour un argument $d \in \mathcal{D}_n$:

$$\forall d \in \mathcal{D}_n, \qquad t'_1 + \alpha n + \beta + \min_{d' \in \mathcal{D}_{n-1}} (T_{tri}(d')) \leqslant T_{tri}(d) \leqslant t'_2 + \gamma n + \delta + \max_{d' \in \mathcal{D}_{n-1}} (T_{tri}(d'))$$

Cette fois encore, une récurrence immédiate permet, pour tout $n \geqslant 1$, d'établir que :

$$\forall d \in \mathcal{D}_n, \qquad n \times (t'_1 + \beta) + \frac{n(n+1)}{2}\alpha + T_{tri}([]) \leqslant T_{tri}(d) \leqslant n \times (t'_2 + \delta) + \frac{n(n+1)}{2}\gamma + T_{tri}([])$$

On a donc un coût quadratique pour notre fonction de tri, une complexité $\Theta(n^2)$.

Dans la pratique, on tentera rarement d'encadrer le temps d'exécution d'une fonction, et on préférera dénombrer le nombre d'occurrences de l'une des opérations qui conditionne la complexité de la fonction. Par exemple, dans un tri, on pourra s'intéresser au nombre de comparaisons. En effet, le temps d'exécution sera, nécessairement, au moins proportionnel au nombre de comparaisons effectuées. Mais, puisque inversement, pour chaque comparaison on effectue un nombre borné d'autres opérations toutes élémentaires, le temps d'exécution est aussi majoré par un temps proportionnel au nombre de comparaisons. Le nombre de comparaisons et le temps d'exécution sont des suites équivalentes.

Si l'on note u_n le nombre de comparaisons effectuées par `minReste` pour une liste comprenant n éléments, et u'_n le nombre de comparaisons effectuées par `tri` pour une liste de n éléments, on peut écrire [5]

$$u_0 = u_1 = 0 \qquad \text{et} \qquad \forall n \geqslant 2, \quad u_n = 2 + u_{n-1}$$

ce qui conduit, pour $n \geqslant 2$, à

$$u_n = 2 \times (n-1)$$

et, pour la fonction `tri`

$$u'_0 = 0 \qquad \text{et} \qquad \forall n \geqslant 2, \quad u'_n = u'_{n-1} + u_n = u'_{n-1} + 2 \times (n-1)$$

ce qui donne, pour $n \geqslant 1$,

$$u'_n = \sum_{k=2}^{n} 2 \times (k-1) = 2 \sum_{k=1}^{n-1} k = n \times (n-1) = n^2 - n$$

Un appel à `tri` effectue donc de l'ordre de n^2 comparaisons, ce qui permet de retrouver que cette fonction a un coût quadratique $\Theta(n^2)$.

1.3 Tri par insertion

Une autre méthode de tri possible consiste à partir d'une liste vide, et à « insérer » tour à tour chacun des éléments à trier dans cette liste, en les plaçant de sorte que la liste que l'on construit reste à chaque instant triée. On parle de *tri par insertion.*

On commence donc par écrire une fonction permettant d'insérer un élément à la bonne place dans une liste d'éléments triés par ordre croissant, en utilisant cette fois encore une approche récursive :

```
# let rec insere elem = function
    | (t::q) when elem > t -> t::insere elem q
    | lst                  -> elem::lst;;

val insere : 'a -> 'a list -> 'a list = <fun>
```

4. Dans la réalité, le temps d'exécution d'une fonction sur un argument donné peut varier, même si l'argument ne change pas, en fonction de nombreux critères, donc on pourrait dire « un quelconque temps d'exécution de la fonction » pour l'argument d. En pratique, on ne se souciera pas, dans la pratique, de tels détails, car on ne recherche qu'un équivalent et ces fluctuations de temps d'exécution ne changeront pas la complexité.

5. Le « 2 » dans la relation de récurrence vient de la présence d'un `min` et d'un `max`, on pourrait le réduire à 1 en utilisant par exemple un test, mais la complexité sera la même dans les deux situations. Notons que l'on n'a pas compté ici les comparaisons que Caml devra effectuer afin de filtrer l'argument. En nombre moindre, dans l'exécution de la fonction, que les comparaisons entre éléments de la liste, elles ne changeraient de toute façon pas la complexité.

Là encore, la taille de la liste passée en second argument décroit à chaque appel, donc la fonction va toujours terminer. Pour la correction, l'élément `elem` se retrouve juste avant le premier élément de la liste qui soit plus grand que lui, aussi est-il placé au bon endroit.

L'insertion d'un « 2 » dans une liste de sept éléments triés par ordre croissant place bien l'élément à la position idoine :

```
insere 2 [ -5; -2; 0; 1; 3; 4; 7; 9 ];;

- : int list = [-5; -2; 0; 1; 2; 3; 4; 7; 9]
```

À présent le tri par insertion proprement dit s'écrit très simplement :

```
# let rec tri = function
    | []   -> []
    | t::q -> insere t (tri q);;

val tri : 'a list -> 'a list = <fun>
```

Cela correspond bien à ce que l'on décrivait tantôt, car l'appel

```
tri [ 2; 3; 9; -5; 4; 1; -2; 7; 0 ];;
```

correspond, si l'on déroule les différents appels récursifs, à

```
insere 2 (insere 3 (insere 9 ( insere (-5) (insere 4
    (insere 1 (insere (-2) (insere 7 (insere 0 [])))))))))
```

Pour déterminer la complexité en temps de ce tri par insertion, nous allons à nouveau nous pencher sur le nombre de comparaisons effectuées par ces deux fonctions. Cette fois, le nombre de comparaisons pour une liste contenant n éléments n'est pas toujours le même. On s'intéresse ici à la complexité *dans le pire des cas*, celui pour lequel le nombre de comparaisons est maximal.

Si l'on note u_n le nombre de comparaisons effectuées par la fonction `insere` dans le pire des cas, pour une insertion d'un élément dans une liste à n éléments, il apparaît très vite que $u_n = n$ (lorsque l'insertion se fait en bout de liste, après avoir constaté que tous les éléments de la liste sont plus grands que l'élément à insérer). La complexité en temps, dans le pire des cas [6], de `insere` est donc $\Theta(n)$.

Le nombre u'_n de comparaisons nécessaires, dans le pire des cas, pour appliquer `tri` à une liste de longueur n vérifie [7]

$$u'_0 = 0 \qquad \text{et} \qquad \forall n \geqslant 1,\ \ u'_n = u'_{n-1} + u_{n-1}$$

6. Dans le cas favorable, en revanche, $u_n = 1$, donc on a une fonction en temps constant.
7. On peut vérifier que le pire des cas pour `tri` est une liste triée par ordre décroissant, qui provoque bien à chaque appel de `insere` la pire insertion possible, en toute fin de liste.

On a donc, pour tout $n \geqslant 1$,

$$u'_n = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Cette fois encore, dans le pire des cas, le nombre de comparaisons est quadratique, donc le tri par insertion a une complexité en temps dans le pire des cas en $\Theta(n^2)$.

En revanche, dans le cas le plus favorable, puisque pour tout n, $u_n = 1$, on a $u'_n = n - 1$, ce qui donne un coût en temps linéaire. Ce genre de situation se produit pour des listes triées par ordre croissant (ou presque triées), pour lesquelles ce tri peut être très efficace.

1.4 Aller plus loin

Les coûts des deux fonctions de tri précédentes deviennent rapidement prohibitifs lorsque le nombre d'éléments à trier est important. Pour seulement 10000 éléments, les tris précédents peuvent nécessiter plusieurs dizaines de secondes sur une machine courante, et cent fois plus (donc près d'une heure) pour 100000 éléments [8].

Dans le cas du tri par insertion, le coût en $\Theta(n^2)$ du tri vient, en partie, du fait que la fonction qui insère un élément dans une liste triée a un coût en $\Theta(n)$.

Dans le cours de tronc commun, il a été montré que l'on pouvait trouver cette position plus efficacement, par une recherche dichotomique dans la liste triée, sous réserve de pouvoir accéder aux éléments de la liste en $O(1)$ [9]. En effet, cette recherche dichotomique peut trouver la position où devra être inséré l'élément en un temps $O(\ln(n))$ (en n'effectuant que $\lceil \log_2(n) \rceil$ comparaisons pour ce faire).

On pourrait envisager d'utiliser cette idée pour transformer notre tri par insertion en un tri en $O(n\ln(n))$. En pratique, cela ne fonctionne pas. En effet, pour déterminer la place en $\log(n)$, il faut pouvoir accéder aux éléments en $O(1)$, ce qui nécessite de conserver les éléments dans une structure de type tableau (`array` en Caml). Mais dans ce cas, l'*insertion* de l'élément à la bonne place est une opération en $O(n)$. La recherche dichotomique ne peut donc pas nous aider directement ici.

En revanche, l'idée derrière la recherche dichotomique est intéressante : lorsque l'on compare l'élément à insérer avec celui au milieu de la liste triée, le résultat permet d'éliminer la moitié des positions possibles. C'est un des très nombreux exemples d'application d'un paradigme en informatique, « *diviser pour régner* ».

8. Une partie de cette « lenteur » est à attribuer au compilateur relativement ancien de Caml Light, mais celui-ci n'est responsable que d'un coefficient multiplicatif, une liste dix fois plus longue nécessitera quoi qu'il arrive un temps cent fois plus grand, ce qui pose problème.
9. Ce qui n'est pas le cas des listes en Caml

2 Diviser pour régner

2.1 Présentation

Le paradigme de programmation « diviser pour régner » consiste à ramener la résolution d'un problème dépendant d'un entier n en un nombre borné de problèmes identiques dépendant d'un entier $n' \simeq \alpha n$ avec [10] $\alpha < 1$.

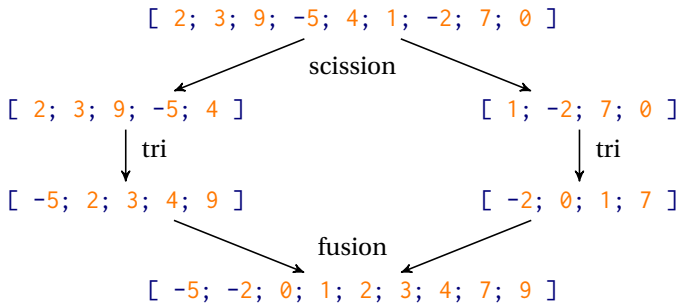
Fréquemment, on a $\alpha = 1/2$. Par exemple, dans le cas de la recherche dichotomique, déjà vue dans le cours de tronc commun et sur laquelle nous allons revenir, rechercher la position adéquate dans une liste à n éléments se traduit en une recherche de la position adéquate dans une liste à $\lceil n/2 \rceil$ éléments.

2.2 Tri fusion

Tentons d'appliquer ce paradigme « diviser pour régner » au problème du tri des éléments d'une liste.

Il est possible de trier une liste à n éléments en procédant de la sorte :

- si la liste est vide ou contient un unique élément, elle est déjà triée, il n'y a donc rien à faire ;
- sinon, on scinde la liste à trier en deux listes contenant respectivement $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$ éléments ;
- puis on trie chacune des deux listes (ce qui revient à résoudre le même problème, sur des listes de taille deux fois plus petite) ;
- enfin, on fusionne les deux listes triées en une seule liste triée.



Commençons par écrire des fonctions pour chacune de ces étapes.

Tout d'abord, considérons le problème de la scission de la liste. Comme on ne dispose pas, en général, de la longueur de la liste, la façon la plus simple de procéder est de prendre les éléments de la liste fournie en paramètre un par un, et de les répartir alternativement dans les deux listes qui seront retournées. Cela peut s'écrire grâce à un filtrage :

```
# let rec scinde = function
    | t1::t2::q -> let q1, q2 = scinde q in t1::q1, t2::q2
    | lst       -> lst, [];;

val scinde : 'a list -> 'a list * 'a list = <fun>
```

Dans le cas d'une liste avec un nombre impair d'éléments, la première des deux listes retournée est plus longue, comme sur l'exemple ci-dessous :

```
# scinde [ 2; 3; 9; -5; 4; 1; -2; 7; 0 ];;

- : int list * int list = ([2; 9; 4; -2; 0], [3; -5; 1; 7])
```

La fusion de deux listes, notre troisième étape, s'écrit aussi avec un filtrage. Si les listes (triées) passées en argument sont non vides, l'élément le plus petit parmi les éléments en tête de chacune des deux listes prend la première place dans la liste résultat, le reste étant constitué de la fusion des autres éléments de chacune des deux listes :

```
# let rec fusionne l1 l2 = match (l1, l2) with
    | (t1::q1), (t2::q2) when t1 <= t2 -> t1::(fusionne q1 l2)
    | l1,       (t2::q2)               -> t2::(fusionne l1 q2)
    | l1,       []                     -> l1;;

val fusionne : 'a list -> 'a list -> 'a list = <fun>
```

Le résultat est bien une liste d'éléments rangés par ordre croissant :

```
# fusionne [ -2; 0; 2; 4; 9 ] [ -5; 1; 3; 7 ];;

- : int list = [-5; -2; 0; 1; 2; 3; 4; 7; 9]
```

Il ne reste plus ensuite qu'à écrire le tri tel que nous l'avons décrit précédemment, en exhibant encore une récursion :

```
# let rec tri lst = match scinde lst with
    | lst, [] -> lst
    | l1, l2  -> fusionne (tri l1) (tri l2);;

val tri : 'a list -> 'a list = <fun>
```

10. On remarquera que $n' = n - 1$ (ou $n' = n - k$) ne convient pas, car on aurait $n' \simeq n$ lorsque $n \to \infty$, en contradiction avec $\alpha < 1$.

Toutes ces fonctions terminent car les appels se font systématiquement sur des listes strictement plus courtes, et sont par ailleurs correctes.

Intéressons-nous à présent aux coûts en terme de temps de calcul de ces différentes fonctions.

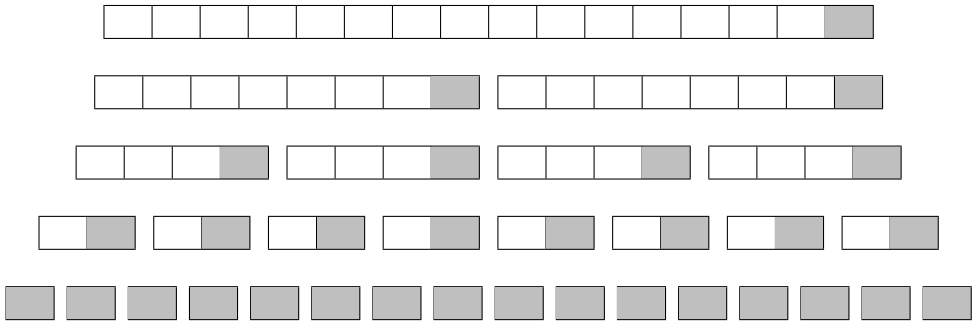
Pour une liste à n éléments, découpée en deux listes de longueurs $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$ éléments, la fonction `fusionne` effectue, dans le pire des cas, $n-1$ comparaisons [11].

Si l'on note u_n le nombre de comparaisons effectuées, dans le pire des cas, par la fonction `tri` lorsqu'elle trie une liste de longueur n, on peut écrire

$$\begin{cases} u_0 = u_1 = 0 \\ \forall n \geqslant 2, \quad u_n = u_{\lfloor n/2 \rfloor} + u_{\lceil n/2 \rceil} + (n-1) \end{cases}$$

On remarquera que la fonction `scinde` n'apparaît pas dans ce décompte, mais puisqu'elle effectue un nombre d'opérations du même ordre de grandeur que la fonction `fusionne`, cela ne changera pas la complexité du tri.

Pour essayer de comprendre le comportement de u_n, supposons que n s'écrit de la forme 2^k (par exemple $2^4 = 16$) et observons comment les choses se passent. Dans le schéma suivant, chaque « groupe » représente un appel pour une liste ayant le même nombre d'éléments que le nombre de cases dans le groupe (chaque ligne représentant une récursion), et chaque case blanche correspond à une comparaison :



On peut voir sur le schéma que pour $n = 2^k$, on a 2^k cases sur chaque ligne, et $k+1$ lignes, donc $2^k \times (k+1)$ cases au total. On peut également montrer que $1+2+4+\ldots+2^k$ de ces cases ne sont pas blanches, autrement dit $2^{k+1}-1$.

Sur une liste à 2^k éléments, le tri fusion effectue donc, dans le pire des cas, $2^k \times (k-1)+1$ comparaisons. Puisque $k = \log_2(n)$, cela revient à $n \times \log_2(n) - n + 1$.

11. et $\lfloor n/2 \rfloor$ comparaisons dans le meilleur des cas.

On s'attend donc à une complexité, pour notre fonction de tri fusion, dans le pire des cas [12], en $\Theta(n\log(n))$.

2.3 Comportement asymptotique de suites récurrentes

Pour s'éviter ces calculs dans d'autres situations similaires, nous allons présenter quelques résultats généraux qui permettront de conclure plus rapidement.

Suites récurrentes d'ordre 1

Théorème 4. *Soit $a \in \mathbf{R}^{+\star}$, $(b_n)_{n\in\mathbb{N}}$ une suite réelle positive, et $(u_n)_{n\in\mathbb{N}}$ une suite vérifiant*

$$u_n = a \cdot u_{n-1} + b_n$$

On montre que

- *si $(b_n) = \Theta(n^\nu)$ et $a = 1$, alors $(u_n) = \Theta(n^{\nu+1})$;*
- *si $(b_n) = o(n^\nu)$ et $a > 1$, alors $(u_n) = \Theta(a^n)$;*
- *si $(b_n) \sim \lambda a^n$ avec $\lambda > 0$, alors $(u_n) = \Theta(na^n)$;*
- *si $(b_n) \sim \lambda b^n$ avec $\lambda > 0$ et $b < a$, alors $(u_n) = \Theta(a^n)$;*
- *si $(b_n) \sim \lambda b^n$ avec $\lambda > 0$ et $b > a$, alors $(u_n) = \Theta(b^n)$.*

Démonstration. La première affirmation se démontre aisément.
Pour les autres, posons $v_n = \frac{u_n}{a^n}$. La relation de récurrence sur v_n s'écrit donc, $\forall n \in \mathbb{N}$,

$$v_n = v_{n-1} + \frac{b_n}{a^n}$$

On a donc $v_n = u_0 + \sum_{k=1}^{n} \frac{b_n}{a^n}$ soit $u_n = a^n \left(u_0 + \sum_{k=1}^{n} \frac{b_n}{a^n} \right)$.

• Dans le cas où $(b_n) = o(n^\nu)$, on peut écrire

$$\forall \varepsilon > 0, \quad \exists n_0 \in \mathbb{N} \quad \text{tel que} \quad \forall k \geqslant n_0, \quad \frac{b_k}{a^k} \leqslant \varepsilon \frac{k^\nu}{a^k}$$

Or la série de terme $\frac{k^\nu}{a^k}$ converge, d'où le second résultat.
Il en est de même si $(b_n) \sim \lambda b^n$ avec $\lambda > 0$ et $b < a$.

• Si $(b_n) \sim \lambda a^n$ avec $\lambda > 0$, alors $u_n \simeq a^n (u_0 + n\lambda)$ donc $(u_n) = \Theta(na^n)$.

• Si $(b_n) \sim \lambda b^n$ avec $\lambda > 0$ et $b > a$, alors $u_n \simeq a^n \left(u_0 + \frac{\lambda}{b-a} \left(\frac{b}{a} \right)^n \right)$ donc $(u_n) = \Theta(b^n)$. □

12. Il en serait de même dans le cas favorable, le fait que l'on effectue $\lfloor n/2 \rfloor$ comparaisons au lieu de $(n-1)$ ne change pas le comportement assymptotique de la suite.

La plupart de ces résultats sont logiques et prévisibles : si le terme (b_n) est négligeable devant la suite $u'_n = au'_{n-1}$, c'est le comportement de cette suite qui gouverne celui de la suite (u_n), soit $\Theta(a^n)$; si, à l'inverse, la suite $u'_n = au'_{n-1}$ croit moins vite que b_n, alors c'est b_n qui va déterminer le comportement assymptotique de la suite. Seul le cas où les deux termes sont de grandeur comparable conduit à un comportement assymptotique un peu plus complexe.

Suites récurrentes de type « diviser pour régner »

Les suites récurrentes de type « diviser pour régner » font généralement apparaître une suite $(u_n)_{n\in\mathbb{N}}$ gouvernée par une relation de la forme [13]

$$u_n = a_1 \cdot u_{\lfloor n/2 \rfloor} + a_2 \cdot u_{\lceil n/2 \rceil} + b_n$$

Avant de s'intéresser au comportement assymptotique de telles suites, établissons d'abord quelques résultats utiles.

Remarquons tout d'abord que

Lemme 1. *Si a_1 et a_2 deux réels positifs vérifiant $a_1 + a_2 \geqslant 1$, et si $(b_n)_{n\in\mathbb{N}}$ et $(b'_n)_{n\in\mathbb{N}}$ sont deux suites de même ordre de grandeur, alors les suites $(u_n)_{n\in\mathbb{N}}$ et $(u'_n)_{n\in\mathbb{N}}$ telles que $u_0 = u'_0$ et pour tout $n \in \mathbb{N}^*$,*

$$u_n = a_1 \cdot u_{\lfloor n/2 \rfloor} + a_2 \cdot u_{\lceil n/2 \rceil} + b_n \quad \text{et} \quad u'_n = a_1 \cdot u'_{\lfloor n/2 \rfloor} + a_2 \cdot u'_{\lceil n/2 \rceil} + b'_n$$

sont du même ordre de grandeur également.

Cette propriété nous permet de substituer à b_n, dans les démonstrations, une suite de même ordre de grandeur. On peut également montrer que

Lemme 2. *Si a_1 et a_2 sont deux réels positifs vérifiant $a_1 + a_2 \geqslant 1$, et si $(b_n)_{n\in\mathbb{N}}$ est une suite croissante positive, alors la suites $(u_n)_{n\in\mathbb{N}}$ définie par*

$$u_n = a_1 \cdot u_{\lfloor n/2 \rfloor} + a_2 \cdot u_{\lceil n/2 \rceil} + b_n$$

est également croissante.

Démonstration. On peut en effet montrer par récurrence que $\forall n \in \mathbb{N}$, $u_{n+1} \geqslant u_n$:
• $u_2 = a_1 \cdot u_1 + a_2 \cdot u1 + b_2 = (a_1 + a_2)u_1 + b_2 \geqslant (a_1 + a_2)u_1 \geqslant u1$, donc c'est vrai pour $n = 1$;
• si l'on suppose vraie la propriété pour tout $k \leqslant n$, on a

$$u_{n+1} = a_1 \cdot u_{\lfloor (n+1)/2 \rfloor} + a_2 \cdot u_{\lceil (n+1)/2 \rceil} + b_{n+1} \geqslant a_1 \cdot u_{\lfloor n/2 \rfloor} + a_2 \cdot u_{\lceil n/2 \rceil} + b_n = u_n$$

car (b_n) est croissante, de même que $u_{\lfloor (n+1)/2 \rfloor} \geqslant u_{\lfloor n/2 \rfloor}$ et $u_{\lceil (n+1)/2 \rceil} \geqslant u_{\lceil n/2 \rceil}$ d'après l'hypothèse de récurrence.
La propriété est donc vraie pour tout $n \geqslant 1$, la suite (u_n) est donc croissante. □

Ces résultats préalables étant mis en place, on peut se pencher à présent sur les questions de complexité.

Théorème 5. *Soient a_1 et a_2 deux réels positifs vérifiant $a_1 + a_2 \geqslant 1$, $(b_n)_{n\in\mathbb{N}}$ une suite positive et croissante, et $(u_n)_{n\in\mathbb{N}}$ une suite vérifiant*

$$u_n = a_1 \cdot u_{\lfloor n/2 \rfloor} + a_2 \cdot u_{\lceil n/2 \rceil} + b_n$$

On montre les résultats suivants, où $\alpha = \log_2(a_1 + a_2)$:

- *si $(b_n) = \Theta(n^\alpha)$, alors $(u_n) = \Theta(n^\alpha \log(n))$;*
- *si $(b_n) = \Theta(n^\beta)$ avec $\beta < \alpha$, alors $(u_n) = \Theta(n^\alpha)$;*
- *si $(b_n) = \Theta(n^\beta)$ avec $\beta > \alpha$, alors $(u_n) = \Theta(n^\beta)$.*

Démonstration. Pour démontrer ces résultats, il suffit de considérer la suite $(v_k)_{k\in\mathbb{N}}$ définie, pour tout $k \in \mathbb{N}$, par $v_k = u_{2^k}$.
Cette suite vérifie, pour tout $k \in \mathbb{N}^\star$, $v_k = (a_1 + a_2) \cdot v_{k-1} + b'_k$ avec $b'_k = b_{2^k}$.
Supposons par exemple que $(b_n) = \Theta(n^\alpha)$. On a alors $(b'_k) = \Theta\left(2^{k\alpha}\right) = \Theta\left((a_1 + a_2)^k\right)$, ce qui conduit à $(v_k) = \Theta\left(k(a_1 + a_2)^k\right)$ grâce aux relations établies pour les suites récurrentes d'ordre 1.
Ensuite, puisque (u_n) est croissante, $v_{\lfloor \log_2(n) \rfloor} \leqslant u_n \leqslant v_{\lfloor \log_2(n) \rfloor + 1}$, ce qui permet de conclure que $(u_n) = \Theta\left(\log_2(n) \times (a_1 + a_2)^{\log_2(n)}\right) = \Theta(n \ln n^\alpha)$. □

2.4 Recherche dichotomique

Supposons que l'on dispose d'un tableau (`'a array`) `tab` d'éléments ordonnés de façon croissante ainsi qu'un élément `elem`, et que l'on souhaite déterminer l'indice i vérifiant

- $i = 0$ si tous les éléments du tableau sont supérieurs à `elem` ;
- $i = n$ (où n est la longueur du tableau) si tous les éléments de la liste sont strictement inférieurs à `elem` ;
- un i vérifiant `tab.(i-1)` < `elem` ⩽ `tab.(i)` sinon.

Autrement dit, une position `i` juste avant laquelle insérer [14] `elem` dans le tableau pour conserver une liste triée [15].

La solution immédiate consisterait à envisager toutes les possibilités une par une, en parcourant le tableau. Évidemment, cet algorithme a une complexité linéaire ($\Theta(n)$) dans le pire des cas, puisque l'on examinera alors les $n+1$ emplacements possibles un à un.

13. En général, a_1 et a_2 sont des entiers positifs (un des deux au moins étant non nul).

14. Ce qui, dans un tableau, nécessitera de décaler tous les éléments aux positions $j \geqslant i$ au préalable.

15. Compte tenu des comparaisons, si plusieurs positions sont possibles, c'est la position la plus à gauche qui sera retournée ici.

L'algorithme de recherche dichotomique travaille plus efficacement. Après avoir envisagé les deux premiers cas ($i = 0$ et $i = n$), il tente de trouver les indices des deux éléments de `tab` qui encadrent `elem` en réduisant de moitié les possibilités pour la valeur de `i` à chaque itération [16].

L'algorithme peut donc s'écrire ainsi :

```
# let dicho elem tab =
    let n = Array.length tab in
      if n == 0 || elem <= tab.(0) then 0
      else if elem > tab.(n-1) then n
      else let rec aux a b =
             if b = a+1 then b
             else let m = (a+b)/2 in
                    if elem <= tab.(m) then aux a m else aux m b
             in aux 0 (n-1);;

val dicho : 'a -> 'a array -> int = <fun>
```

Dans cette fonction, l'invariant lors de chaque appel récursif à la fonction aux peut s'écrire « le `i` recherché se trouve dans l'intervalle $]a, b]$ ».

Si le résultat n'est pas $i = 0$ ou $i = n$, on a initialement $n-1$ possibilités (de $a+1 = 1$ à $b = n-1$ inclus). Après une comparaison, on en est réduit à $\lceil (n-1)/2 \rceil$ possibilités. Et ainsi de suite. Le nombre u_n de comparaisons effectuée dans la partie principale de l'algorithme [17] vérifie donc

$$u_n = u_{\lceil n/2 \rceil} + 1$$

ce qui conduit (avec ici $a_1 = 0$, $a_2 = 1$, $\alpha = 0$, $b_n = 1$ donc $(b_n) = \Theta(n^0)$) à une complexité pour la recherche dichotomique en $\Theta\left(n^0 \log(n)\right) = \Theta\left(\log(n)\right)$.

L'algorithme termine bien, car $|b-a|$ est bien une suite strictement décroissante. En effet, on a toujours $a < m < b$ *car b est toujours strictement supérieur $a+1$* ! Dans le cas $b = a+1$, on aurait $m = \lfloor (a+b)/2 \rfloor = a$, mais on n'effectue alors plus d'appel récursif. Si l'algorithme de la recherche dichotomique paraît simple, un grain de sable se glisse très facilement dans son implémentation, et il convient d'être très prudent quant à sa terminaison !

Nous avons utilisé ici des tableaux (`'a array`), car il était nécessaire de pouvoir accéder directement (en un temps $O(1)$) à l'élément d'index `i`.

Ce n'est pas le cas avec des listes, et le temps nécessaire pour accéder aux éléments qui ne sont pas en tête de liste conduirait, si l'on essayait d'appliquer l'algorithme dichotomique à des listes, en une complexité $\Theta(n)$, dénuée d'intérêt car elle n'est pas meilleure qu'une recherche linéaire dans la liste !

16. On se reportera au cours de tronc commun pour les détails.
17. Sans tenir compte des deux comparaisons initiales qui ne changeront pas le résultat.

2.5 Algorithme d'exponentiation rapide

On s'intéresse à présent au calcul de la n$^{\text{e}}$ puissance d'un élément « x » (qui peut être un réel, une matrice, etc.).

La solution naive pour calculer x^n consiste à écrire

$$x^n = x \times x^{n-1} = \ldots = ((((\ldots((x \times x) \times x) \times \cdots) \times x) \times x) \times x)$$

Soit, en Caml, pour une utilisation sur des x entiers :

```
# let rec power x = function
    | 0 -> 1
    | n -> x * power x (n-1);;

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

Le calcul nécessite alors $n-1$ multiplications, donc une complexité en $\Theta(n)$ si le coût de la multiplication est constant. Si n est grand, cela peut représenter un temps de calcul important, surtout si la multiplication est complexe (si x est une matrice par exemple).

Une solution plus efficace [18] consiste à utiliser le paradigme « diviser pour régner », en remarquant que, pour $n \geqslant 2$,

- $x^n = x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor}$ si n est pair ;
- $x^n = x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} \times x$ si n est impair.

L'écriture en Caml de cet algorithme est immédiate :

```
# let rec power x = function
    | 0 -> 1
    | 1 -> x
    | n -> let y = power x (n/2) in
             if n mod 2 = 0 then y * y
             else y * y * x;;

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

Le second argument dans l'appel récursif est bien un entier positif strictement inférieur à n, ce qui garantit la terminaison de l'algorithme. Sa correction découle immédiatement des deux égalités précédentes.

Le nombre de multiplications nécessaires pour un argument n est u_n où $u_0 = u_1 = 1$ et

$$u_n = u_{\lfloor n/2 \rfloor} + 1 + (n \mod 2)$$

18. Cette méthode n'est pas non plus celle qui effectue toujours le moins de multiplications, la méthode des arbres de Knuth donnant fréquemment une solution un peu meilleure. On ne connaît pas de méthode donnant systématiquement et efficacement la solution optimale, en terme de nombre de multiplications, à ce problème.

Pour déterminer le comportement de cette suite, on peut changer légèrement le second membre (puisqu'il suffit de ne pas changer la complexité de la suite (b_n)), et s'intéresser à u'_n vérifiant

$$u'_n = u'_{\lfloor n/2 \rfloor} + 1$$

D'après les résultats précédents (avec ici $a_1 = 1$, $a_2 = 0$, $\alpha = 1$, $b_n = 1$ donc $(b_n) = \Theta(n^0)$), la complexité de l'algorithme sera donc $\Theta(\log(n))$.

Il convient aussi d'être prudent. On aurait pu considérer $x = x^{\lfloor n/2 \rfloor} \times x^{\lceil n/2 \rceil}$ et en déduire un autre algorithme :

```
# let rec power x = function
    | 0 -> 1
    | 1 -> x
    | n -> power x (n/2) * power x ((n+1)/2);;

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

On montre aisément que cette seconde possibilité termine et est correcte. Mais le nombre de multiplications est gouverné par une suite $(u_n)_{n\in\mathbb{N}}$ vérifiant la récurrence

$$u_n = u_{\lfloor n/2 \rfloor} + u_{\lceil n/2 \rceil} + 1$$

Or, cette récurrence conduit [19] à une complexité en $\Theta(n)$ (linéaire, donc), qui n'apporte donc aucun gain par rapport à la méthode « naive », contrairement à ce que l'on pourrait croire au premier coup d'œil !

2.6 Plus proches voisins dans un nuage de points du plan

On s'intéresse à présent à un ensemble de n points M_i du plan, de coordonnées (x_i, y_i). On souhaite connaître la distance minimale entre deux points de ce nuage (et éventuellement un couple de points M_i et M_j réalisant cette distance).

Une solution naïve consisterait à calculer toutes les distances M_iM_j avec $i > j$, soit $n(n-1)/2$ calculs de distance, ce qui conduit à une complexité en $\Theta(n^2)$. Toutefois, avec le paradigme « diviser pour régner », il est possible de faire mieux, et d'obtenir une complexité quasi-linéaire.

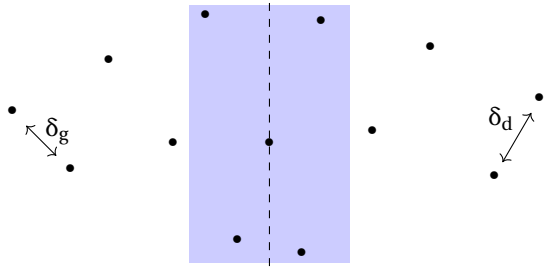
Dans un premier temps, on crée tout d'abord deux listes $\mathcal{P}$ et $\mathcal{P}'$ dans lesquels les différents points du nuage sont rangés respectivement par abscisse croissante [20], et par ordonnée croissante. Ces deux opérations peuvent être réalisées en $\Theta\big(n\log(n)\big)$, par exemple en utilisant le tri fusion présenté tantôt.

19. Puisque $a_1 = 1$, $a_2 = 1$, donc $\alpha = 1$, et $\beta = 0 < \alpha$

20. Pour l'implémentation proprement dite, il peut être utile que la liste $\mathcal{P}$ contiennent les points rangés par ordre lexicographique de leurs coordonnées, ce qui implique qu'ils soient rangés par abscisses croissantes, et cela ne change pas la complexité de l'opération.

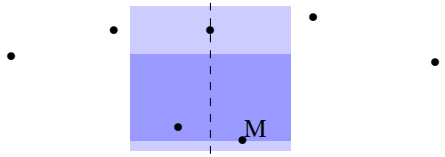
Puis on utilise le paradigme « diviser pour régner » :

- Si le nuage contient trois points ou moins, on calcule explicitement toutes les distances, et on en extrait le minimum ;
- Sinon, on sépare le nuage de points en deux ensembles $\mathcal{E}_g$ et $\mathcal{E}_d$ de respectivement $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$ points, séparés par une droite verticale [21] d'abscisse x_d. Puisque les points ont été ordonnés par abscisse croissante dans $\mathcal{P}(i)$, cette opération est en $\Theta(n)$ (elle serait même en $\Theta(1)$ si l'on utilisait des tableaux à la place des listes).
- Trois possibilités existent alors : les plus proches voisins sont tous deux dans $\mathcal{E}_g$, ils sont tous deux dans $\mathcal{E}_d$, ou bien l'un est dans $\mathcal{E}_g$ et l'autre dans $\mathcal{E}_d$. On va donc envisager les trois possibilités.
  - On détermine récursivement δ_g la plus courte distance entre deux points de $\mathcal{E}_g$ et δ_d celle entre deux points de $\mathcal{E}_d$, et on pose $\delta = \min(\delta_g, \delta_d)$.
  - Si la troisième possibilité donne une paire de points strictement plus proches que δ, leurs abscisses strictement comprise entre $x_d - \delta$ et $x_d + \delta$. On dresse une liste $\mathcal{P}''$ des points de cette bande, ordonnés par abscisses croissantes. Puisque l'on dispose de $\mathcal{P}'$, la construction de $\mathcal{P}''$ peut être faite en temps linéaire.



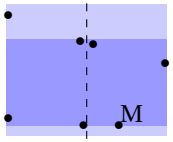
Enfin, on prends les points de $\mathcal{P}''$ dans l'ordre. L'objectif est simplement de savoir si un couple de points de $\mathcal{P}''$ ont une distance inférieure à δ. Si c'est le cas, ils sont nécessairement de part et d'autre de la droite $x = x_d$.

Un point M de $\mathcal{P}''$ ne peut avoir une distance inférieure à δ qu'avec un point P de $\mathcal{P}''$ vérifiant par ailleurs $y_M - \delta < y_P < y_M + \delta$. Puisque l'on prend les points par abscisses croissantes, l'éventuel cas $y_P < y_M$ aura déjà été traité, cela se réduit à s'intéresser aux points P de $\mathcal{P}''$ vérifiant $y_M \leqslant y_P < y_M + \delta$.



21. Si plusieurs points ont pour abscisse x_d, on les sépare en fonction de leur ordonnée, de façon à avoir deux moitiés équilibrées, d'où l'intérêt de trier les points dans $\mathcal{P}$ par ordre lexicographique.

Les points vérifiant ces deux inégalités figurent nécessairement parmi les *cinq* points suivant M dans la liste $\mathcal{P}''$, car les points d'un même côté de la droite sont écartés d'au moins δ. Quelle que soit la disposition des points dans le plan, on traitera donc moins de $5n$ couples dans cette dernière étape.



— Il ne reste qu'à clore l'algorithme en comparant la distance δ à la plus petite distance obtenue dans cette dernière étape de l'algorithme.

Pour déterminer la complexité de cet algorithme, on peut noter u_n le nombre de calculs de distance effectués pour un ensemble de n points. Pour $n \geqslant 4$, on peut écrire

$$u_n = u_{\lfloor n/2 \rfloor} + u_{\lceil n/2 \rceil} + b_n$$

où (b_n) représente le nombre de couples de points examinés dans la bande autour de la droite $x = x_d$, d'où $(b_n) = \Theta(n)$. On a donc $a_1 = a_2 = 1$, donc $\alpha = 1$, ce qui conduit à $(u_n) = \Theta\big(n\log(n)\big)$.

Puisque les précalculs de $\mathcal{P}$ et $\mathcal{P}'$, effectués une fois pour toute, sont également en $\Theta\big(n\log(n)\big)$ et que la construction de $\mathcal{P}''$, linéaire, a une complexité similaire à celle de (b_n), ces opérations ne changent pas la complexité générale de l'algorithme.

On peut donc ainsi obtenir la plus courte distance entre deux points (et éventuellement les deux points qui réalisent cette distance) avec une complexité en temps $\Theta\big(n\log(n)\big)$.

À titre d'illustration, nous allons construire une implémentation possible de cet algorithme en Caml. Pour ce faire, il nous faut construire plusieurs fonctions.

Tout d'abord, il nous faudra ordonner les points du plan par ordre lexicographique (pour $\mathcal{P}$), et par ordonnée croissante (pour $\mathcal{P}'$) en $O\big(\log(n)\big)$. Pour ce faire, nous allons réutiliser un tri fusion.

Pour éviter d'écrire deux tris distincts, nous allons réécrire notre fonction de tri de sorte qu'elle accepte en argument supplémentaire une fonction `foo` permettant de préciser que deux éléments `x` et `y` doivent être ordonnés selon les valeurs de `foo x` et `foo y`. On pourra ainsi les trier par abscisses croissantes en utilisant la fonction `snd`, et par ordre lexicographique en utilisant l'identité. La fonction `scinde` ne change pas :

```
# let rec scinde = function
    | t1::t2::q -> let q1, q2 = scinde q in t1::q1, t2::q2
    | lst       -> lst, [];;

val scinde : 'a list -> 'a list * 'a list = <fun>
```

En revanche, `fusionne` et `tri` prennent `foo` en paramètre :

```
# let rec fusionne foo l1 l2 = match (l1, l2) with
    | (t1::q1), (t2::q2) when foo t1 <= foo t2
                                -> t1::(fusionne foo q1 (t2::q2))
    | l1,       (t2::q2)        -> t2::(fusionne foo l1 q2)
    | l1,       []              -> l1;;

val fusionne : ('a -> 'b) -> 'a list -> 'a list -> 'a list = <fun>

# let rec tri foo lst = match scinde lst with
    | lst, [] -> lst
    | l1,  l2 -> fusionne foo (tri foo l1) (tri foo l2);;

tri : ('a -> 'b) -> 'a list -> 'a list = <fun>
```

Il nous faut une fonction prenant une liste de n points ordonnés lexicographiquement et retournant deux listes de longueurs respectives $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$, avec un ordre lexicographique *inverse* :

```
# let partition lst =
    let rec partitionAux pool lst1 = function
      | 0 -> lst1, (List.rev pool)
      | n -> partitionAux (List.tl pool) ((List.hd pool)::lst1) (n-1)
    in partitionAux lst [] ((List.length lst + 1) / 2);;

val partition : 'a list -> 'a list * 'a list = <fun>
```

Une fonction construisant la liste $\mathcal{P}''$ à partir de $\mathcal{P}'$ et des paramètres de la bande [22] :

```
let rec filtre pmin pmax x delta = function
    | []   -> []
    | t::q when (t >= pmin && t <= pmax && (fst t) > (x-.delta)
                                        && (snd t) < (x+.delta))
           -> t::(filtre pmin pmax x delta q)
    | _::q ->    (filtre pmin pmax x delta q);;

val filtre : float * float -> float * float -> float -> float ->
               (float * float) list -> (float * float) list = <fun>
```

22. On fournit également le plus « petit » point et le plus « grand » point (pour l'ordre lexicographique) de $\mathcal{E}_g \cup \mathcal{E}_d$ pour que les points de la bande soient bien uniquement des points de $\mathcal{E}_g \cup \mathcal{E}_d$, car si les points sont par exemple tous alignés le long d'une droite parallèle à l'axe des ordonnées, on courrait le risque d'obtenir systématiquement la totalité des points du nuage dans la bande!

Une fonction calculant la distance entre deux points (couples) :

```
# let dist pt1 pt2 =
  sqrt ((fst pt1 -. fst pt2)**2. +. (snd pt1 -. snd pt2)**2.);;

val dist : float * float -> float * float -> float = <fun>
```

Une fonction prenant un paramètre δ et une liste $\mathcal{P}''$ de points à l'intérieur d'une bande de largeur 2δ (les points étant ordonnés dans la liste passée en argument par abscisse croissante), et retournant le minimum entre δ et la plus petite des distances entre deux points de la bande [23] :

```
let minBande delta = function
    | []    -> delta              (* = delta si la liste est vide   *)
    | _::[] -> delta              (* = delta s'il y a un seul point *)
    | p1::q -> let pool = ref q             (* points à examiner    *)
               and i = ref 0                (* index dans la bande  *)
               and mini = ref delta         (* plus petite distance *)
               and tab = Array.make 5 p1 in (* -> 5 derniers points *)
                 while !pool <> [] do
                   let np = List.hd !pool in
                     for j = !i downto (max 0 (!i-4)) do
                       mini := min !mini (dist np tab.(j mod 5))
                     done;
                     i := !i+1;
                     tab.(!i mod 5) <- np;  (* On garde le point    *)
                     pool := List.tl !pool; (* pour la suite...     *)
                 done;
                 !mini;;

val minBande : float -> (float * float) list -> float = <fun>
```

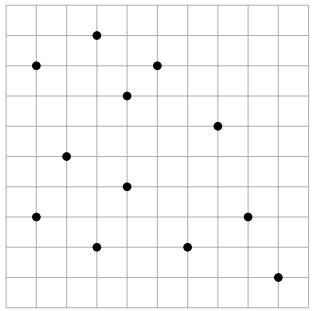
23. En profitant bien évidemment du fait qu'on ne doit au plus considérer que 5 voisins pour chaque point ; plutôt que de calculer les distances avec les 5 points suivants, on préfère ici déterminer les distances avec les 5 points *précédents* dans la liste $\mathcal{P}''$, le tableau `tab` contenant, à tout instant, les cinq précédents points examinés.

Et enfin, l'algorithme proprement dit qui utilise les fonctions précédemment définies :

```
# let minDist lst =
    let p = tri (function x -> x) lst
    and pprime = tri snd lst in
      let rec aux = function
        (* Terminaisons *)
        | []             -> failwith "Pas assez de points"
        | _::[]          -> failwith "Pas assez de points"
        | p1::p2::[]     -> dist p1 p2
        | p1::p2::p3::[] -> min (min (dist p1 p2) (dist p1 p3))
                                (dist p2 p3)

        (* Récursions *)
        | lst -> let l1, l2 = partition lst in   (* Appels récursifs *)
                 let dg = aux (List.rev l1)     (* sur les moitiés  *)
                 and dd = aux (List.rev l2) in (* du nuage         *)
                   let x = fst (List.hd l1)     (* x du pt milieu   *)
                   and delta = min dd dg
                   and pmin = List.hd lst       (* premier point    *)
                   and pmax = List.hd l2 in     (* dernier point    *)
                     let bande = filtre pmin pmax x delta pprime in
                       minBande delta bande
    in aux p;;

val minDist : (float * float) list -> float = <fun>
```

Terminons avec un test :

```
# minDist [ 1.,3.; 4.,7.; 6.,2.; 9.,1.; 8.,3.; 3.,2.;
            1.,8.; 3.,9.; 4.,4.; 5.,8.; 2.,5.; 7.,6. ];;

- : float = 1.4142135623730951
```



7 Logique propositionnelle

1 Propositions logiques

1.1 Introduction

Comme en mathématiques, la logique très naturellement a un rôle majeur en informatique. Il n'est en effet pas rare d'être amené à devoir prouver, par exemple, la correction ou la terminaison d'un programme, ce qui nécessitera un cadre logique rigoureux.

L'objectif de ce cours est de poser quelques bases de raisonnements logiques, permettant notamment la formalisation de propositions logiques formulées en langage naturel. Avant de nous lancer, efforçons nous d'illustrer le type de logique que l'on s'efforcera d'étudier dans ce cours, afin d'éviter d'éventuels malentendus. Nous ne nous intéresserons ici qu'à la logique appelée *logique propositionnelle*.

On qualifiera de *variable propositionnelle*[1] un énoncé qui est, sans ambiguité, soit *vrai*, soit *faux*. On les représente typiquement par des lettres majuscules (cursives dans ce cours). Par exemple, les énoncés suivants qui, au moment présent, sont soit vrai, soit faux, sont de possibles variables propositionnelles :

$\mathscr{A}$ – il pleut ;
$\mathscr{B}$ – je n'ai pas de parapluie ;
$\mathscr{C}$ – je suis mouillé.

Il est possible de combiner ces variables propositionnelles avec des *connecteurs logiques* pour construire des *formules propositionnelles*. Par exemple, « $\mathscr{A}$ *et* $\mathscr{B}$ », *il pleut et je n'ai pas de parapluie*, est une formule propositionnelle. De même que « $\mathscr{A}$ *ou* $\mathscr{C}$ », *il pleut ou je suis mouillé*. Ces formules propositionnelles sont également soit vraies, soit fausses. Par exemple, s'il ne pleut pas, que je n'ai pas de parapluie et que je suis mouillé, la première formule propositionnelle est fausse et la seconde est vraie. On peut également nier une variable propositionnelle : « *non* $\mathscr{A}$ » correspond ainsi à l'énoncé logique *il ne pleut pas*.

Les formules ainsi obtenues peuvent à leur tour être combinées grâce aux connecteurs logiques pour former d'autres formules propositionnelles plus complexes.

1. Le terme « variables » dans un cadre informatique, peut prêter à confusion : il n'y a rien de « variable » à proprement parler, cela fait simplement ici référence aux énoncés, vrais ou faux, servant de base à la construction des formules.

Attention, on ne s'intéresse pas, dans le cadre de la logique propositionnelle, aux relations possibles (de cause à effet par exemple) entre les différents énoncés logiques. Je peux très bien être mouillé tout en ayant mon parapluie, ou être mouillé même s'il ne pleut pas.

Des affirmations logiques telles que « *quel que soit le temps, si j'ai un parapluie, je ne suis pas mouillé* » appartiennent au domaine de la *logique des prédicats*, qui sort du cadre de ce cours. En particulier, nous ne ferons pas intervenir dans ce cours les quantifieurs que l'on trouve dans les démonstrations mathématiques, tels que « *quel que soit* » ou « *il existe* », qui appartiennent à cette logique des prédicats.

On verra cependant apparaître dans ce cours la notion logique d'*implication* et d'*équivalence*, dans une utilisation toutefois subtilement différente de ce qu'elle est usuellement en mathématiques. Par exemple, l'énoncé « $\mathscr{A}$ *et* $\mathscr{B} \to \mathscr{C}$ » peut très bien être faux : par exemple s'il pleut, que j'ai mon parapluie et que je suis mouillé. Si je veux pouvoir exprimer que, quel que soit le temps, si j'ai mon parapluie, je ne suis pas mouillé, c'est du domaine de la logique des prédicats.

Cela ne veut pas dire que l'on ne puisse pas obtenir des résultats logiques à partir de la seule logique propositionnelle : si les trois énoncés « $\mathscr{A}$ *et* $\mathscr{B} \to \mathscr{C}$ », « $\mathscr{A}$ » et « *non* $\mathscr{C}$ » sont tous trois vrais, alors nous verrons qu'il est possible d'en déduire que l'énoncé « $\mathscr{B}$ » est faux (j'ai mon parapluie). En effet, si « $\mathscr{B}$ » était vrai, au moins l'un des trois énoncés précédent serait également faux.

1.2 Définition

Définition. Formellement, on construit une *formule propositionnelle*[a] par induction structurelle à partir
- de deux constantes $\top$ et $\bot$ (faisant respectivement référence à quelque chose de « toujours vrai » et « toujours faux ») ;
- d'un ensemble $\mathcal{V} = \{\mathscr{A}, \mathscr{B}, \ldots\}$ fini ou dénombrable de *variables propositionnelles* ;
- des constructeurs binaires (qualifiés de *connecteurs logiques*) de *conjonction*, noté $\wedge$, de *disjonction*, noté $\vee$, d'*implication*, noté $\to$, et d'*équivalence*, noté $\leftrightarrow$, ainsi que d'un constructeur unaire de *négation*, noté $\neg$.

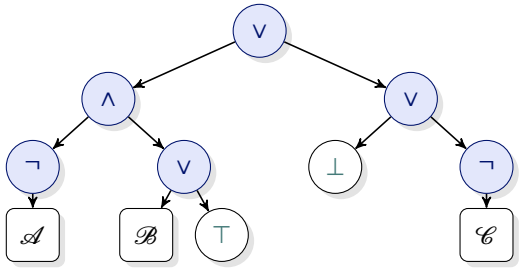
a. On parle également de *proposition logique*, de *formule logique* ou d'*expression logique*, les termes et notations peuvent varier d'un ouvrage à l'autre.

Nous le verrons, les connecteurs logiques de conjonction et de disjonctions sont liés aux notions de « et logique » et de « ou[2] logique ». Il est donc d'usage de lire $\wedge$ et $\vee$ respectivement « et » et « ou ». De même, la négation sera généralement lue « non ». Nous reviendrons sur la signification donnée aux opérateur d'implication et d'équivalence un peu plus loin.

2. Il s'agit ici d'un ou *inclusif*, « $\mathscr{A}$ ou $\mathscr{B}$ » étant vrai également lorsque $\mathscr{A}$ et $\mathscr{B}$ sont tous deux vrais. Le « ou » en langage courant pouvant être tantôt inclusif, tantôt exclusif.

1.3 Représentation arborescente

Il est naturel de représenter une proposition logique par un arbre binaire, où les constantes $\top$ et $\bot$ et variables propositionnelles se retrouvent dans les feuilles, et les connecteurs logiques binaires et unaires font office de nœuds internes. Par exemple :



Précisons que, même si ce n'est pas le cas sur cet exemple, une même variable propositionnelle peut apparaître à plusieurs endroits dans l'arbre, ou bien ne pas apparaître du tout. Il en est évidemment de même pour $\top$ et $\bot$[3].

Il en découle naturellement une implémentation en langage OCaml d'un type permettant de représenter des propositions logiques :

```
# type 'a proposition =
    | Vrai        (* pour représenter ⊤ *)
    | Faux        (* pour représenter ⊥ *)
    | Var of 'a   (* pour une variable propositionnelle *)
    | Conj of 'a proposition * 'a proposition   (* ∧ *)
    | Disj of 'a proposition * 'a proposition   (* ∨ *)
    | Impl of 'a proposition * 'a proposition   (* → *)
    | Equiv of 'a proposition * 'a proposition  (* ↔ *)
    | Neg of 'a proposition;;                  (* ¬ *)
```

Le type `'a` correspond au type utilisé pour identifier les différentes variables. Puisque nous représentons les variables propositionnelles par des lettres, on utilisera généralement dans la suite des caractères. Notre arbre d'exemple se déclare donc en OCaml de la sorte :

```
Disj (Conj (Neg (Var 'A'), Disj (Var 'B', Vrai)),
      Disj (Faux, Neg (Var 'C')));;
```

Définition. La *hauteur* et la *taille* d'une proposition logique correspondent respectivement à la hauteur et la taille de l'arbre qui lui est associé.

3. Nous verrons un peu plus tard que $\top$ et $\bot$ sont même inutiles, excepté dans les cas particuliers où l'arbre est réduit à une feuille $\top$ ou à une feuille $\bot$.

La formule propositionnelle nous servant d'exemple a donc une hauteur égale à 3 et une taille égale à 11.

On peut donc déterminer la hauteur d'une formule propositionnelle en OCaml comme on a pu le faire précédemment dans le cas d'un arbre binaire (et il en serait de même pour sa taille) :

```
# let rec hauteur = function
    | Vrai | Faux | Var _
        -> 0
    | Disj (a, b) | Conj (a, b) | Impl (a, b) | Equiv (a, b)
        -> 1 + max (hauteur a) (hauteur b)
    | Neg a
        -> 1 + hauteur a;;

val hauteur : 'a proposition -> int = <fun>
```

1.4 Expressions parenthésées

Représenter une formule propositionnelle par un arbre n'est pas toujours commode. On préfère souvent, comme il est d'usage en mathématiques, utiliser une écriture infixe de l'arbre, en employant systématiquement des parenthèses pour éviter les ambiguïtés.

Par exemple, la formule propositionnelle précédente s'écrira :

$$((\neg(\mathscr{A})) \wedge ((\mathscr{B}) \vee (\top))) \vee ((\bot) \vee (\neg(\mathscr{C})))$$

Il est possible de définir les formules propositionnelles directement sous cette forme par induction structurelle[4] :

Définition. Soit $\mathcal{V}$ un ensemble fini ou dénombrable de variables v_i

- « $\top$ » est une formule propositionnelle ;
- « $\bot$ » est une formule propositionnelle ;
- pour toute variable propositionnelle $v_i \in \mathcal{V}$, « v_i » est une formule propositionnelle ;
- si f est une formule propositionnelle, alors « $\neg(f)$ » est une formule propositionnelle ;
- si f et g sont deux formules propositionnelles, alors « $(f) \wedge (g)$ », « $(f) \vee (g)$ », « $(f) \to (g)$ » et « $(f) \leftrightarrow (g)$ » sont également des formules propositionnelles.

4. Les définitions sont équivalentes, cette écriture n'étant qu'une retranscription systématique et rigoureuse du parcours infixe de l'arbre.

Afin d'éviter l'abondance de parenthèses dans les formules propositionnelles, on convient généralement d'une priorité pour les constructeurs : $\neg$ est prioritaire sur $\wedge$, lui-même prioritaire sur $\vee$, à son tour prioritaire sur $\rightarrow$ et $\leftrightarrow$[5]. En outre, les formules propositionnelles s'interprètent usuellement de la gauche vers la droite[6]. On peut alors ne pas écrire les parenthèses inutiles, ce que nous ferons dans la suite de ce cours. L'écriture de la formule propositionnelle précédente peut donc se simplifier en

$$\neg\mathscr{A} \wedge (\mathscr{B} \vee \top) \vee (\bot \vee \neg\mathscr{C})$$

1.5 Écriture de Łukasiewicz

Afin d'éviter la profusion de parenthèses dans certaines formules, le logicien polonais J. Łukasiewicz a proposé d'écrire les formules propositionnelles au moyen du parcours préfixe de l'arbre. De cette façon, il n'est pas besoin d'utiliser des parenthèses, l'arité des connecteurs logiques étant connue. Ainsi, pour l'exemple précédent, on écrirait :

$$\vee \ \wedge \ \neg \ \mathscr{A} \ \vee \ \mathscr{B} \ \top \ \vee \ \bot \ \neg \ \mathscr{C}$$

On remarquera que l'ordre des symboles correspond précisément à l'ordre dans lequel ils apparaissent dans la définition de la proposition en OCaml (ce qui est naturel dans la mesure où les contructeurs en OCaml précèdent leurs arguments), même si la syntaxe OCaml requiert l'utilisation de parenthèses.

Bien qu'elle présente des avantages en terme de notation et d'utilisation, cette notation n'est plus guère utilisée en logique. Elle a un pendant, dite *notation polonaise inverse*, où l'on utilise le parcours *postfixe* de l'arbre.

2 Sémantique des propositions logiques

2.1 Distributions de vérité

Une *distribution de vérité* consiste à préciser, pour un ensemble de variables propositionnelles, lesquelles sont vraies et lesquelles sont fausses. Formellement :

Définition. Soit un ensemble fini $\mathcal{V}$ de n variables propositionnelles v_i.

On qualifie de *distribution de vérité* μ sur cet ensemble $\mathcal{V}$ une application de $\mathcal{V}$ dans $\mathcal{B}^n$, où $\mathcal{B} = \{\mathbf{V}, \mathbf{F}\}$ désigne l'ensemble des booléens.

5. Les priorités entre $\rightarrow$ et $\leftrightarrow$ sont mal définies, on préférera systématiquement utiliser des parenthèses dans ce cas pour éviter les ambiguïtés.

6. Ainsi, $\mathscr{A} \vee \mathscr{B} \vee \mathscr{C}$ correspond à $(\mathscr{A} \vee \mathscr{B}) \vee \mathscr{C}$, tandis que $\mathscr{A} \vee (\mathscr{B} \vee \mathscr{C})$ est une formule différente. Nous verrons toutefois qu'elles sont équivalentes.

Notons que la définition peut aisément être étendue au cas d'un ensemble $\mathcal{V}$ dénombrable. Le point important est que μ permette d'associer à toute variable propositionnelle v un booléen ($\mathbf{V}$ ou $\mathbf{F}$), et il n'est pas rare que l'on adopte une définition plus générale de μ qui ne se préoccupe pas précisément de la question de l'ensemble de définition.

S'il est relativement fréquent de noter, en français, les deux éléments de l'ensemble $\mathcal{B}$ des « booléens » (ou valeurs booléennes) $\mathbf{V}$ et $\mathbf{F}$, cette fois encore, cette notation peut varier d'un ouvrage à l'autre[7]. Le booléen $\mathbf{V}$ est associé à la notion de « vrai », tandis que le booléen $\mathbf{F}$ est associé à la notion de « faux ».

Lemme 3. *Il y a 2^n distributions de vérité sur un ensemble $\mathcal{V}$ de cardinal n.*

2.2 Évaluation

Définition. Soit μ une distribution de vérité sur un ensemble de variables $\mathcal{V}$.

L'*évaluation* associée à la distribution de vérité μ est l'application, notée $\mathcal{E}_\mu$ ou $[\![\mu]\!]$ de l'ensemble des formules propositionnelles sur $\mathcal{V}$ vers l'ensemble des booléens $\mathcal{B} = \{\mathbf{V}, \mathbf{F}\}$ définie par induction structurelle par :

- $\mathcal{E}_\mu(\top) = \mathbf{V}$;
- $\mathcal{E}_\mu(\bot) = \mathbf{F}$;
- pour tout $v_i \in \mathcal{V}$, $\mathcal{E}_\mu(v_i) = \mu(v_i)$;
- $\mathcal{E}_\mu(\neg f) = \mathbf{V}$ si $\mathcal{E}_\mu(f) = \mathbf{F}$, et $\mathcal{E}_\mu(\neg f) = \mathbf{F}$ sinon ;
- $\mathcal{E}_\mu(f \wedge g) = \mathbf{V}$ si $\mathcal{E}_\mu(f) = \mathcal{E}_\mu(g) = \mathbf{V}$, et $\mathcal{E}_\mu(f \wedge g) = \mathbf{F}$ sinon ;
- $\mathcal{E}_\mu(f \vee g) = \mathbf{F}$ si $\mathcal{E}_\mu(f) = \mathcal{E}_\mu(g) = \mathbf{F}$, et $\mathcal{E}_\mu(f \vee g) = \mathbf{V}$ sinon ;
- $\mathcal{E}_\mu(f \rightarrow g) = \mathbf{F}$ si $\mathcal{E}_\mu(f) = \mathbf{V}$ et $\mathcal{E}_\mu(g) = \mathbf{F}$, et $\mathcal{E}_\mu(f \rightarrow g) = \mathbf{V}$ sinon ;
- $\mathcal{E}_\mu(f \leftrightarrow g) = \mathbf{V}$ si $\mathcal{E}_\mu(f) = \mathcal{E}_\mu(g)$, et $\mathcal{E}_\mu(f \leftrightarrow g) = \mathbf{F}$ sinon.

Par exemple, pour la distribution de vérité $\mu = \{\mathscr{A} \mapsto \mathbf{V}, \mathscr{B} \mapsto \mathbf{F}, \mathscr{C} \mapsto \mathbf{V}\}$, l'évaluation de notre proposition logique $\neg\mathscr{A} \wedge (\mathscr{B} \vee \top) \vee (\bot \vee \neg\mathscr{C})$ donne $\mathbf{F}$.

Le comportement des constructeurs de conjonction, disjonction et négation justifient ici qu'on les appelle « et », « ou » et « non ». En effet, pour que $\mathscr{A} \wedge \mathscr{B}$ soit évalué à $\mathbf{V}$ (vrai), il faut que $\mathscr{A}$ *et* $\mathscr{B}$ le soient tous deux. Les opérateurs d'implication et d'équivalence ont un comportement qui est cohérent avec le sens qu'on leur donne usuellement en mathématiques, mais il faut prendre garde qu'ils ne représentent, dans le cadre de la logique propositionnelle, que des connecteurs logiques et non des articulations du raisonnement.

On peut obtenir très simplement une fonction OCaml évaluant une formule propositionnelle, la distribution de vérité étant fournie sous la forme d'une fonction de `'a` dans l'ensemble des booléens $\mathcal{B} = \{\mathbf{V}, \mathbf{F}\}$. On utilisera naturellement pour $\mathcal{B}$ le type `bool` du

7. Pour des raisons à la fois pratiques et liées à l'informatique, $\mathbf{V}$ est fréquemment noté $\mathbf{1}$ et $\mathbf{F}$, $\mathbf{0}$.

langage OCaml (et donc les constantes `true` et `false`), ce qui permettra d'utiliser les opérateurs booléens `not`, `&&` et `||` :

```
# let rec eval mu = function
    | Vrai -> true
    | Faux -> false
    | Var i -> mu i
    | Neg a -> not (eval mu a)
    | Conj (a, b) -> (eval mu a) && (eval mu b)
    | Disj (a, b) -> (eval mu a) || (eval mu b)
    | Impl (a, b) -> not (eval mu a) || (eval mu b)
    | Equiv (a, b) -> (eval mu a) = (eval mu b);;

val eval : ('a -> bool) -> 'a proposition -> bool = <fun>
```

2.3 Tables de vérité

Définition. La *table de vérité* d'une formule propositionnelle f est le tableau contenant son évaluation pour toutes les distributions de vérité μ possibles.

La table de vérité associée à notre proposition logique $\neg\mathscr{A} \wedge (\mathscr{B} \vee \top) \vee (\bot \vee \neg\mathscr{C})$ est :

$\mu(\mathscr{A})$	$\mu(\mathscr{B})$	$\mu(\mathscr{C})$	$\mathcal{E}_\mu(\neg\mathscr{A} \wedge (\mathscr{B} \vee \top) \vee (\bot \vee \neg\mathscr{C}))$
F	F	F	V
F	F	V	V
F	V	F	V
F	V	V	V
V	F	F	V
V	F	V	F
V	V	F	V
V	V	V	F

Pour construire une fonction OCaml prenant une liste d'identifiants de variables propositionnelles et une formule propositionnelle, et affichant la table de vérité correspondante, on peut utiliser un dictionnaire [8] [9] mémorisant les $\mu(v_i)$, afin de permettre l'évaluation de la proposition pour les différentes distributions de vérité à considérer.

8. Si les variables étaient indexées par des entiers, c'est-à-dire si le type `'a` correspond au type `int`, il serait plus simple d'utiliser un vecteur pour ce faire. Nous avons préservé ici la possibilité d'indexer les variables avec un type `'a` quelconque, la seule contrainte étant qu'il soit hachable.

9. Les instructions `Hashtbl.remove dico vi` ne sont pas indispensables, mais l'implémentation des dictionnaires en Caml mémorisant l'historique des associations, cela évite de gaspiller de la mémoire inutilement.

Cela donnerait par exemple [10] :

```
# let table vars expr =
    let dico = Hashtbl.create 97 in
    let mu = function vi -> Hashtbl.find dico vi in
    let rec aux pre = function
      | [] -> print_string pre;
              print_char (if (eval mu expr) then 'V' else 'F');
              print_newline ()
      | vi::q -> Hashtbl.add dico vi false;
                 aux (pre^"F ") q;
                 Hashtbl.remove dico vi;
                 Hashtbl.add dico vi true;
                 aux (pre^"V ") q;
                 Hashtbl.remove dico vi;
    in aux "" vars;;

val table : 'a list -> 'a proposition -> unit = <fun>
```

Les tables de vérité fournissent naturellement une manière alternative de décrire le comportement d'un connecteur logique dans le cadre de l'évaluation. .

Par exemple, les opérateurs de négation ¬, de conjonction ∧ et de disjonction ∨ sont associés aux tables de vérités suivantes [11] :

$\mathscr{A}$	$\neg\mathscr{A}$
F	V
V	F

$\mathscr{A}$	$\mathscr{B}$	$\mathscr{A} \wedge \mathscr{B}$
F	F	F
F	V	F
V	F	F
V	V	V

$\mathscr{A}$	$\mathscr{B}$	$\mathscr{A} \vee \mathscr{B}$
F	F	F
F	V	V
V	F	V
V	V	V

Ceux d'implication et d'équivalence correspondent aux tables suivantes :

$\mathscr{A}$	$\mathscr{B}$	$\mathscr{A} \to \mathscr{B}$
F	F	V
F	V	V
V	F	F
V	V	V

$\mathscr{A}$	$\mathscr{B}$	$\mathscr{A} \leftrightarrow \mathscr{B}$
F	F	V
F	V	F
V	F	F
V	V	V

10. Les concaténations de chaînes en série ne sont pas idéales en terme de complexité, mais pour une fonction effectuant un affichage avec un nombre limité de variables propositionnelles, cela n'est pas bien gênant.

11. On notera que les intitulés de colonnes ont été simplifiés pour en faciliter la lecture.

2.4 Équivalence logique

Définition. Deux formules propositionnelles f et g sont dites *(logiquement) équivalentes* si et seulement si leurs tables de vérités coïncident, c'est-à-dire si leurs évaluations sont les mêmes pour toute distribution de vérité μ.

L'équivalence logique de deux propositions logiques f et g est notée $f \equiv g$.

De façon évidente, $\neg\neg\mathscr{A}$ et $\mathscr{A}$ sont deux formules propositionnelles équivalentes. Mais attention, elles ne sont pas pour autant égales ! En effet, les arbres correspondant à ces propositions sont différents.

De même, les propositions logiques $(\mathscr{A} \wedge \mathscr{B}) \wedge \mathscr{C}$ et $\mathscr{A} \wedge (\mathscr{B} \wedge \mathscr{C})$ sont équivalentes, comme le prouvent les tables de vérité ci-dessous, mais ne sont pas égales.

$\mathscr{A}$	$\mathscr{B}$	$\mathscr{C}$	$\mathscr{A} \wedge \mathscr{B}$	$(\mathscr{A} \wedge \mathscr{B}) \wedge \mathscr{C}$
F	F	F	F	F
F	F	V	F	F
F	V	F	F	F
F	V	V	F	F
V	F	F	F	F
V	F	V	F	F
V	V	F	V	F
V	V	V	V	V

$\mathscr{A}$	$\mathscr{B}$	$\mathscr{C}$	$\mathscr{B} \wedge \mathscr{C}$	$\mathscr{A} \wedge (\mathscr{B} \wedge \mathscr{C})$
F	F	F	F	F
F	F	V	F	F
F	V	F	F	F
F	V	V	V	F
V	F	F	F	F
V	F	V	F	F
V	V	F	F	F
V	V	V	V	V

Théorème 6 (Principe de substitution)**.** *Soit $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ un ensemble de variables propositionnelles, et $\mathrm{F}_1(v_1, v_2, \ldots, v_n)$ et $\mathrm{F}_2(v_1, v_2, \ldots, v_n)$, deux formules propositionnelles équivalentes faisant intervenir tout ou partie de ces variables.*

Quelles que soient les formules propositionnelles $f_1, f_2, \ldots, f_n$, on a

$$\mathrm{F}_1(f_1, f_2, \ldots, f_n) \equiv \mathrm{F}_2(f_1, f_2, \ldots, f_n)$$

Ce résultat découle très naturellement des règles par induction structurelle régissant l'évaluation de formules propositionnelles, mais l'écriture rigoureuse de la preuve est un peu lourde, donc nous l'admettrons ici. Si les formules propositionnelles f_i font intervenir des variables propositionnelles différentes, l'équivalence se fait en considérant l'ensemble des variables propositionnelles figurant dans au moins une des formules f_i.

Théorème 7 (Lois de De Morgan)**.** *Si f et g sont deux propositions logiques, alors*

- $\neg f \wedge \neg g \equiv \neg(f \vee g)$;
- $\neg f \vee \neg g \equiv \neg(f \wedge g)$.

Démonstration. Pour la première loi, il suffit de vérifier que pour deux variables logiques $\mathscr{A}$ et $\mathscr{B}$, on a $\neg\mathscr{A} \wedge \neg\mathscr{B} \equiv \neg(\mathscr{A} \vee \mathscr{B})$ en comparant les tables de vérité :

$\mathscr{A}$	$\mathscr{B}$	$\neg\mathscr{A} \wedge \neg\mathscr{B}$
F	F	V
F	V	F
V	F	F
V	V	F

$\mathscr{A}$	$\mathscr{B}$	$\neg(\mathscr{A} \vee \mathscr{B})$
F	F	V
F	V	F
V	F	F
V	V	F

Puis d'utiliser le principe de substitution pour le généraliser à deux propositions logiques quelconques f et g.

Il en est de même pour la seconde loi. □

Lemme 4. *Soit f une formule propositionnelle.*

L'une des propositions suivantes est nécessairement vraie :

- $f \equiv \top$;
- $f \equiv \bot$;
- *il existe une formule propositionnelle g ne faisant intervenir ni $\top$ ni $\bot$ telle que $f \equiv g$*

Démonstration. Une façon triviale de parvenir à ce résultat est de remarquer les équivalences $\top \equiv v \vee \neg v$ et $\bot \equiv v \wedge \neg v$, de sorte qu'il est de toute façon toujours possible de construire $\top$ et $\bot$ à partir d'une variable quelconque. Cette solution rend toutefois la proposition logique plus complexe, or avec à peine plus de travail, on peut éliminer les $\bot$ et $\top$ en *simplifiant* la proposition.
Grâce au principe de substitution, on montre aisément que, pour toute formule propositionnelle f,

- $f \vee \top \equiv \top$ et $f \vee \bot \equiv f$ (et de même $\top \vee f \equiv \top$ et $\bot \vee f \equiv f$) ;
- $f \wedge \top \equiv f$ et $f \wedge \bot \equiv \bot$ (et de même $\top \wedge f \equiv f$ et $\bot \wedge f \equiv \bot$) ;
- $f \to \top \equiv \top$, $f \to \bot \equiv \neg f$, $\top \to f \equiv f$ et $\bot \to f \equiv \top$;
- $f \leftrightarrow \top \equiv f$ et $f \leftrightarrow \bot \equiv \neg f$ (et de même $\top \leftrightarrow f \equiv f$ et $\bot \leftrightarrow f \equiv \neg f$) ;

On procède ensuite par induction structurelle : pour toute proposition logique f,

- si $f \equiv \top$ ou $f \equiv \bot$, on en a terminé ;
- si $f = \neg f'$, on s'intéresse à f' :
  - si $f' \equiv \top$, alors $f \equiv \bot$;
  - si $f' \equiv \bot$, alors $f \equiv \top$;
  - si $f' \equiv g'$ où g' ne fait intervenir ni $\top$ ni $\bot$, il suffit de choisir $g = \neg g'$.
- si $f = f_1 \wedge f_2$, $f = f_1 \vee f_2$, $f = f_1 \to f_2$ ou $f = f_1 \leftrightarrow f_2$, on détermine deux équivalents à f_1 et f_2 satisfaisant aux conditions, et on applique les équivalences précédentes si les équivalents de f_1 ou de f_2 sont $\top$ ou $\bot$. □

Par exemple, notre proposition logique $\neg\mathscr{A} \wedge (\mathscr{B} \vee \top) \vee (\bot \vee \neg\mathscr{C})$ est équivalente à la

proposition logique $\neg\mathscr{A} \vee \neg\mathscr{C}$ (soit encore $\neg(\mathscr{A} \wedge \mathscr{C})$ d'après les lois de De Morgan), ce que l'on retrouve d'ailleurs dans la table de vérité précédemment écrite.

Ce résultat relativise l'utilité de $\top$ et $\bot$ dans les formules propositionnelles : on peut toujours trouver une formule équivalente qui ne les fait pas intervenir [12].

2.5 Autres connecteurs binaires

Il est possible de construire $2^4 = 16$ connecteurs logiques binaires distincts, c'est-à-dire qui donnent, pour deux arguments $\mathcal{A}$ et $\mathcal{B}$, des formules propositionnelles qui ne sont pas équivalentes, dont six n'ont pas d'intérêt : deux ont des évaluations qui ne dépendent ni de l'évaluation de $\mathscr{A}$, ni de celle de $\mathscr{B}$ (ils sont équivalents à $\top$ et $\bot$), et quatre autres ne dépendent que de celle d'un seul des deux arguments (ils sont équivalents à $\mathscr{A}$, $\neg\mathscr{A}$, $\mathscr{B}$ et $\neg\mathscr{B}$).

Nous avons d'ores et déjà rencontré quatre de ces connecteurs logiques : conjonction, disjonction, implication et équivalence. Quatre autres connecteurs logiques binaires sont plus rarement utilisés en mathématiques et en logique, mais beaucoup plus souvent en informatique et en électronique : les connecteurs logiques « *ou exclusif* » (noté $\oplus$), « *non-et* » (noté NAND), « *non-ou* » (noté NOR) et « *inhibition* » (noté INH). On peut les définir par équivalence [13] :

- $\mathscr{A} \oplus \mathscr{B} \equiv (\mathscr{A} \wedge \neg\mathscr{B}) \vee (\neg\mathscr{A} \wedge \mathscr{B})$ (ou $\neg(\mathscr{A} \leftrightarrow \mathscr{B})$) ;
- $\mathscr{A}$ NAND $\mathscr{B} \equiv \neg(\mathscr{A} \wedge \mathscr{B})$;
- $\mathscr{A}$ NOR $\mathscr{B} \equiv \neg(\mathscr{A} \vee \mathscr{B})$;
- $\mathscr{A}$ INH $\mathscr{B} \equiv \mathscr{A} \wedge \neg\mathscr{B}$ (ou $\neg(\mathscr{A} \rightarrow \mathscr{B})$).

Ou bien via leurs tables de vérité :

$\mathscr{A}$	$\mathscr{B}$	$\mathscr{A} \oplus \mathscr{B}$	$\mathscr{A}$ NAND $\mathscr{B}$	$\mathscr{A}$ NOR $\mathscr{B}$	$\mathscr{A}$ INH $\mathscr{B}$
F	F	F	V	V	F
F	V	V	V	F	F
V	F	V	V	F	V
V	V	F	F	F	F

On remarquera qu'il s'agit de la négation des quatres connecteurs déjà étudiés. Parmi ces huit connecteurs binaires, deux (le connecteur d'implication et celui d'inhibition) ne sont pas commutatifs. Les deux connecteurs binaires restants correspondent simplement à la permutation de leurs arguments.

12. Excepté éventuellement dans le cas d'une formule toujours vraie ou toujours fausse, encore que si l'on dispose d'une variable propositionnelle $\mathscr{A}$ quelconque, on peut utiliser $\top \equiv \mathscr{A} \vee \neg\mathscr{A}$ et $\bot \equiv \mathscr{A} \wedge \neg\mathscr{A}$!

13. Dans le cas du connecteur d'inhibition, il arrive que le rôle des deux arguments soit inversé, l'écriture utilisée ici n'est pas universelle.

2.6 Systèmes complets

Si disposer de nombreux connecteurs logiques permet d'exprimer plus simplement une formule propositionnelle donnée, on peut se passer de la plupart d'entre eux, grâce notamment aux équivalences vues précédemment.

Définition. Un ensemble de connecteurs logiques forme un *système complet* si, pour toute formule propositionnelle sur un ensemble de variables propositionnelles $\mathcal{V}$, il est possible d'écrire une formule équivalente avec ces seuls connecteurs.

Lemme 5. $\{\wedge, \neg\}$ *et* $\{\vee, \neg\}$ *sont des systèmes complets.*

Démonstration. Pour vérifier que $\{\wedge, \neg\}$ est un système complet, il suffit de constater, grâce aux lois de De Morgan, que $\mathscr{A} \vee \mathscr{A}$ est équivalent à $\neg(\neg\mathscr{A} \wedge \neg\mathscr{A})$.

Nous avons déjà précisé que $\rightarrow$ et $\leftrightarrow$ pouvaient être remplacées par des combinaisons de $\neg$, $\wedge$ et $\vee$. Pour n'importe quelle formule propositionnelle f, on peut ainsi construire une formule propositionnelle f' équivalente à f à partir des seuls connecteurs $\wedge$ et $\neg$.

Il en est de même pour $\{\wedge, \neg\}$: $\mathscr{A} \wedge \mathscr{B}$ est en effet équivalent à $\neg(\neg\mathscr{A} \vee \neg\mathscr{B})$. □

Lemme 6. $\{NAND\}$ *et* $\{NOR\}$ *sont des systèmes complets.*

Démonstration. Pour montrer que NAND constitue, à lui seul, un système complet, il suffit de voir que $\neg\mathscr{A}$ est équivalent à $\mathscr{A}$ NAND $\mathscr{A}$, et que $\mathscr{A} \wedge \mathscr{B}$ est équivalent à $(\mathscr{A}$ NAND $\mathscr{B})$ NAND $(\mathscr{A}$ NAND $\mathscr{B})$.

Puisque $\{\wedge, \neg\}$ est un système complet, $\{NAND\}$ l'est également.

De même, pour $\{NOR\}$, on remarque que $\neg\mathscr{A}$ est équivalent à $\mathscr{A}$ NOR $\mathscr{A}$ et $\mathscr{A} \wedge \mathscr{B}$ à $(\mathscr{A}$ NOR $\mathscr{A})$ NOR $(\mathscr{B}$ NOR $\mathscr{B})$. □

NAND étant à lui seul un système complet, il s'agit de la porte logique la plus courante en électronique, tout circuit logique pouvant être construit à partir de cette seule porte, y compris des cellules mémoire.

Outre cet intérêt évident dans le domaine de l'électronique, les systèmes complets permettent également de simplifier certaines démonstrations. Par exemple, nous avons montré précédemment que, pour toute formule propositionnelle qui n'était équivalente ni à $\top$, ni à $\bot$, on pouvait trouver une formule propositionnelle équivalente ne faisant intervenir ni $\top$, ni $\bot$. On peut simplifier cette démonstration en commençant par éliminer tous les connecteurs logique exceptés par exemple $\wedge$ et $\neg$. Une fois ceci fait, il reste beaucoup moins de cas à analyser dans la démonstration !

Usuellement, un système dit complet ne nécessite pas l'utilisation des constantes $\top$ et $\bot$. Mais on peut généraliser les choses et construire des systèmes complets basés sur un ou plusieurs constructeur(s) et l'une ou l'autre des constantes $\bot$ et $\top$.

Par exemple, $\{\to, \bot\}$ est un système complet. En effet, on a $((\mathscr{A} \to \bot) \to \mathscr{B}) \to \bot \equiv \mathscr{A}$ NOR $\mathscr{B}$. {NOR} étant un système complet, il en est de même pour $\{\to, \bot\}$.

$\mathscr{A}$	$\mathscr{B}$	$\mathscr{A} \to \bot$	$(\mathscr{A} \to \bot) \to \mathscr{B}$	$((\mathscr{A} \to \bot) \to \mathscr{B}) \to \bot$
F	F	V	F	V
F	V	V	V	F
V	F	F	V	F
V	V	F	V	F

3 Satisfiabilité et déduction logique

3.1 Tautologies et antilogies

Définition. On qualifie de *tautologie* une formule propositionnelle qui est évaluée à V quelle que soit la distribution de vérité. On qualifie d'*antilogie* une formule propositionnelle qui est évaluée à F quelle que soit la distribution de vérité.

Par exemple, les formules $\mathcal{A} \vee \neg\mathcal{A}$, $\mathcal{A} \to \mathcal{A}$ ou bien $(\mathcal{A} \wedge \neg\mathcal{B}) \leftrightarrow (\neg(\neg\mathcal{A} \vee \mathcal{B}))$ sont des tautologies. Pour une propriété logique en général, cela peut se vérifier aisément en travaillant avec les tables de vérité. Par exemple, pour la troisième formule propositionnelle :

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A} \wedge \neg\mathcal{B}$	$\neg\mathcal{A} \vee \mathcal{B}$	$(\mathcal{A} \wedge \neg\mathcal{B}) \leftrightarrow (\neg(\neg\mathcal{A} \vee \mathcal{B}))$
F	F	F	V	V
F	V	F	V	V
V	F	V	F	V
V	V	F	V	V

3.2 Satisfiabilité

Définition. Une formule propositionnelle f est dite *satisfiable* s'il existe une distribution de vérité μ telle que son évaluation $\mathcal{E}_\mu(f)$ soit égale à V.

Une telle distribution est appelée un *modèle* de f.

Une antilogie est donc une formule propositionnelle qui n'est pas satisfiable, ou en d'autres termes une formule propositionnelle qui n'admet aucun modèle. Une tautologie est toujours satisfiable, puisque n'importe quelle distribution de probabilité μ convient.

Par exemple, la proposition logique $(\mathscr{A} \to \mathscr{B}) \wedge (\mathscr{B} \to \neg\mathscr{A})$ est satisfiable. En effet, la distribution de vérité μ telle que $\mu(\mathscr{B}) =$ V et $\mu(\mathscr{B}) =$ F convient.

3.3 Déduction logique

Définition. Soient deux formules propositionnelles f et g. On dit que g est une *conséquence* de f, et on note $f \vDash g$, lorsqu'il n'existe aucune distribution de vérité telle que g soit évaluée à faux et f à vrai. En d'autres termes, lorsque tout modèle de f est également un modèle de g.

f est qualifié de *prémisse* et g de *conclusion*.

Attention, $f \to g$ et $f \vDash g$ ne signifient pas du tout la même chose :

- $f \to g$ est une proposition logique, qui peut tout à fait être fausse (on peut écrire $\top \to \bot$, c'est une proposition logique fausse) ;
- $f \vDash g$ affirme qu'il est impossible que f soit fausse tandis que g est vraie.

Ainsi, $f \vDash g$ signifie que $f \to g$ est une tautologie.

La différence est la même entre le connecteur logique $\leftrightarrow$ et la notation $\equiv$ dénotant l'équivalence entre deux formules propositionnelles, et de fait, $f \equiv g$ sigifie que $f \leftrightarrow g$ est une tautologie.

Il est possible de se servir de cette notation pour décrire les propriétés d'une formule propositionnelle f. Par exemple, f est une tautologie si et seulement si $\top \vDash f$. De même, f est une antilogie si et seulement si $f \vDash \bot$ [14].

3.4 Conséquence d'un ensemble de formules

On peut étendre cette notion de conséquence logique à un *ensemble* de formules propositionnelles $\Gamma = \{f_1, f_2, \ldots\}$:

Définition. Soient un ensemble de formules propositionnelles $\Gamma = \{f_1, f_2, \ldots\}$ et une formule propositionnelle g. On dit que g est une conséquence de Γ, et on note $\Gamma \vDash g$, lorsqu'il n'existe aucune distribution de vérité telle que g soit évaluée à faux et que *toutes* formules f_i soient évaluées à vrai. En d'autres termes, lorsque tout modèle de l'*ensemble* des f_i est également un modèle de g.

Si l'ensemble Γ est un ensemble fini de n formules [15], $\Gamma \vDash g$ correspond simplement [16] à $f_1 \wedge f2 \wedge \ldots \wedge f_n \vDash g$.

$\top$ étant l'élément neutre pour $\wedge$, si l'on écrit « $\vDash f$ » (sans rien à gauche du symbole $\vDash$, soit $\Gamma = \emptyset$), on sous-entend $\top \vDash f$. Autrement dit, « $\vDash f$ » signifie que f est une tautologie.

14. Attention au sens de ces expressions ! On a toujours $f \vDash \top$ et $\bot \vDash f$ quelle que soit f.
15. Ce qui n'est pas forcément toujours le cas
16. On considère que $\vDash$ a ici la précédence la plus faible, de sorte que les parenthèses sont inutiles.

3.5 Un exemple d'utilisation

Reprenons l'exemple de l'introduction, avec les variables propositionnelles $\mathscr{A}$, $\mathscr{B}$ et $\mathscr{C}$ énoncées de la sorte :

$\mathscr{A}$ – il pleut ;
$\mathscr{B}$ – je n'ai pas de parapluie ;
$\mathscr{C}$ – je suis mouillé.

Considérons les trois formules propositionnelles suivantes :

$\mathscr{A} \wedge \mathscr{B} \to \mathscr{C}$ – s'il pleut et que je n'ai pas de parapluie, je suis mouillé ;
$\mathscr{A}$ – il pleut ;
$\neg\mathscr{C}$ – je ne suis pas mouillé.

Ces trois formules peuvent-elles être toutes les trois vraies simultanément, ou en d'autres termes admettent-elles un modèle commun ?

Cela revient à se poser la question de la satisfiabilité de la conjonction des trois formules propositionnelles, c'est-à-dire de $(\mathscr{A} \wedge \mathscr{B} \to \mathscr{C}) \wedge (\mathscr{A}) \wedge (\neg\mathscr{C})$.

Pour le savoir, une solution simple est de construire la table de vérité correspondante :

$\mathscr{A}$	$\mathscr{B}$	$\mathscr{C}$	$(\mathscr{A} \wedge \mathscr{B} \to \mathscr{C}) \wedge (\mathscr{A}) \wedge (\neg\mathscr{C})$
F	F	F	F
F	F	V	F
F	V	F	F
F	V	V	F
V	F	F	V
V	F	V	F
V	V	F	F
V	V	V	F

On constate qu'il existe une distribution de vérité (plus précisément la distribution de vérité $\mu = \{\mathscr{A} \mapsto \mathbf{V}, \mathscr{B} \mapsto \mathbf{F}, \mathscr{C} \mapsto \mathbf{F}\}$) pour laquelle l'évaluation de la conjonction des trois formules propositionnelle est évaluée à vrai. La conjonction des trois formules considérée est donc effectivement satisfiable.

On peut constater par ailleurs que cette distribution de vérité est unique, et qu'elle impose $\mu(\mathscr{B}) = \mathbf{F}$, autrement dit « j'ai un parapluie ».

On peut donc écrire $(\mathscr{A} \wedge \mathscr{B} \to \mathscr{C}) \wedge (\mathscr{A}) \wedge (\neg\mathscr{C}) \models \neg\mathscr{B}$.

Ou bien encore, de façon équivalente, $\{\mathscr{A} \wedge \mathscr{B} \to \mathscr{C}, \mathscr{A}, \neg\mathscr{C}\} \models \neg\mathscr{B}$.

4 Quelques propriétés logiques

4.1 Propriétés de $\wedge$

Théorème 8. *Soient f_1, f_2, f_3 trois formules propositionnelles quelconques.*

élément neutre : $\top \wedge f_1 \equiv f_1 \wedge \top \equiv f_1$
élément absorbant : $\bot \wedge f_1 \equiv f_1 \wedge \bot \equiv \bot$
idempotence : $f_1 \wedge f_1 \equiv f_1$
commutativité : $f_1 \wedge f_2 \equiv f_2 \wedge f_1$
associativité : $(f_1 \wedge f_2) \wedge f_3 \equiv f_1 \wedge (f_2 \wedge f_3)$

Ces propriétés sont assez immédiates, et se démontrent aisément en utilisant les tables de vérité et le principe de substitution. Précisons que les noms associés aux propriétés précédentes du connecteur logique $\wedge$ sont liés au vocabulaire des opérateurs. En effet, $\wedge$ (comme tout connecteur logique) peut être vu comme une loi de composition interne sur l'ensemble des booléens $\mathcal{B}$.

4.2 Propriétés de $\vee$

Théorème 9. *Soient f_1, f_2, f_3 trois formules propositionnelles quelconques.*

élément neutre : $\bot \vee f_1 \equiv f_1 \vee \bot \equiv f_1$
élément absorbant : $\top \vee f_1 \equiv f_1 \vee \top \equiv \top$
idempotence : $f_1 \vee f_1 \equiv f_1$
commutativité : $f_1 \vee f_2 \equiv f_2 \vee f_1$
associativité : $(f_1 \vee f_2) \vee f_3 \equiv f_1 \vee (f_2 \vee f_3)$

4.3 Propriétés mêlant $\wedge$ et $\vee$

Théorème 10. *Soient f_1, f_2, f_3 trois formules propositionnelles quelconques.*

subsomption : $f_1 \vee (f_1 \wedge f_2) \equiv f_1$
$f_1 \wedge (f_1 \vee f_2) \equiv f_1$
distributivité : $f_1 \wedge (f_2 \vee f_3) \equiv (f_1 \wedge f_2) \vee (f_1 \wedge f_3)$
$f_1 \vee (f_2 \wedge f_3) \equiv (f_1 \vee f_2) \wedge (f_1 \vee f_3)$
lois de De Morgan : $\neg(f_1 \vee f_2) \equiv \neg f_1 \wedge \neg f_2$
$\neg(f_1 \wedge f_2) \equiv \neg f_1 \vee \neg f_2$

4.4 Propriétés de $\leftrightarrow$

Théorème 11. *Soient f_1, f_2, f_3 trois formules propositionnelles quelconques.*

élément neutre :	$\top \leftrightarrow f_1 \equiv f_1 \leftrightarrow \top \equiv f_1$
négation :	$\bot \leftrightarrow f_1 \equiv f_1 \leftrightarrow \bot \equiv \neg f_1$
réflexivité :	$f_1 \leftrightarrow f_1 \equiv \top$
commutativité :	$f_1 \leftrightarrow f_2 \equiv f_2 \leftrightarrow f_1$
associativité :	$(f_1 \leftrightarrow f_2) \leftrightarrow f_3 \equiv f_1 \leftrightarrow (f_2 \wedge f_3)$

Le terme de *réflexivité* qualifie usuellement, en mathématique, des *relations*. Les connecteurs logiques binaires peuvent en effet être vus comme des relations entre formules propositionnelles, puisqu'une formules propositionnelles telle que $f \leftrightarrow g$, par exemple, sera vraie ou fausse.

La propriété de commutativité de $\leftrightarrow$ (considéré comme une loi de composition interne sur $\mathscr{B}$) peut donc être également vue comme une propriété de *symétrie* (en considérant $\leftrightarrow$ comme une relation sur $\mathscr{B}$).

On peut également ajouter une propriété, fréquemment utilisée dans les raisonnements, qui mélange $\leftrightarrow$ et $\wedge$:

Théorème 12. *Soient f_1, f_2, f_3 trois formules propositionnelles quelconques.*

transitivité :	$(f_1 \leftrightarrow f_2) \wedge (f_2 \leftrightarrow f_3) \vDash f_1 \leftrightarrow f_3$

Cela fait de $\leftrightarrow$ une relation d'équivalence sur l'ensemble des booléens $\mathcal{B}$ (d'où le nom donné au connecteur logique !)

4.5 Propriétés de $\rightarrow$

Théorème 13. *Soient f_1, f_2, f_3 trois formules propositionnelles quelconques.*

réflexivité :	$f_1 \rightarrow f_1 \equiv \top$
antisymétrie :	$(f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1) \equiv f_1 \leftrightarrow f_2$
transitivité :	$(f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_3) \vDash f_1 \rightarrow f_3$

Ces propriétés confèrent à $\rightarrow$, lorsqu'on le considère comme une relation entre formules propositionnelles, un caractère de relation d'ordre.

Rappelons que la loi de composition interne associée à $\rightarrow$ n'est pas commutative, comme nous l'avons vu avec sa table de vérité. Elle n'est pas non plus associative ($(\bot \rightarrow \bot) \rightarrow \bot$ est par exemple évaluée à *faux*, tandis que $\bot \rightarrow (\bot \rightarrow \bot)$ est évaluée à *vrai*).

Là encore, les deux dernières propriétés exposées précédemment sont des idées utilisées fréquemment dans les raisonnements.

4.6 Propriétés de $\oplus$

Théorème 14. *Soient f_1, f_2, f_3 trois formules propositionnelles quelconques.*

élément neutre :	$\bot \oplus f_1 \equiv f_1 \oplus \bot \equiv f_1$
négation :	$\top \oplus f_1 \equiv f_1 \oplus \top \equiv \neg f_1$
nilpotence :	$f_1 \oplus f_1 \equiv \bot$
commutativité :	$f_1 \oplus f_2 \equiv f_2 \oplus f_1$
associativité :	$(f_1 \oplus f_2) \oplus f_3 \equiv f_1 \oplus (f_2 \oplus f_3)$

On remarquera par ailleurs que $(f_1 \oplus f_2) \oplus f_2 \equiv f_1$, ce qui se révèle assez fréquemment utile en électronique et en informatique.

4.7 Outils pour les raisonnements

Nous avons vu précédemment quelques propriétés logiques utiles pour le raisonnement. Il en existe de nombreuses autres. En voici quelques-unes :

Théorème 15. *Soient f_1, f_2, f_3 trois formules propositionnelles quelconques.*

tiers exclu :	$f_1 \vee \neg f_1 \equiv \top$
modus ponens :	$f_1 \wedge (f_1 \rightarrow f_2) \vDash f_2$
modus tollens :	$(f_1 \rightarrow f_2) \wedge \neg f_2 \vDash \neg f_1$
disjonction de cas :	$(f_1 \rightarrow f_2) \wedge (\neg f_1 \rightarrow f_2) \vDash f_2$
double implication :	$(f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1) \equiv (f_1 \leftrightarrow f_2)$
contraposition :	$(f_1 \rightarrow f_2) \equiv (\neg f_2 \rightarrow \neg f_1)$
raisonnement par l'absurde :	$(\neg f_1 \rightarrow \bot) \equiv f_1$
ex-falso quodlibet :	$\bot \vDash f_1$

Ces propriétés peuvent évidemment être combinées. Par exemple, de la double implication et de la contraposition, on peut en déduire que $(f_1 \rightarrow f_2) \wedge (\neg f_1 \rightarrow \neg f_2) \equiv (f_1 \leftrightarrow f_2)$.

Précisons que la dernière propriété ne présente guère d'intérêt pratique... Elle affirme simplement qu'une contradiction permet de démontrer n'importe quoi. Ce qui fait que l'on ne s'intéresse généralement qu'à des théories non-contradictoires, sinon n'importe quelle affirmation est vraie (de même que sa négation).

5 Formes normales et formes canoniques

5.1 Formes conjonctives et disjonctives

Pour faciliter la manipulation de propositions logiques, il est utile de pouvoir se ramener à une forme aussi simple et standard que possible. Les formes conjonctives et disjonctives remplissent cet office.

Définition. On appelle *littéral* toute formule propositionnelle de la forme v ou $\neg v$ où v est une variable propositionnelle ($\mathscr{A}$, $\mathscr{B}$, $\mathscr{C}$...)

On appelle *conjonction* des formules propositionnelles f_1, ..., f_n la formule propositionnelle $f_1 \wedge f_2 \wedge \ldots \wedge f_n$.

On appelle *disjonction* des formules propositionnelles f_1, ..., f_n la formule propositionnelle $f_1 \vee f_2 \vee \ldots \vee f_n$.

On qualifie de *forme conjonctive* une conjonction de disjonction de littéraux.

On qualifie de *forme disjonctive* une disjonction de conjonction de littéraux.

Par exemple :
- « $(\mathscr{A} \vee \mathscr{C}) \wedge (\mathscr{B} \vee \mathscr{C}) \wedge \mathscr{D}$ » une forme conjonctive ;
- « $\mathcal{A} \wedge \mathscr{B} \vee \mathscr{A} \wedge \mathscr{D} \vee \mathscr{B} \wedge \mathscr{C}$ » est une forme disjonctive [17].

Il s'agit en quelque sorte des équivalents des notions de « forme factorisée » et « forme développée » des propositions mathématiques.

5.2 Formes normales

Théorème 16. *Pour toute formule propositionnelle, on peut trouver une formule propositionnelle équivalente ayant une forme conjonctive, ainsi qu'une formule propositionnelle équivalente ayant une forme disjonctive.*

Définition. On dit qu'une proposition logique est sous *forme normale conjonctive* (FNC) lorsqu'elle est écrite sous une forme conjonctive, et sous *forme normale disjonctive* (FND) lorsqu'elle est écrite sous une forme disjonctive.

Démonstration. Pour démontrer l'affirmation précédente, le plus simple est d'exhiber une méthode permettant de construire des propositions logiques équivalentes sous forme conjonctive et disjonctive.

Si l'on considère la table de vérité associée à une proposition logique, il est trivial que de construire une disjonction de conjonction de littéraux équivalente à cette proposition logique : il suffit en effet de traduire les différentes lignes de la table correspondant à des résultats « vrais ».

Pour trouver une conjonction de disjonctions de littéraux équivalente à une proposition f, on détermine une disjonction de conjonction de littéraux équivalente à $\neg f$ (ce qui revient à dresser la liste des lignes de la table de vérité correspondant à des résultats « faux ») puis on utilise une loi de De Morgan pour transformer la forme disjonctive en forme conjonctive. □

Par exemple, pour la proposition logique $f = ((\mathscr{A} \oplus \mathscr{B}) \vee \mathscr{B} \wedge \neg\mathscr{C}) \wedge \mathscr{A}$, on construit la table de vérité :

$\mathscr{A}$	$\mathscr{B}$	$\mathscr{C}$	$((\mathscr{A} \oplus \mathscr{B}) \vee \mathscr{B} \wedge \neg\mathscr{C}) \wedge \mathscr{A}$
F	F	F	F
F	F	V	F
F	V	F	F
F	V	V	F
V	F	F	V
V	F	V	V
V	V	F	V
V	V	V	F

Cette table de vérité contient trois lignes pour lesquelles f est évaluée à vrai. La formule propositionnelle f est donc équivalente à la disjonction des trois conjonctions correspondantes :

$$\mathscr{A} \wedge \neg\mathscr{B} \wedge \neg\mathscr{C} \quad \vee \quad \mathscr{A} \wedge \mathscr{B} \wedge \neg\mathscr{C} \quad \vee \quad \mathscr{A} \wedge \neg\mathscr{B} \wedge \mathscr{C}\,.$$

Les cinq autres lignes correspondent aux modèles pour la négation de notre formule propositionnelle, $\neg f = \neg(((\mathscr{A} \oplus \mathscr{B}) \vee \mathscr{B} \wedge \neg\mathscr{C}) \wedge \mathscr{A})$. $\neg f$ est donc équivalente à la disjonction des cinq conjonctions suivantes :

$$\neg\mathscr{A} \wedge \neg\mathscr{B} \wedge \neg\mathscr{C} \quad \vee \quad \neg\mathscr{A} \wedge \mathscr{B} \wedge \neg\mathscr{C} \quad \vee \quad \neg\mathscr{A} \wedge \neg\mathscr{B} \wedge \mathscr{C} \quad \vee \quad \neg\mathscr{A} \wedge \mathscr{B} \wedge \mathscr{C} \quad \vee \quad \mathscr{A} \wedge \mathscr{B} \wedge \mathscr{C}\,.$$

Par conséquent, notre formule propositionnelle f est équivalente à :

$$\neg\Big(\neg\mathscr{A} \wedge \neg\mathscr{B} \wedge \neg\mathscr{C} \quad \vee \quad \neg\mathscr{A} \wedge \mathscr{B} \wedge \neg\mathscr{C} \quad \vee \quad \neg\mathscr{A} \wedge \neg\mathscr{B} \wedge \mathscr{C} \quad \vee \quad \neg\mathscr{A} \wedge \mathscr{B} \wedge \mathscr{C} \quad \vee \quad \mathscr{A} \wedge \mathscr{B} \wedge \mathscr{C}\Big).$$

En utilisant les lois de De Morgan sur le terme de droite, on en déduit que f est équivalente à la conjonction des cinq disjonctions suivantes :

$$(\mathscr{A} \vee \mathscr{B} \vee \mathscr{C}) \wedge (\mathscr{A} \vee \neg\mathscr{B} \vee \mathscr{C}) \wedge (\mathscr{A} \vee \mathscr{B} \vee \neg\mathscr{C}) \wedge (\mathscr{A} \vee \neg\mathscr{B} \vee \neg\mathscr{C}) \wedge (\neg\mathscr{A} \vee \neg\mathscr{B} \vee \neg\mathscr{C})\,.$$

Les formes conjonctive et disjonctive ainsi obtenues ne sont pas les seules possibles, la formule f est par exemple équivalente aux deux formes suivantes, plus succintes :

$$f \equiv \mathscr{A} \wedge \neg\mathscr{B} \quad \vee \quad \mathscr{A} \wedge \neg\mathscr{C} \equiv \mathscr{A} \wedge (\neg\mathscr{B} \vee \neg\mathscr{C})\,.$$

17. On rappelle que $\wedge$ est prioritaire sur $\vee$.

5.3 Formes canoniques

Définition. Soit un ensemble $\mathcal{V} = \{\mathscr{A}, \mathscr{B}, \ldots\}$ de n variables propositionnelles.

Un *maxterme* sur $\mathcal{V}$ est une disjonction de n littéraux où chacune des n variables propositionnelles apparaît exactement une fois.

Un *minterme* sur $\mathcal{V}$ est une conjonction de n littéraux où chacune des n variables propositionnelles apparaît exactement une fois.

On qualifie de *conjonctive canonique* un conjonction de maxtermes, et de *forme disjonctive canonique* un disjonction de mintermes.

Les dénominations « maxterme » et « minterme » méritent un mot d'explication. Comme on l'a évoqué, il est fréquent d'associer à **F** la valeur 0, et à **V** la valeur 1. Dans ce cadre, l'évaluation d'une disjonction de plusieurs termes revient à prendre le maximum de l'évaluation de chacun des termes : en effet, le résultat vaudra « 0 » si et seulement si tous les termes sont évalués à 0, et 1 dès qu'au moins un terme est évalué à 1. De la même façon, l'évaluation d'une conjonction de plusieurs termes revient à conserver le minimum des évaluations de chacun des termes.

Théorème 17. *Pour toute formule propositionnelle f sur un ensemble $\mathcal{V} = \{\mathscr{A}, \mathscr{B}, \ldots\}$ de n variables propositionelles, il existe une unique (à une permutation près des termes) forme conjonctive canonique équivalente à f, et une unique forme disjonctive canonique équivalente à f.*

Démonstration. La méthode pour construire les formes normales conjonctives et disjonctives dans la preuve précédente construit en fait des formes normales conjonctives et disjonctives canoniques, ce qui permet d'affirmer leur existence.

Pour justifier leur unicité, on peut remarquer que les mintermes et maxtermes apparaissant dans les formes normales canoniques correspondent exactement aux lignes d'une table de vérité. Par conséquent, deux formes conjonctives canoniques distinctes (autrement que par permutation des termes) conduisent à des tables de vérité différentes, et ne peuvent donc pas être sémantiquement équivalentes. □

5.4 Les problèmes « k-SAT »

Les problèmes « k-SAT » s'efforcent de répondre à la question suivante : une proposition logique sous forme normale conjonctive est-elle satisfiable ?

Dans ce cadre, une disjonction de littéraux est généralement appelée « *clause* ». On parle de « k-clause » lorsque la clause fait intervenir k littéraux.

Par exemple, $(\mathscr{A} \vee \mathscr{D})$ et $(\mathscr{B} \vee \mathscr{C} \vee \neg\mathscr{E})$ sont des exemples de clauses (respectivement une 2-clause et une 3-clause).

Les formules propositionnelles qui nous intéressent ici sont donc des conjonctions de clauses. Par exemple, on s'intéresse à la satisfiabilité de la proposition

$$(\mathscr{A} \vee \mathscr{D}) \wedge (\neg\mathscr{A} \vee \neg\mathscr{B} \vee \mathscr{C}) \wedge (\mathscr{B} \vee \neg\mathscr{D}) \wedge (\neg\mathscr{C} \vee \mathscr{E}) \wedge (\mathscr{C} \vee \mathscr{E}) \wedge (\mathscr{B} \vee \neg\mathscr{C} \vee \neg\mathscr{E})$$

Cet proposition logique est bien satisfiable, avec, par exemple, la distribution de vérité $\{\mathscr{A} \mapsto \mathbf{V}, \mathscr{B} \mapsto \mathbf{F}, \mathscr{C} \mapsto \mathbf{F}, \mathscr{D} \mapsto \mathbf{F}, \mathscr{E} \mapsto \mathbf{V}\}$.

Un problème « k-SAT » s'intéresse spécifiquement à la satisfiabilité de conjonction de k-clauses. Le problème « 1-SAT » est trivial (la conjonction est satisfiable si et seulement si un littéral et sa négation n'apparaissent pas tous deux dans la conjonction).

Il a été montré qu'il était possible de déterminer si une conjonction de 2-clauses était satisfiable (problème « 2-SAT ») avec un algorithme dont la complexité est polynomiale. De même, il a été montré que l'on pouvait résoudre la question de la satisfiabilité d'une conjonction de k-clauses avec $k > 3$ aussi efficacement que la question de la satisfiabilité d'une conjonction de 3-clauses (problème « 3-SAT »).

L'essentiel des recherches actuellement porte sur ce problème « 3-SAT ». À l'heure actuelle, on ne connaît pas d'algorithme permettant de déterminer si une telle conjonction est satisfiable en temps polynomial, même si l'on peut vérifier en temps polynomial si une distribution de vérité donnée est un modèle pour la conjonction.

Des problèmes de ce type sont qualifiés par les chercheurs en théorie de la complexité de problème « NP ». Le problème « 3-SAT » a une particularité supplémentaire : il a été montré que n'importe quel problème NP peut être résolu en résolvant un problème de type « 3-SAT ». Les problèmes « 3-SAT » sont en quelque sorte les problèmes les plus difficiles parmi les problèmes « NP ».

Ainsi, trouver un algorithme permettant de déterminer si une conjonction de 3-clauses est satisfiable en temps polynomial aurait pour conséquence que pour *tout* problème dont une solution peut être vérifiée en temps polynomial, il existerait nécessairement un algorithme permettant de décider en temps polynomial si une telle solution existe ou non. C'est le fameux problème dit « $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ », un problème considéré comme l'un des plus importants en informatique, notamment en raison de ses nombreuses conséquences pratiques (par exemple, les nombreux moyens d'authentification et de chiffrement actuels reposent sur des problèmes pour lesquels il est possible de vérifier en un temps raisonnable si une solution est correcte, mais que trouver une solution correcte est en revanche bien trop coûteux en temps de calcul).