SU/FSI/licence/info/LU2IN019
Programmation fonctionnelle

M. Journault – P. Manoury

2020

Prolégomènes

Ce cours présente et étudie divers aspects du style de programmation appelé *fonctionnel*. Il utilise pour cela le langage de programmation OCaml.

Le style fonctionnel s'inspire de la manière dont on définit les calculs en mathématique. Par exemple, le calcul du volume d'une sphère est *exprimé* par la formule $\frac{4}{3}\pi r^3$ où r est le rayon de la sphère. Cette formule permet de *définir une fonction* pour le calcul du volume d'une sphère ; ce qu'en mathématique on écrit $r \mapsto \frac{4}{3}\pi r^3$. C'est une fonction des réels dans les réels, ce que l'on note $\mathbb{R} \to \mathbb{R}$. On dit que son *paramètre* est r et que son *corps* est la formule, $\frac{4}{3}\pi r^3$. En programmation on dit *expression* plutôt que formule. Le principe qui transforme une expression en fonction est appelé *abstraction*. Ce principe a été formalisé par A. Church (inventeur du λ-calcul) qui utilise la lettre grecque λ pour marquer les arguments de fonctions. Il écrirait la fonction de calcul du volume d'une sphère de rayon r de cette manière : $\lambda r\, \frac{4}{3}\pi r^3$. Nous appellerons ce genre d'expression une *expression fonctionnelle*.

Appelons V la fonction $\lambda r\, \frac{4}{3}\pi r^3$. On peut désigner et déterminer le volume d'une sphère de rayon $6,83$ en écrivant *l'application* de la fonction V à *l'argument* $6,83$. Ce que l'on écrit en général de cette manière : $V(6,83)$. La valeur de cette application est la valeur de l'expression $\frac{4}{3}\pi(6,83)^3$ que l'on obtient en remplaçant le paramètre `r` par $6,83$ dans le corps de la fonction V. Ce qui donne la valeur réelle (approchée) $1334,59877027312882$. La valeur de l'application $V(6,83)$ est identique, par définition de V à celle de l'application de l'expression fonctionnelle $\lambda r\, \frac{4}{3}\pi r^3$ à l'argument $6,83$. On peut résumer cela par la suite d'égalités :

$$\begin{aligned} V(6,83) &= (\lambda r\, \tfrac{4}{3}\pi r^3)(6,83) \\ &= \tfrac{4}{3}\pi(6,83)^3 \\ &= 1334,59877027312882 \end{aligned}$$

Chaque expression est associée à une *valeur* (*sauf exception*, nous y reviendrons). Dans notre exemple, l'application $V(6,83)$ est associée à la valeur $1334,59877027312882$. Les mathématiques nous ont habitués aux valeurs numériques, mais également à des valeurs plus complexes, commes les vecteurs, les matrices, etc. De même, en informatique on a des valeurs numériques entières ou flottantes, des valeurs booléennes, des caractères, des chaînes de caractères, des structures, des listes, des tableaux, des dictionnaires, etc.

Une particularité importante de la programmation fonctionnelle est de considérer que **les fonctions sont aussi des valeurs**. Ce sont les valeurs des expressions fonctionnelles. Nous les appellerons des *valeurs fonctionnelles*. À ce titre, une fonction est possiblement argument aussi bien que résultat dans une application.

Synatxe des expression L'écriture des expressions en mathématiques utilise tout un jeu de symboles, de conventions, et d'astuces typographiques. Pour passer de l'écriture mathématique à l'écriture de programmes, il faut fortement limiter les possibilités d'écriture. Pour trois raisons :

1. La première est que les codes sources des programmes sont écrits à l'aide d'un clavier – ce qui limite le jeux de symboles – de manière linéaire – ce qui interdit la notation des exposants comme dans r^3 où l'usage de barres de fraction comme dans $\frac{4}{3}$.
2. La deuxième est que pour pouvoir être analysée informatiquement l'écriture des expressions doit être suffisament régulière et explicite. Ce n'est pas le cas de l'écriture mathématique qui ne fait pas figurer de symbole pour la multiplication, utilise des conventions différentes pour l'application de la division (barre de fraction) ou l'élévation à la puissance (exposant), etc.
3. Finalement en mathématiques une expression ne décrit pas toujours un calcul qu'il est possible de faire faire par une machine. Or dans notre cas les expressions décrivent un programme informatique, il est donc important que celles-ci soient des expressions *calculables* par un ordinateur.

Le λ-calcul est le langage le plus simple pour l'écriture des expressions et des fonctions calculables. Pour définir l'ensemble des expressions [1] du λ-calcul, on choisit un ensemble de symboles X quelconque, potentiellement infini, et on pose :

1. tout symbole de X est une expression ;
2. si x est un symbole de X et t une expression alors *l'abstraction* notée $\lambda x\, t$ est une expression ;
3. si t et u sont des expressions du λ-calcul, alors *l'application* notée $(t\ u)$ est une expression.

Dans ce langage, une transcription de l'expression fonctionnelle $\lambda r\, \frac{4}{3}\pi r^3$ qui utilise les symboles informatiques usuels des opérations de multiplication (`*`), division (`/`) et élévation à la puissance (`**`), serait

$$\lambda r\, ((*\ ((*\ ((/\ 4)\ 3))\ \pi))\ ((**\ r)\ 3))$$

Ce qui n'est guère pratique ni lisible. C'est pourquoi, en général on utilise l'abréviation $(t\ u\ v)$ pour signifier $((t\ u)\ v)$ afin de ne pas multiplier trop les parenthèses. Appliquée à notre exemple, cette abréviation donne

$$\lambda r\, (*\ (*\ (/\ 4\ 3)\ \pi)\ (**\ r\ 3))$$

Cette manière d'écrire l'application d'une fonction à ses arguments s'appelle notation *préfixe*. Elle ne correspond pas à l'usage que nous avons pour les opérations binaires arithmétiques ou booléennes, par exemple. C'est pourquoi, dans les langages de programmation on autorise pour ces symboles la notation usuelle dite *infixe* :

$$\lambda\texttt{r ((4/3) * } \pi\texttt{) * (r ** 3)}$$

Pour les autres applications, on utilise la notation préfixe.

Évalaution des expressions L'association entre une expression et sa valeur est le résultat du *calcul* de l'expression. En informatique, le mécanisme qui effectue cette association est lié au mécanisme d'exécution des programmes. Le calcul réalisé sur les expressions est appelé *évaluation*.
Reprenons notre exemple du calcul du volume d'une sphère en posant que `v` est un *nom* pour l'expression fonctionnelle λ`r ((4/3) * `π`) * (r ** 3)`. Si l'on veut *évaluer*, au sens informatique du terme, l'application de v, il nous reste un point à préciser : quelle est la valeur de la *constante* mathématique π. On ne peut prendre sa valeur *réelle* que l'on ne peut pas écrire. On utilise alors une valeur (approchée) et on pose que π est égale, par définition, à `3.14159265358979312` [2]. L'évaluation détaillée de l'application `(v 6.83)` se déroule ainsi :

1. Dans le monde du λ-calcul, on parle de *terme*, mais nous conserverons ici le mot expression.
2. Dans les langages de programmation, on utilise le *point décimal* plutôt que la virgule. Nous prendrons désormais cette convention.

```
(v 6.83)  =  ((λr ((4/3) * π) * (r ** 3)) 6.83)                          déf. v
          =  ((4/3) * π) * (6.83 ** 3)                                   app.
          =  ((4/3) * 3.14159265358979312) * (6.83 ** 3)                 déf. π
          =  (1.3333333333333326 * 3.14159265358979312) * (6.83 ** 3)    éval. /
          =  4.18879020478639053 * (6.83 ** 3)                           éval. *
          =  4.18879020478639053 * 318.611987                            éval. **
          =  1334.59877027312882                                         éval. *
```

Type et notation préfixe La notation préfixe de l'application dans le λ-calcul ou dans langage OCaml appelle une remarque importante. En mathématiques et dans la plupart des lanages de programmation, on note $V(6.83)$ l'application de V à l'argument 6.83. Mais ce n'est pas ce que l'on fait en OCaml. Dans ce langage, on conserve la notation préfixe du λ-calcul : $(V\ 6.83)$, ou, en style *code source* : `(v 6.83)`.

Considérons un autre exemple d'expression, celle du calcul du volume d'un cylindre : $\pi r^2 \times h$ où r est le rayon de la base du cylindre et h sa hauteur. Si l'on tranforme cette expression en fonction, on obtient une fonction *à deux arguments* (r et h). Appelons C cette fonction. L'habitude mathématique est de dire que C est une fonction de $\mathbb{R}^2$ dans $\mathbb{R}$, c'est-à-dire, une fonction dont l'argument est *un couple* de réels. Ce que l'on écrit $C : \mathbb{R}^2 \to \mathbb{R}$ ou $C : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Son application s'écrit, en mathématique et dans la plupart des langages de programmation, $C(8.63, 10)$ – par exemple.

Ce n'est pas ainsi que l'on voit les choses en λ-calcul et en OCaml. Pour ces derniers, on dit que C est définie par l'expression fonctionnelle `λr λh ((π * (r ** 2)) * h)` et que c'est **une fonction qui à tout réel associe une fonction** des réels dans les réels. Ce que l'on écrit $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$. On dit que C *est de type* $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$. Ce que l'on écrit :

$$C : \mathbb{R} \to (\mathbb{R} \to \mathbb{R})$$

Avec cette vision, **l'application** $(C\ 6.83)$ **est une fonction** des réels dans les réels, c'est-à-dire, une fonction de type $\mathbb{R} \to \mathbb{R}$. À savoir, la fonction définie par l'expression fonctionnelle `λh ((π * (6.83 ** 2)) * h)`. Appelons C' cette fonction. L'application $(C'\ 10)$ est alors de type $\mathbb{R}$. Sa valeur est le réel obtenu par évaluation de l'expression `(π * (6.83 ** 2)) * 10`. Résumons :

- C est de type $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$
- $C' = (C\ 6.83)$ est de type $\mathbb{R} \to \mathbb{R}$
- $(C'\ 10) = ((C\ 6.83)\ 10)$ est de type $\mathbb{R}$

Avec notre simplification de la notation de l'application : $(C\ 6.83\ 10)$ est de type $\mathbb{R}$. Comme on a simplifié la notation de l'application, on simplifie la notation des types et l'on écrit $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$ à la place de $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$. Réécrivons les applications et leur type avec nos notations simplifiées :

- C est de type $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$
- $(C\ 6.83)$ est de type $\mathbb{R} \to \mathbb{R}$
- $(C\ 6.83\ 10)$ est de type $\mathbb{R}$

Plus on ajoute d'arguments à l'expression, plus on retire de $\mathbb{R} \to$ au type de l'expression.

On peut ajouter moins d'arguments qu'il n'y a de flèche dans le type, mais pas plus. Dans le premier cas (comme dans $(C\ 6.83)$), on parle *d'application partielle* et la valeur de l'application est une valeur fonctionnelle. On est en *programmation fonctionnelle*, les fonctions sont aussi des *valeurs*. Dans le second cas, il est légitime d'interdire d'appliquer plus d'argument qu'il n'y a de flèche dans le type car, par exemple, appliquer un réel à un réel, c'est-à-dire, un nombre à un autre nombre, n'a aucun sens : l'expression $(1,5\ 0,3)$ ne désigne aucun réel, n'a pas de valeur.

Type et fonction d'ordre supérieur En mathématique, on définit ainsi le taux de variation d'une fonction [3] : étant donné deux valeurs x_1 et x_2 du domaine d'une fonction f, le taux de variation de cette

3. source : `https://lexique.netmath.ca/taux-de-variation-dune-fonction/`

fonction de x_1 à x_2 est le rapport :

$$\frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

Cette formule est applicable à n'importe quelle fonction f, et n'importe quelles valeurs x_1 et x_2 du domaine de f. Elle permet donc de définir une fonction dont les paramètre sont une fonction (f) et deux valeurs (x_1 et x_2). La fonction de calcul du taux de variation d'une fonction est définie par l'expression fonctionnelle

```
λf λx1 λx2 (((f x2) - (f x1)) / (x2 - x1))
```

Si nous restons dans le domaine du calcul sur $\mathbb{R}$, le type de cette expression est celui d'une fonction dont le premier paramètre est une fonction (des réels dans les réels), le deuxième un réel et le troisième un réel. Le résultat du calcul est un réel. Le type de cette expression s'écrit :

$$(\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} \to \mathbb{R}$$

Les parenthèses autour du type du premier argument sont indispensables. Si nous les avions omises, en écrivant $\mathbb{R} \to \mathbb{R} \to \mathbb{R} \to \mathbb{R} \to \mathbb{R}$, nous aurions donné le type d'une fonction à quatre arguments, ce qui n'est pas ce que nous voulons.

Il faut ici lire avec précaution la règle que nous avons donnée : «plus on ajoute d'arguments [...], plus on retire de flèche». Appelons T la fonction de calcul du taux de variation d'une fonction définie par l'expression ci-dessus. Pour appliquer cette fonction à la fonction V de calcul du volume d'une sphère, on écrit l'expression $(T\ V)$. Son type est $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$. Il est obtenu en retirant la fléche qui se trouve entre $(\mathbb{R} \to \mathbb{R})$ et $\mathbb{R}$ dans le type de T.

Le type de $(T\ V)$ n'est évidemment pas obtenu en retirant la flèche du type du premier argument $(\mathbb{R} \to \mathbb{R})$. On obtiendrait alors que $(T\ V)$ serait de type $\mathbb{R} \to \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ ce correspond au type d'une fonction à trois arguments réels. Or la valeur de l'application $(T\ V)$ est égale à la valeur de l'expression fonctionnelle `λx1 λx2 (((V x2) - (V x1)) / (x2 - x1))` qui est une fonction à deux arguments réels (donc de type $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$). Intuitivement la règle de «retrait des flèches par application» concerne les flèches de «premier niveau». Pour l'exemple du type de la fonction T, ce sont les flèches que nous encadrons ci-dessous

$$(\mathbb{R} \to \mathbb{R}) \boxed{\to} \mathbb{R} \boxed{\to} \mathbb{R} \boxed{\to} \mathbb{R}$$

Pour conclure sur l'écriture complètement parenthésées ou simplifiée des applications et des types en OCaml, on peut observer que

$$A \to (B \to C) = A \to B \to C \text{ alors que } (A \to B) \to C \neq A \to B \to C$$

et de manière duale dans le monde des expressions :

$$((f\ a)\ b) = (f\ a\ b) \text{ alors que } (f\ (a\ b)) \neq (f\ a\ b)$$

1 OCaml : un langage d'expressions typées

Nous tirons de ce qui précède la leçon suivante : une expression est une suite de symboles (les caractères du clavier) qui

1. obéit à une *syntaxe*
2. appartient à un *type*
3. désigne une *valeur*

Ainsi dans le reste de cette section nous nous efforcerons de présenter chacun de ces points pour toutes nouvelles expressions que nous introduirons.

Pour définir la syntaxe des expressions, on distingue entre

1. expressions simples (appelées aussi *expressions atomiques*)
2. expressions composées soit par *abstraction*, soit par *application*.

1.1 Constantes

Les notations définies pour les *constantes* sont des expressions atomiques. Par exemple :

- `true` et `false` sont deux constantes qui appartiennent au type `bool` et désignent les deux valeurs booléennes (le «vrai» et le «faux»). Il n'y a pas d'autres symboles de constantes pour les booléens.
- `0` et `42` sont deux constantes du type `int` qui ont respectivement pour valeur l'entier nul *0* et l'entier positif *42*. Il n'y a pas d'expression atomique pour les valeurs entières négatives. Nous y reviendrons.
- `3.1416` est une constante du type `float` qui désigne une valeur (très) approchée de la constante mathématique π. L'expression atomique `3.0` est aussi une contante du type `float` qui n'a pas du tout la même valeur que la constante `3` du type `int`. Nous y reviendrons.
- `'a'`, `'b'`, `'?'` sont des constantes du type `char` qui ont respectivement pour valeur la caractère *a*, le caractère *b* et le caractère *?*.
- `"hello"`, `"h"` et `""` sont des constantes du type `string` qui ont respectivement pour valeur les chaînes de caractères *hello*, *h* et la chaîne vide. La valeur de la constante `"h"` n'est pas égale à celle de la constante `'h'`.
- `[]` qui désigne la liste vide. Son type s'écrit de manière un peu particulière : `'a list`, nous y reviendrons.

1.2 Variables

L'autre catégorie des expressions atomiques est celle des *variables*. Les variables sont des noms que l'on utilise pour faire référence à une valeur. Nous avons déjà rencontré les «variables» dans les expressions fonctionnelles : ce sont les paramètres de fonctions.

On peut également explicitement *déclarer* des variables en OCaml. Nous y reviendrons.

L'utilisation des «variables» dans les programmes fonctionnels est souvent troublante car en programmation fonctionnelle, **on n'utilise jamais l'instruction d'affectation**. Bien comprendre la différence de conception des programmes entre le style dit *impératif* où l'on utilise l'affectation (et les boucles) et le style fonctionnel est un enjeu important de ce cours.

L'écriture des noms de variables en OCaml est à peu près similaire à celle que l'on rencontre dans d'autres langages de programmation à cette différence que, en OCaml, un nom de variable ne peut pas commencer par une majuscule. La valeur d'une variable dépend bien évidemment du reste du programme et ne peut être déterminé (sauf à de rares exceptions) sans exécuter le programme. Le type d'une variable dépend lui aussi du reste du programme mais, dans le langage OCaml, il peut être déterminé sans exécuter le programme (nous y reviendrons).

1.3 Application

On obtient des *expressions composées* par *l'application* d'un symbole d'opérateur ou de fonction à une ou plusieurs autres expressions. Dans le style fonctionnel où les expression peuvent avoir pour valeur des fonctions, on a, de façon plus générale que l'on obtient des expression composées par application d'une expression à une ou plusieurs autres expressions.

Syntaxe L'application est représentée soit par une notation *infixe*, comme dans `(1 + 1)` ou `(true || false)`, soit par une notation *préfixe*, comme dans `(not true)`, ou `(- 42)`, ou `(-. 3.1416)` ou `(f x)`. Il existe aussi un usage infixe des opérateurs `-` et `-.` comme dans `(1 - 1)` et `(1.0 -. 0.5)`.

> Voir `https://caml.inria.fr/pub/docs/manual-ocaml/libref/Stdlib.html` pour connaître les opérateurs et fonctions standards du langage.

La composition par application des expressions est *incrémentale*. Pour prendre l'exemple des booléens :
- `true` et `false` sont des expressions
- puisque `true` est une expression, `(not true)` est une expression
- puisque `false` et `(not true)` sont expressions, `(false && (not true))` est une expression
- etc.

C'est, en fait, ce que dit la règle de construction des applications du λ-calcul :

3. si t et u sont des expressions du λ-calcul, alors *l'application* notée $(t\ u)$ est une expression.

si on y tolère l'écriture infixe de l'application.

Insistons sur ce que nous avons déjà mentionné : la syntaxe de l'application des fonctions en OCaml diffère fortement de celle que l'on utilise dans des langages comme C ou Python (pour ne citer que ces deux là). Dans ces langages, pour appliquer une fonction $\texttt{f}_1$ à un argument (appelons le $\texttt{a}_1$), on écrit : $\texttt{f}_1\texttt{(a}_1\texttt{)}$, pour appliquer une fonction $\texttt{f}_2$ à deux arguments (disons $\texttt{a}_1$ et $\texttt{a}_2$), on écrit $\texttt{f}_2\texttt{(a}_1\texttt{,a}_2\texttt{)}$, etc. Les parenthèses délimitent la suite des arguments qui sont séparés par une virgule. En OCaml, la syntaxe de l'application est inspirée de celle du λ-calcul comme l'est la syntaxe de l'application des langages de la famille LISP ou Scheme. On écrit : $\texttt{(f}_1\ \texttt{a}_1\texttt{)}$ et $\texttt{(f}_2\ \texttt{a}_1\ \texttt{a}_2\texttt{)}$ où les parenthèses délimitent l'application elle-même, les arguments sont séparés du symbole de fonction et entre eux seulement par des espaces [4]. **Attention,** si un argument dans une application est lui même une expression composée, il faut prendre garde au parenthèsage, même dans le cas où l'argument fait intervenir un opérateur en notation infixe. Par exemple, pour appliquer `f` à `x+1`, il faut écrire `(f (x+1))` car l'écriture `(f x+1)` est lue par OCaml comme `(f x) + 1` ce qui n'est pas ce que l'on veut. Autre exemple, pour appliquer `f` à `-1`, il faut écrire `(f (-1))` car l'écriture `(f -1)` est lue par OCaml comme `(f - 1)`, c-à-d, la soustraction de `1` à `f` ; ce qui n'est pas non plus ce que l'on veut exprimer.

> ***Lorsque l'on débute en OCaml,*** il est fortement conseillé de parenthéser systématiquement les applications (même pour les opérateurs infixes) ; ce n'est que plus tard, avec l'habitude que l'on pourra s'autoriser à relâcher cette contrainte.

Typage Comme toutes les autres expressions, les expressions composées par application ont un type. Contrairement au type des constantes qui est donné *a priori*, le type des applications est «calculé» à partir du type de ses composantes. Ce calcul se base sur des *règles de typage*. La règle de typage des applications la plus générale est celle-ci :

Règle 1. Si l'expression e est de type t_1 `->` `...->` t_n `->` t, si l'expression e_1 est de type t_1, ...et si l'expression e_n est de type t_n alors l'application $\texttt{(}e\ e_1\ \texttt{...}e_n\texttt{)}$ est de type t.

En OCaml, on utilise `->` là où les mathématiques utilisent la flèche $\rightarrow$ pour écrire le type des fonctions. On parle de *type fonctionnel*. Une application qui obéit à cette règle de typage est dite *correctement typée* ou *bien typée*.

Par exemple on calcule le type de l'application `(not true)` de la manière suivante : la fonction prédéfinie `not` est de type `bool -> bool`, la constante `true` est de type `bool` donc l'application `(not true)` est de type `bool`. On utilise un raisonnement similaire pour l'application des opérateurs en notation infixe : l'opérateur prédéfini `+` est de type `int -> int -> int`, les constantes `4` et `2` sont de type `int`, donc l'application `(4 + 2)` est de type `int`.

4. ou, éventuellement, des retours à la ligne.

Les règles de typage ont pour but d'éliminer les applications dont on ne peut pas calculer le type. C'est pourquoi `(not 1)` **n'est pas une expression**, pas plus que `(true + false)`, ni `(true + 0)`, car aucune de ces trois applications ne respecte les règles de typage.

Une expression dont on ne peut pas calculer le type est une expression dont on ne peut pas calculer la valeur. Donc les règles de typage ont pour but d'éliminer les applications dont on ne peut pas calculer la valeur.

Valeurs Afin de calculer la valeur d'une application `(`e e_1 `...`e_n`)`, nous calculons les valeurs v_1, ..., v_n des expressions e_1, ..., e_n, nous calculons ensuite la valeur f de e. Cette dernière doit être une fonction d'au moins n arguments. Si l'application est correctement typée, ce sera toujours le cas. Nous pouvons alors faire le calcul de l'application (au sens mathématiques) $f(v_1, \ldots, v_n)$. Certaines valeurs seront considérées comme données *immédiatement*, comme la valeur des constantes (nous écrirons que la valeur de `42` est l'entier 42) ou la valeur (fonctionnelle) des opérateurs prédéfinis (arithmétiques, booléens, etc.). Dans ce cas, nous dirons simplement que la valeur de `(4 + 2)` est égale à $4 + 2$, c'est-à-dire 6. Ce principe s'applique *récursivement* aux applications plus complexes :

`(4 + (3 - 1))` = $4 + 2$ car 2 est la valeur de (3 - 1)
= 6

Au delà de ces opérateurs de base, lorsque que l'on calcule la valeur de l'application d'une fonction définie par le programmeur (voir ci-dessous **??**), on calcule les valeurs des arguments et on *substitue* ces valeurs dans le corps de la fonction. Cette substitution nous donne une nouvelle expression sur laquelle on itère le processus d'évaluation. Nous avons illustré ce procédé dans notre introduction (voir calcul de `(v 6.83)`) et nous y revenons plus en détails au paragraphe **??**.

Remarque 1. Comme dans de nombreux langages de programmation l'application des opérateurs booléens de conjonction (*et*) et de disjonction (*ou*) – que l'on note respectivement `&&` et `||` en OCaml – s'évaluent de manière dite *paresseuse*. C'est-à-dire que, par exemple, dans l'application `true || (not true)`, l'argument `(not true)` n'est pas évalué car, par définition, `(true ||` e`)` a pour valeur `true` quelle que soit la valeur de e. En revanche, dans `(not true) || true`, l'expression `(not true)` est évaluée. Pour les opérateurs booléens de conjonction (*et*) et de disjonction (*ou*), on commence par évaluer le premier argument et selon la valeur obtenue, on a besoin ou non d'évaluer le second.

Npus donnons un exemple de l'utilisation de ce trait particulier au paragraphe **??**, page 36.

1.4 Abstraction

On appelle *abstraction* le mécanisme qui permet de désigner une variable d'une expression comme paramètre de fonction. Ce mécanisme vient du λ-calcul et existe d'autres langages de programmation que OCaml (en Python ou en Javascript, par exemple). L'abstraction est un constructeur d'expression qui combine un nom de variable et une expression pour construire une expression fonctionnelle.

Syntaxe Dans le langage OCaml, on écrit **`fun`** x `->` e où x est un symbole de variable et e une expression. En général, la variable x apparaît dans l'expression e. Pour prendre un exemple simplissime, l'expression qui désigne la fonction qui à tout entier x associe $2x + 3$ est construite par abstration de la variable `x` dans l'expression `((2 * x) + 3)`. Elle s'écrit **`(fun`** `x -> ((2 * x) + 3))`.

Puisqu'une expression fonctionnelle est une expression, on peut accumuler des abstractions. Par exemple la fonction qui calcul le ou-exclusif de deux booléens s'obtient par abstraction des variables `x` et `y` à partir de l'expression `((x || y) && (not (x && y)))` :

`(fun` `x -> (`**`fun`** `y -> ((x || y) && (not (x && y)))))`

Typage Puisqu'une abstraction désigne une fonction, le type d'une abstraction est un type fonctionnel. L'expression `(fun x -> ((2 * x) + 3))` est une expression représentant une fonction prenant en argument un entier et calculant un autre entier. Elle est donc de type `int -> int`. Plus exactement, si l'on suppose que `x` est de type `int` alors l'expression `((2 * x) + 3)` est de type `int`. On en conclut que l'abstraction sur `x` est de type `int -> int`.

Règle 2. Si e a le type t lorsque x a le type t_1 alors `fun` x `->` e a le type t_1 `->` t.

Lorsque l'on écrit une abstraction, on peut spécifier le type du paramètre de la fonction en écrivant

```
(fun (x:int) -> ((2 * x) + 3))
```

Notez l'usage des deux points (`:`) entre le nom du paramètre `x` et son type `int`. Notez également les parenthèses qui entourent la *déclaration* du paramètre et de son type.

Pour reprendre l'exemple de la fonction à deux paramètres booléens x et y et qui donne la valeur du ou-exclusif entre leur deux valeurs s'écrit :

```
(fun (x: bool) -> (fun (y: bool) -> (x || y) && (not (x && y))))
```

elle est de type `bool -> bool -> bool`.

Valeur Une abstraction a pour valeur la fonction qui associe à toute valeur du paramètre abstrait, la valeur de l'expression d'où ce paramètre est abstrait : intuitivement, la valeur de `(fun n -> (2 * n) + 3)` est la fonction $n \mapsto 2n + 3$. Remarquons ici que les expressions fonctionnelles `(fun n -> ((2 * n) + 3))` et `(fun x -> ((2 * x) + 3))` ont la même valeur car elles désignent la même fonction.

Techniquement, la valeur d'une expression fonctionnelle est appelé une *fermeture*. C'est une structure informatique assez complexe dont l'étude dépasse la portée de ce cours (on l'étudie plutôt en L3 ou en M1). Nous nous contenterons d'utiliser les expressions fonctionnelles dans le *modèle d'évaluation équationnel* que nous présentons ci-dessous.

1.5 Modèle équationnel d'évaluation

La valeur d'une abstraction, c'est-à-dire d'une expression fonctionnelle, ne devient «concrète» que lorsqu'elle est appliquée. Dans ce cas, le paramètre abstrait de la fonction prend une valeur concrète : celle de l'argument appliqué. Considérons l'application de l'abstraction `(fun n -> (2 * n) + 3)` à l'expression `(5 + 2)` que l'on écrit :

```
((fun n -> (2 * n) + 3) (5 + 2))
```

(notez comment sont disposées les parenthèses)

On écrit les *étapes d'évaluation* de cette application comme une suite d'égalités :

```
((fun n -> (2 * n) + 3) (5 + 2))  =  ((fun n -> (2 * n) + 3) 7)
                                  =  (2 * 7) + 3
                                  =  14 + 3
                                  =  17
```

On y voit que la *valeur* de l'application `((fun n -> (2 * n) + 3) (5 + 2))` est celle de l'expression `(2 * n) + 3` où `n` à la valeur de `(5 + 2)`. Et cette valeur est identique à celle de l'expression que l'on écrit `(2 * 7) + 3`.

Nous appelons ce mode de calcul le *modèle équationnel* du processus d'évaluation des fonctions. Il s'inspire du mode de «calcul» que l'on utilise en mathématiques (et, plus précisément, en algèbre) qui revient à aligner des suites d'égalités, c'est-à-dire, des équations. Ce n'est qu'un modèle, ce n'est pas le processus réellement mis en œuvre dans le code compilé des programmes OCaml, mais il permet de *simuler* fidèlement et de comprendre les étapes d'un processus d'évaluation d'un programme en style fonctionnel. Il nous sera utile pour, à la fois, concevoir des fonctions et vérifier «à la main» qu'elles donnent les résultats attendus.

1.6 Expression alternative

Une construction primordiale des programmes est la construction *alternative*. C'est un opérateur de choix gouverné par une valeur booléenne. Elle permet de diriger le processus d'évaluation vers une expression ou une autre. C'est une expression d'un genre particulier que l'on appelle *structure de contrôle*.

Syntaxe L'expression alternative s'écrit à l'aide des *mots réservés* `if`, `then` et `else`. Elle a la forme

`if` e `then` e_1 `else` e_2

où e, e_1 et e_2 sont des expressions. Il faut le voir comme un *opérateur* à trois arguments. Sa syntaxe qui n'est ni préfixe, ni infixe pour se rapprocher de l'utilisation de la construction alternative usuelle dans les langages de programmation.

Le premier argument de l'alternative est appelé *condition*, les deux derniers sont appelés *alternant*. On dit *alternant positif* pour le deuxième et *alternant négatif* pour le troisième.

Par exemple, l'expression

`(if (`n `< 10) then "non admis" else "admis")`

a pour valeur une chaîne de caractères qui correspond au résultat d'un examen pour lequel on a obtenu la note n.

Dans le style de programmmation fonctionnelle, une expression alternative est une *expression*. Ceci permet d'obtenir, par abstraction des fonctions dont le corps est une alternative :

```
(fun n -> (if (n < 10) then "non admis" else "admis"))
```

Ceci permet également d'imbriquer des alternatives comme, par exemple, pour le calcul d'une mention :

```
(if (n < 10) then ""
 else (if (n < 14) then
         (if (n < 12) then "Pass"
          else "AB")
       else
         (if (n < 16) then "B"
          else "TB")))
```

Enfin, mais c'est d'un usage peu courant, ceci permet également d'utiliser une expression alternative comme argument comme dans l'expression :

```
("Resultat: " ^ (if (n < 10) then "non admis" else "admis"))
```

où l'opérateur infixe ^ est l'opérateur de concaténation des chaînes de caractères en OCaml.

Typage La règle de typage de l'alternative est un peu particulière puisqu'elle autorise ses alternants à prendre n'importe quel type pourvu qu'ils soient identiques. La condition doit être de type `bool`. Le type de l'expression alternative est le type commun des alternants. Cette contrainte est résumée par la règle qui suit.

Règle 3. Si e est de type `bool` et si e_1 et e_2 sont de type t alors `(if` e `then` e_1 `else` e_2`)` est de type t.

Cela signifie que l'expression `(if c then 1 else false)` n'est pas correctement typée car `1` est de type `int` et `false` est de type `bool`.

Valeur La valeur d'une expression alternative dépend de la valeur de sa condition :

- si e a la valeur `true` alors `(if` e `then` e_1 `else` e_2`)` a la valeur de e_1 ;
- si e a la valeur `false` alors `(if` e `then` e_1 `else` e_2`)` a la valeur de e_2.

Dans le cadre de notre modèle équationnel d'évaluation, on peut poser que

`(if true then` e_1 `else` e_2`)` $=$ e_1

`(if false then` e_1 `else` e_2`)` $=$ e_2

Il est important de noter que l'opérateur alternatif `if-then-else` possède une propriété d'évaluation particulière : sur ces trois arguments, deux seulement sont évalués, et ce, dans un ordre fixé. La condition est toujours évaluée en premier, puis, selon sa valeur, l'alternant positif ou (exclusif) l'alternant négatif est évalué. Cette propriété fait l'intérêt pratique de l'alternative en programmation. Nous le verrons en particulier lorsque nous aborderons la programmation récursive.

1.7 Programme et définition

Un programme informatique est constitué d'un ensemble de *définitions* et d'un «point d'entrée» par quoi débute l'exécution du programme. Par exemple, dans le langage C, il faut définir une fonction au nom fixé d'avance (`main`) qui est le «point d'entrée» pour l'exécutiton. En programmation fonctionnelle, on retrouve cette structure, à cette différence près que le «point d'entrée» du programme est réduit à une simple expression.

Syntaxe Dans le langage OCaml les définitions ont la forme générale suivante

`let` x `=` e

où x est une variable et e une expression.

Par exemple, on écrira

```
let pi = 3.1416
```

pour définir la constante mathématique π.

Comme on a des expression fonctionnelles, cette forme générale convient également pour définir des fonctions. Par exemple :

```
let v = (fun (r:float) -> (4.0 /. 3.0) *. pi *. (r ** 3.0))
```

C'est la définition de la fonction de calcul du volume d'une sphère. Notez que cette définition dépend de la valeur donnée à la variable `pi`. Nous y reviendrons.

Dans le cas où l'expression utilisée dans une définition est une expression fonctionnelle, on a la possibilité d'écrire de manière un peu plus naturelle :

```
let v (r:float) = (4.0 /. 3.0) *. pi *. (r ** 3.0))
```

en faisant passer les paramètres de la fonction définie à gauche du signe = dans la clause de définition. Le symbole d'abstraction `fun` disparaît. Ces deux formes de définition d'une fonction sont rigoureusement équivalentes.

On peut même ajouter à la définition le *type de retour* de la fonction :

```
let v (r:float) : float = (4.0 /. 3.0) *. pi *. (r ** 3.0))
```

Ici, c'est le `float` qui se trouve entre `:` et =. Nous donnerons sytématiquement cette indication.

De manière générale on peut placer à gauche du signe = tous les paramètres des fonctions. Par exemple, on définit la fonction de calcul du ou-exclusif de cette façon :

```
let xor (x:bool) (y:bool) : bool = (x || y) && (not (x && y))
```

Évaluation L'objet d'une définition est d'associer une valeur à un nom de variable. Dans la déclaration `let` x `=` e la valeur associée à x est celle de l'expression e. Ces déclarations n'ont pas de valeur en elle-même.

La déclaration `let pi = 3.1416` donne à la variable `pi` la valeur 3.1416. Cette valeur **n'est pas modifiable**. Dans le style fonctionnel, **toute déclaration est une déclaration de constante**. C'est évident si l'on se

souvient que le style fonctionnel n'utilise pas d'affectation : une fois définie une liaison entre un nom (de variable) et une valeur ne peut plus être modifiée.

En revanche, il est possible de *redéfinir* une variable. Nous y reviendrons.

Dans le cas de la définition d'une fonction, on peut lire celle-ci comme la donnée d'une équation. Pour notre exemple, on peut lire la définition

```
let v (r:float) : float = (4.0 /. 3.0) *. pi *. (r ** 3.0))
```

comme la donnée de l'équation

$$\texttt{(v } r\texttt{)} \quad = \quad \texttt{(4.0 /. 3.0) *. pi *. (}r\texttt{ ** 3.0))}$$

On lit cette équation comme :

pour toute expression e, l'application `(v e)` *a la valeur de l'expression* `(4.0 /. 3.0) *. pi *. (`e` ** 3.0)`

Et on utilisera cette donnée équationnelle dans notre modèle équationnelle d'évaluation :

$$\begin{array}{rcl} \texttt{(v 6.83)} & = & \texttt{(4.0 /. 3.0) *. pi *. (6.83 ** 3.0))} \\ & = & \text{etc.} \end{array}$$

ou

$$\begin{array}{rcl} \texttt{(v (5.8 +. 1.03)} & = & \texttt{(v 6.83)} \\ & = & \texttt{(4.0 /. 3.0) *. pi *. (6.83 ** 3.0))} \\ & = & \text{etc.} \end{array}$$

Notez que l'on retrouve les égalités que nous avions en utilisant l'application de l'expression fonctionnelle.

Typage Une déclaration donne une valeur à une variable et elle lui donne également un type : le type de l'expression qui la définit. Sur nos deux exemples, on a que `pi` est de type `float` et `v`, de type `float -> float`. De manière générale on a la règle :

Règle 4. Si e est de type t et que l'on a la déclaration `let` x `=` e alors x est de type t.

Le calcul du type de e dépend du type des symboles de constantes, opérations, fonctions et variable qui apparaissent dans e. Par exemple la définition

```
let v (r:float) : float = (4.0 /. 3.0) *. pi *. (r ** 3.0))
```

ne sera correctement typée que si la variable `pi` possède un type (en l'occurrence, le type `float`). Pour que cela soit possible, il faut que la variable `pi` ait été définie *avant* la fonction `v`, comme cela se passe dans une grande part des langages de programmation. Dans ce cas, `v` est de type `float -> float`.

Dans ce cas, l'application de `v` sera également évaluable.

Équations conditionnelles Nous avons dit que l'on peut lire une définition de fonction `let` f x_1 `...`x_n `=` e comme la définition équationnelle `(`f x_1 `...`x_n`)` $=$ e. Dans le cas où e est une alternative, comme dans `let` f x_1`...`x_n `=` `if` e `then` e_1 `else` e_2, on y voit des *équations conditionnelles* que l'on écrit en général sous la forme

$$\begin{array}{rcll} \texttt{(}f\ x_1\texttt{...}x_n\texttt{)} & = & e_1 & \text{si } e \\ & = & e_2 & \text{sinon} \end{array}$$

ou aussi

$$\texttt{(}f\ x_1 \ldots x_n\texttt{)} = \begin{cases} e_1 & \text{si } e \\ e_2 & \text{sinon} \end{cases}$$

Programme On a un *programme* qui calcule la valeur du volume d'une sphère dont la valeur du rayon est saisie au clavier en écrivant la séquence suivante :

```
let pi = 3.1416
let v (r:float) : float =  (4.0 /. 3.0) *. pi *. (r ** 3.0)
let _ = (v (read_float()))
```

où `(read_float())` est l'expression qui permet d'obtenir la valeur saisie au clavier.

Dans ce programme, on a 2 définitions (`pi` et `v`) et un point d'entrée : l'expression `(v (read_float()))`. Notez la dernière «définition» qui commence par `let _ = ...` C'est bien une définition au sens OCmal du terme, mais on utilise le symbole `_` pour signifier que l'on n'associe la valeur de l'expression à aucun nom de variable. On souhaite juste que l'expression (ici `(v (read_float()))`) soit évaluée.

1.8 Définition locale

On peut qualifier les définitions introduites au paragraphe précédent de *globales*. Avec ces définitions, les noms de variables ou de fonctions définies sont utilisables dans toutes les lignes du programme qui suivent la définition. Il existe une autre forme de définition qui permettent de limiter l'utilisation d'une variable à une seule expression. On qualifie ces définitions de *locales*. Les définitions locales appartiennent à une catégorie syntaxique différente de celle des définitions globales. **Une définition locale est une expression**. Elle a un type et une valeur.

Syntaxe Une définition locale combine un nom de variable et deux expressions selon la syntaxe suivante :

`let` x `=` e_1 `in` e_2

L'utilisation de x avec la valeur de e_1 est ici limitée à la seule expression e_2.

Par exemple, pour élever un nombre à la puissance 4, on peut, couper la poire en deux en élevant ce nombre au carré puis en élevant la valeur ainsi obtenue au carré. Appliquer au calcul de 145^4, cela donne :

```
let x = (145 * 145) in (x * x)
```

Nous verrons l'avantage de cette écriture du point de vue de l'évaluation.

Typage Le type de `let` x `=` e_1 `in` e_2 est le type de l'expression e_2. Comme, dans une définition locale, l'expression e_2 fait usage de x, il faut avoir le type de x pour calculer le type de e_2. Le type de x est obtenu en calculant le type de e_1.

Règle 5. Si e_1 est de type t_1 et si e_2 est de type t lorsque x est de type t_1 alors `let` x `=` e_1 `in` e_2 est de type t.

Pour l'exemple de `let x = (145 * 145) in (x * x)`, on a que `(145 * 145)` est de type `int` et, si `x` est de type `int` alors `(x * x)` est de type `int`. Donc `let x = (145 * 145) in (x * x)` est de type `int`.

Valeur De façon analogue, la valeur de `let` x `=` e_1 `in` e_2 est la valeur de e_2 lorsque x a la valeur de e_1. En d'autres termes, la valeur de `let` x `=` e_1 `in` e_2 est la valeur de e_2 dans laquelle on substitue la valeur de e_1 à x. Il y a un effet de *séquence* dans une déclaration locale puisque l'on calcule d'abord la valeur de e_1 puis la valeur de e_2.

On peut représenter cela dans notre modèle équationnel d'évaluation :

$$\begin{array}{rcl} \texttt{let x = (145 * 145) in (x * x)} & = & \texttt{let x = } 21025 \texttt{ in (x * x)} \\ & = & 21025 * 21025 \\ & = & 442050625 \end{array}$$

Notez bien comment on a complètement déterminé la valeur de `x` avant d'effectuer la substitution.

Bien entendu cela donne la même valeur que `(145 * 145) * (145 * 145)`. Mais cette dernière expression demande deux fois le calcul de `(145 * 145)` alors que l'utilisation de la déclaration locale ne le demande qu'une fois.

Plusieurs déclarations locales Puisqu'une déclaration locale est une expression, rien n'empêche dans `let` x `=` e_1 `in` e_2 que e_2 soit elle-même une déclaration locale.

Par exemple, pour tester si un point (x, y) est dans un disque de centre (a, b) et de rayon r on a besoin de calculer $(x-a)^2 + (y-b)^2$ et de vérifier que le résultat est inférieur à r^2. Ce pourrait donc être représenté par l'expression OCaml

```
((x-.a)*.(x-.a) +. (y-.b)*.(y-.b)) <= (r*.r)
```

Pour ne pas répéter le calul de `(x-.a)`, on écrit

```
let dx = (x-.a) in ((dx*.dx) +. (y-.b)*.(y-.b)) <= (r*.r)
```

Puis, pour ne pas répéter le calcul de `(y-.b)`, on écrit

```
let dx = x-.a in let dy = y-.b in ((dx*.dx) +. (dy*.dy)) <= (r*.r)
```

On a donc deux déclarations imbriquées.

Déclaration locale à une fonction À partir de l'expression ci-dessus, on peut définir la fonction de test de l'appartenance d'un point à un cercle. Les paramètres de cette fonction sont les coordonnées du point et le rayon du cercle :

```
let inc (a:float) (b:float) (r:float) (x:float) (y:float) : bool =
   let dx = x-.a in
   let dy = y-.b in
      ((dx*.dx) +. (dy*.dy)) <= (r*.r)
```

On peut naturellement définir une fonction locale à une fonction. par exemple, on définit localement la fonction d'élévation au carré :

```
let inc (a:float) (b:float) (r:float) (x:float) (y:float) : bool =
   let p2 (x:float) : float = x *. x in
   let dx = x-.a in
   let dy = y-.b in
      ((p2 dx) +. (p2 dy)) <= (p2 r)
```

Portée des variables Au risque d'insister : lors d'une définition locale `let` x `=` e_1 `in` e_2, la variable x n'est définie qu'à l'intérieur de l'expression e_2. Considérons par exemple l'expression :

```
if true then
  let x = 3 in x
else
  x
```

Cette expression n'a pas de sens (et est d'ailleurs refusée par OCaml) puisque la variable `x` n'est pas défini pour la branche `else`.

De même pour l'expression :

```
(let x = 3 in x) + x
```

la variable `x` en deuxième argument du + n'est pas définie.

Lorsque plusieurs définitions portant sur un même nom de variable sont imbriquées, c'est la définition la plus proche qui prédomine. En effet lors de l'évaluation une nouvelle définition masque l'ancienne valeur de la variable (de manière locale bien sûr). Par exemple l'expression

```
let x = 3 in
let x = 4 in
  x
```

a pour valeur 4. et l'expression

```
let x = 3 in
  (let x = 4 in x) + x
```

a pour valeur 7 $(4 + 3)$.

Noter qu'il en est de même pour les arguments de fonction :

```
let x = 3 in
let x = 4 in
  (fun x -> x) 5
```

a pour valeur 5.

1.9 Couples, n-uplets

Les couples (ou paires), triplets, quadruplets, etc. sont des *structures de données* prédéfinies en OCaml. De manière générale, on parle de *n-uplet*. Par exemple, on peut *construire* le couple $(4, 2)$ en écrivant simplement `(4, 2)` où la virgule est utilisé comme *l'opérateur de construction* des couples. On dit aussi *constructeur*.

Syntaxe Si e_1, ..., e_n sont des expressions alors `(` e_1`, ...,` e_n `)` est une expression.

Typage Jusqu'à présent nous n'avons croisé dans le monde des *expressions de type* que deux types de type : les types atomiques (types de base : `int`, `bool`, ...) et le type composé servant à dénoter un type de fonction : $t_1 \to t_2$ où t_1 et t_2 sont des expressions de type. Le langage OCaml fournit un type composé t_1`*`t_2 afin de représenter le type d'une paire dont le premier membre est de type t_1 et le second de type t_2.

Ainsi le type de l'expression `(4,2)` est `int * int` (le produit cartésien de `int`). L'opérateur `,` (virgule) permet bien évidemment de construire des valeurs couplant des données hétérogène, comme dans `(4, "quatre")` qui est de type `int * string`, ou encore des *n-uplets*, par exemple le *quadruplet* `("jeudi", 23, "janvier", 2020)` qui est de type `string * int * string * int`.
Règle 6. Si e_1 est de type t_1, ...et si e_n est de type t_n alors `(`e_1`,...,`e_n`)` est de type `(`t_1 `* ...*` t_n`)`.

Attention : `(`t_1 `*` `(`t_2 `*` t_3`))` n'est pas `((`t_1 `*` t_2`)` `*` t_3`)`, ni `(`t_1 `*` t_2 `*` t_3`)`

Valeur La valeur de l'expression `( `e_1`, ..., `e_n `)` est le n-uplet $(v_1, \ldots, v_n)$ où v_1 est la valeur de l'expression e_1, ..., v_n la valeur de l'expression e_n.

Décomposition et types polymorphes Lorsqu'un couple est donné par une variable, par exemple, un paramètre de fonction, on peut *accéder* à ces composants en utilisant la construction `let-in`. Par exemple, on définit la première projection d'un couple [5] de cette manière :

```
let fst c =
  let (c1,c2) = c in c1
```

Nous n'avons pas indiqué ici le type de l'argument `c` ni le type de retour de `fst` car il sont particuliers. En effet, la fonction `fst` est utilisable avec n'importe quel couple : des couples d'entiers (`int * int`), des couples combinant un entier et une chaîne (`int * string`), etc. Cette fonction a toutefois un type que l'on peut écrire `('a * 'b) -> 'a`. On dit que `fst` est *polymorphe*.

Notez l'apostrophe de `'a` et `'b` dans l'écriture de ce type. Elle signifie que `'a` et `'b` sont des *variables de type*. C'est-à-dire que l'application de `fst` sera bien typée pour toute valeur dont le type a la forme `(`t_1 `*` t_2`)` quelque soit les types t_1 et t_2. Par exemple, l'application `(fst (6,"six"))` est bien typée en donnant à la variable de type `'a` le type `int` et à `'b` le type `string`. Pour calculer le type de cette application, on calcule en fait une *instance* du type de `fst`. Pour cela, on calcule le type de l'argument (ici, `(int * string)` ce qui permet d'instancier les variables de type `'a` et `'b`. Dans notre exemple, l'instance calculée est `(int * string) -> string`. Et, par la règle de typage de l'application, l'expression `(fst (6,"six"))` a pour type `int`.

La seule contrainte que le type de `fst` impose est que son argument soit un couple : les applications `(fst 0)` et `(fst (1,2,3))` ne sont pas correctement typées. En revanche `(fst (1, (2,3)))` est correctement typée : `'a` est instanciée en `int` et `'b` en `(int * int)`.

Nous reviendrons sur les fonctions polymorphes, en particulier lorsque nous aborderons les structures de listes (**??**).

En attendant, nous pouvons préciser la définition de `fst` en ajoutant les indications de type :

```
let fst (c: 'a * 'b) : 'a =
  let (c1,c2) = c in c1
```

Rappel : `fst` est de type `('a * 'b) -> 'a`

Définition 1 (Type ou fonction polymorphe)**.** Un *type polymorphe* est un type s'écrit avec des variables de type. Une *fonction polymorphe* est une fonction dont le type est polymorphe.

1.10 Filtrage

Nous avons défini la fonction `xor` par composition de disjonction, conjonction et négation. Nous aurions également pu la définir directement à partir de sa *définition par cas* de valeur de ses arguments. Pour les fonctions booléennes, il est d'usage de représenter ce genre de définition dans un tableau appelé *table des vérité* :

x	y	(xor x y)
true	true	false
true	false	true
false	true	true
false	false	false

5. Cette fonction est définie dans la bibliothèque standard du langage OCaml.

On peut également donner sa *spécification équationnelle* :

```
(xor true true)    =  false
(xor true false)   =  true
(xor false true)   =  true
(xor false false)  =  false
```

Les langages de la famille ML offrent une structure de contrôle très particulière appelée *filtrage* (*pattern matching* en anglais). Elle est construite avec les mots clé `match`, `with` et les symboles réservés `|` et `->`.

Cette construction permet d'écrire une *définition par cas* de la fonction `xor`

```
let xor (b1:bool) (b2:bool) : bool =
   match (b1, b2) with
     (true, true) -> false
   | (true, false) -> true
   | (false, true) -> true
   | (false, false) -> false
```

Notez que l'on a appliqué la construction de filtrage *au couple* de booléens `(b1, b2)`.

Syntaxe La forme générale d'une expression de filtrage est

`match` e `with` m_1 `->` e_1 `|` `...|` m_k `->` e_k

où e, e_1, e_k sont des expressions, au sens usuel, et m_1, ..., m_k sont des expressions particulières appelées *motifs* de filtrage.

Les notions de filtrage et de motif s'appliquent à un très vaste ensemble de valeurs et d'expressions en OCaml. Donner une définition rigoureusement exacte de ce qu'est un motif de filtrage dépasse le cadre que nous avons fixé à ce cours. Nous nous contenterons d'en donner des définitions partielles adaptées à la progression de notre étude du langage. Au point où nous en sommes nous pouvons poser :

Définition 2 (Motif I)**.**

— les constantes sont des motifs ;
— les variables sont des motifs. On les appelle des *variables de motif* ;
— les n-uplets de motifs sont des motifs. On parle alors de *motif composé*.

Il faut ajouter à ces trois clauses de définition une *contrainte de linéarité*

— dans un motif composé, les variables de motif n'ont qu'une seule occurrence.

Par exemple, `(x,x)` n'est pas un motif ; `(x1,x2)` est un motif.

Nous ne le ferons pas figurer explicitement dans nos définitions, mais puisque les motifs sont un sous ensemble des expressions, ils doivent respecter les règles de typage.

Lorsqu'un motif m_i fait mention d'une variable, celle-ci peut-être utilisée dans l'expression e_i associée à ce motif dans la construction de filtrage. Nous y reviendrons.

Typage Les constructions de filtrage obéissent aux contraintes de type suivantes :

1. l'expression e et les motifs m_1, ..., m_k sont de même type.
   Il est à noter que le calcul du type des motifs a pour résultat *l'assignation* d'un type aux variable présentes dans le motif.
2. les expressions e_1, ..., e_k sont de même type (qui n'est pas nécessairement celui de e, m_1, ..., m_k).
   Pour chaque e_i, le type de e_i est calculé en tenant compte du type assigné aux variables de m_i.

Dans ces conditions, l'expression de filtrage elle-même est du type des e_1, ..., e_k.

Évaluation Le filtrage fonctionne un peu comme une alternative multiple (un *switch*). Chaque motif m_i exprime une condition que doit satisfaire l'expression filtrée e. On peut décrire approximativement le mécanisme d'évaluation du filtrage `match` e `with` m_1 `->` e_1 `| ...|` m_k `->` e_k comme celui d'une imbrications d'alternatives :

si e *satisfait le motif* m_1 alors
  e_1
sinon ...
sinon si e *satisfait le motif* m_k alors
  e_k
sinon
  échec

Il faut retenir de cette approximation que, dans un filtrage, les motifs sont examinés séquentiellement, dans l'ordre où ils sont donnés.

Notez que dans notre description, nous n'avons pas écrit «si $e = m_i$», mais «si e *satisfait le motif* m_i». Cela tient au rôle très particulier que jouent les variables de motif dans le filtrage.

Nous limitant pour l'instant aux cas simples de motifs que nous avons définis (constante, variable, n-uplet), nous pouvons définir la relation e *satisfait le motif* m de la mainière suivante :

1. Lorsque m est une constante alors e *satisfait le motif* m si et seulement si $e = m$. Plus exactement, la valeur de e est égale à la valeur de m.
2. Lorsque m est une variable alors pour toute expression e, e *satisfait le motif* m.
   Dans ce cas, le mécanisme de filtrage associe à la variable *la valeur de* e. On appelle cette valeur la *valeur filtrée* par la variable de motif.
3. Lorsque m est un n-uplet de motifs `(`m_1`,...,`m_k`)`, e *satisfait le motif* m, la valeur de e est un n-uplet `(`v_1`,...,`v_k`)` et v_1 satisfait le motif m_1, ..., v_k satisfait le motif m_k.
   Les variables du motifs `(`m_1`,...,`m_k`)` sont associées à leur valeur filtrée.

La grande puissance du filtrage tient dans l'association des variables de motifs à leur valeur filtrée. En effet, nous avons préciser que dans une construction de filtrage, les expressions e_i (à droite du symbole `->`) peuvent utiliser les variables de motifs de m_i (à gauche du symbole `->`). Ainsi, nous pouvons affiner notre description grossière du mécanisme d'évaluation d'une construction de filtrage :

si e *satisfait le motif* m_1 alors
  e_1 où les variables de filtrage prennent leur valeur filtrée
sinon ...
sinon si e *satisfait le motif* m_k alors
  e_k où les variables de filtrage prennent leur valeur filtrée
sinon
  échec

Variable de filtrage Donnons un premier exemple d'utilisation de variables dans un motif de filtrage. En regardant la définition de la fonction `xor`, on peut remarquer que dans les deux derniers cas, le résultat associé est exactement la valeur du deuxième argument. En utilisant une variable, on peut «factoriser» ces deux cas :

```
let xor (b1:bool) (b2:bool) : bool =
   match (b1, b2) with
     (true, true) -> false
   | (true, false) -> true
   | (false, b) -> b
```

Utilisons notre modèle équationnel pour suivre les étapes d'évalaution de l'application `(xor false true)` :

```
(xor false true)  =  match (false,true) with (true, true) -> false | ...
                  =  match (false,true) with (true, false) -> true | ...
                  =  match (false,true) with (false, b) -> b | ...
                  =  true
```

Ce qui s'est passé ici, c'est que l'expression `(false,true)` satisfait le motif `(false,b)` lorsque `b` prend la valeur `true`. L'expression associée à ce motif est simplement `b` dont la valeur (donnée par le mécanisme d'évaluation du filtrage) devient la valeur de l'expression entière du filtrage.

Illustrons à nouveau ce phénomène de liaison de valeur à une variable sur l'exemple *ad hoc* suivant

```
match (3+2) with
| 0 -> 0
| x -> x - 1
```

L'expression `3+2` s'évalue en 5, la valeur 5 ne satisfait pas le motif `0` (cas de motif constant), toutefois **il est possible** de «filtrer» 5 avec la variable `x`. En effet lorsqu'une variable apparaît dans un motif le programme essaie de trouver une valeur pour cette variable qui rend le filtrage valide. Ainsi ici `x` va recevoir la valeur 5 et l'expression complète s'évalue en la valeur 4.

Voici les étapes d'évaluation de cette expression dans notre modèle équationnel :

```
match (3+2) with 0 -> 0 | x -> x - 1  =  match 5 with 0 -> 0 | x -> x - 1
                                       =  match 5 with x -> x - 1
                                       =  5 - 1
                                       =  4
```

Ainsi, le filtrage **ne teste pas l'égalité** entre `5` et la variable `x`, mais donne à `x` la valeur `5` et la valeur `5` *satisfait le motif* `x`.

2 Programmation récursive

Un langage de programmation ne permet pas d'aller très loin s'il n'offre pas un moyen de pouvoir *itérer*, c-à-d, *répéter* une séquence de calcul. Dans le style impératif, le moyen de procéder aux itérations est fourni par les *instructions de boucles*. N'en disposant pas en programmation fonctionnelle, on a recours à un autre moyen (utilisable également en impératif) : les *définitions récursives* de fonctions.

Exemple La fonction d'élévation à la puissance n'est pas définie pour les entiers dans la bibliothèque standard. Il n'y a pas d'expression qui donne la valeur de «x à la puissance n». Mais :

«x à la puissance 0» = 1
«x à la puissance 1» = x
«x à la puissance 2» = $x \times x$
«x à la puissance 3» = $x \times x \times x$
etc.

Schématiquement :

$$\text{«}x\text{ à la puissance }n\text{»} \quad \text{est égal à} \quad \underbrace{x \times \cdots \times x}_{n\ fois}$$

Il est alors crucial de faire la remarque suivante. Lorsque n est plus grand que 0 :

$$\underbrace{x \times \cdots \times x}_{n\ fois} \quad \text{est égal à} \quad x \times \underbrace{x \times \cdots \times x}_{n-1\ fois}$$

Cette remarque nous permet de poser *l'équation récursive suivante* :

$$\text{«}x \text{ à la puissance } n\text{»} = x \times \text{«}x \text{ à la puissance } (n-1)\text{»}$$

Et on a par convention que «x à la puissance 0» = 1. De cela on déduit la *définition récursive* :

$$\begin{array}{rcll} \text{«}x \text{ à la puissance } n\text{»} & = & 1 & \text{, si } (n = 0) \\ & = & x \times \text{«}x \text{ à la puissance } (n-1)\text{»} & \text{, sinon} \end{array}$$

Écrivons $pow(x, n)$ pour «x à la puissance n». Les *équations récursives* ci-dessus deviennent :

$$\begin{array}{rcll} pow(x,n) & = & 1 & \text{si } n = 0 \\ & = & x \times pox(x, n-1) & \text{sinon} \end{array}$$

On peut transcrire cette définition en Ocaml. Pour poser une *définition récursive*, on utilise les mots clé `let` et `rec`, l'un après l'autre :

```
let rec pow (x:int) (n:int) : int =
   if (n = 0) then 1
   else x * (pow x n)
```

Cette définition donne les schémas d'évaluations attendus :

```
(pow x 0)   =  1
(pow x 1)   =  x * (pow x 0)
            =  x * 1
            =  x
(pow x 2)   =  x * (pow x 1)
            =  x * (x * (pow x 0))
            =  x * (x * 1)
            =  x * x
(pow x 3)   =  x * (pow x 2)
            =  x * (x * (pow x 1))
            =  x * (x * (x * (pow x 0)))
            =  x * (x * (x * 1))
            =  x * (x * x)
```

etc.

Fonction partielle et exception La fonction `pow` est de type `int -> int -> int`. Comme `(-1)` est de type `int`, l'application `(pow` e `(-1))` est correctement typée, pour toute expression entière e. Mais,

`(pow` e `(-1))` = e `*` `(pow` e `(-2))` = e `*` e `*` `(pow` e `(-3))` = ...

La fonction `pow` est partielle en son second argument : il y a des arguments d'appel e_2 de type `int` pour lesquelles, quelle que soit l'expression entière e_1, l'application `(pow` e_1 e_2`)` n'a pas de valeur. En langage courant, on dit que «la fonction boucle», sous entendu, «infiniment».

Pour éviter ce cas de figure, on peut adopter une attitude de *programmation défensive* qui consiste à vérifier que le second argument ne risquera pas de provoquer une évaluation indéterminée. Pour la fonction `pow`, on vérifie que la valeur du second argument n'est pas strictement négative.

On a deux manières de mettre en œuvre la programmation défensive :

a) on complète le domaine de la fonction, en convenant, par exemple, que pour les puissances négatives, la valeur de l'application est 1. Ce qui donne la définition :

```
let rec pow (x:int) (n:int) : int =
  if (n <= 0) then 1
  else x * (pow x (n-1))
```

On aura ici que `(pow x (-1))` a pour valeur 1. Cette première manière a toutefois l'inconvénient que l'application d'un argument impropre peu passer inapercue.

b) on intercepte les applications hors domaine en *déclenchant* une exception. L'utilisation d'un argument impropre ne passera pas alors inaperçue.
On peut pour cela utiliser la *fonction prédéfinie* `failwith` :

```
let rec pow (x:int) (n:int) : int =
  if (n < 0) then failwith "invalid exponent"
  else if (n = 0) then 1
  else x * (pow x (n-1))
```

La fonction prédéfinie `failwith` déclenche l'exception `Failure` accompagnée d'un message sous forme de chaîne de caractère (voir **??** pour les chaînes de caractères).
On peut également utiliser la *fonction primitive* `raise` avec une exception prédéfinie du langage. L'exception `Invalid_argument` est d'ailleurs prévue à cet effet :

```
let rec pow (x:int) (n:int) : int =
  if (n < 0) then raise (Invalid_argument "pow: negative exponent")
  else if (n = 0) then 1
  else x * (pow x (n-1))
```

Programmation défensive et définition locale de fonction Notre mise en œuvre de la programmation défensive pour la fonction `pow` à l'inconvénient d'avoir à évaluer à chaque appel récursif de la fonction un test qui n'est utile qu'une seule fois : lors du «premier» appel de la fonction. Pour palier cet inconvénient, on divise le calcul de la fonction en deux temps :

— Tester si le second argument est négatif ou non.
— Calculer récursivement l'élévation à la puissance, si le second argument n'est pas négatif. Dans ce cas, on pourra utiliser une définition «incomplète».

Pour réaliser cela, nous allons utiliser une définition locale.

```
let pow (x:int) (n:int) : int =
  let rec loop (n:int) =
    if (n = 0) then 1
    else x * (loop (n-1))
  in
    if (n < 0) then raise (Invalid_argument "pow: negative exponent")
    else (loop n)
```

Notez que la définition de `pow` elle-même n'est pas une définition récursive : on n'a pas écrit `let rec pow ...` La fonction récursive est la fonction locale `loop`. Celle-ci est utilisée dans l'expression qui vient après le mot clé `in`. L'ensemble syntaxique `let rec loop ...in ...` forme l'expression qui définit la fonction `pow`.

Portée des variables Il y a plusieurs remarques à formuler concernant la *portée* des variables dans notre dernière définition de `pow`.

— la déclaration locale de la fonction `loop` a pour portée l'expression qui vient après le mot clé `in`. C'est-à-dire que dans cette expression, le nom `loop` est connu et peut être utilisé. Cette définition de `loop` ne peut être utilisée qu'à cet endroit : c'est sa *portée lexicale*. Aucune autre portion de programme ne peut utiliser cette définition.

— la fonction locale `loop` utilise la variable `x` alors que celle-ci ne fait pas partie des arguments de `loop`. Ceci est possible car la définition de `loop` est *dans la portée* des arguments de la définition de `pow`.
— enfin, l'argument de `loop` s'appelle `n`, comme le second argument de `pow`. Il ne peut toutefois pas y avoir ici de confusion pour le processus d'évaluation : la mention de `n` dans la définition de `loop` *masque* celle de `n` dans la définition de `pow`.

Récurrence terminale Le schéma d'évaluation de `(pow 5 3)` se développe ainsi :

```
(pow 5 3)  =  (loop 3)
           =  5 * (loop 2)
           =  5 * (5 * (loop 1))
           =  5 * (5 * (5 * (loop 0)))
           =  5 * (5 * (5 * 1))
           =  5 * (5 * 5)
           =  5 * 25
           =  125
```

On y observe deux phases :
— la première s'achève lorsque les appels récursifs de `loop` ont tous été «résolus». dans cette phase, les multiplications par 5 sont «mises en attente».
— la seconde consiste à effectuer les multiplications jusqu'à obtenir le résultat final.

On appelle ces deux phases, respectivement la *descente récursive*[6] et la *remontée récursive*. Les étapes de la descente récursive ont un coût mémoire qui peut devenir rédhibitoire pour l'évaluation des expressions. Pour comprendre d'où vient ce coût il faut entrer un peu dans les détail du processus d'évaluation. Par exemple, on décompose l'évaluation l'expression e_1 `*` e_2 en trois étapes :

1. évaluer e_1 (qui donne une valeur n_1)
2. évaluer e_2 (qui donne une valeur n_2)
3. multiplier n_1 par n_2

En attente du déclenchement de la multiplication, les valeurs n_1 et n_2 sont stockées dans une zone mémoire appelée *pile* d'évaluation. Ainsi, l'évaluation de l'expression `5 * (loop 2)` stocke la valeur 5 sur la pile avant d'évaluer l'appel récursif `(loop 2)` qui, à son tour *empile* 5 avant d'évaluer `(loop 1)` qui, à son tout empile 5 avant d'évaluer l'application `(loop 0)` qui, elle, empile la valeur 1. Dans notre écriture équationnelle de l'évaluation, on peut voir les empilements successifs de la valeur 5 dans l'expression `5 * (5 * (5 * ...))`.

Ces trois multiplications par 5 viennent de ce que nous avons évalué `(loop 3)`. En général, pour tout entier n (positif), pour évaluer `(loop` n`)`, il faut empiler n fois la valeur 5. Si n devient très grand, la taille de la pile d'évaluation (qui est une zone mémoire finie) peut ne pas suffire. Et dans ce cas, la tentative d'évaluation sur machine échoue. Avec Ocaml, on obtient un message d'erreur signalant un *débordement de pile* : `Stack overflow during evaluation`.

Il est possible d'éviter dans certains cas la mise en attente croissante de calculs, et donc la consommation de pile, si l'on adopte une forme particulière de définition récursive : les définitions *récursives terminales*.

Pour transformer la définition de `loop` en forme récursive terminale, on ajoute à celle-ci un argument appelé *accumulateur*. Cet argument supplémentaire nous permettra d'effectuer les multiplications que notre première définition de `loop` laisse en attente. Voici comment les choses se présentent :

```
let pow (x:int) (n:int) : int =
  let rec loop (n:int) (r:int) =
    if (n = 0) then r
    else (loop (n-1) (x*r))
```

6. Nous empruntons cette expression au domaine de l'analyse syntaxique.

```
  in
    if (n < 0) then raise (Invalid_argument "pow: negative argument")
    else (loop n 1)
```

Si l'on développe le schéma d'évaluation de `(pow 5 3)`, on obtient à présent :

$$\begin{array}{lcl} \texttt{(pow 5 3)} & = & \texttt{(loop 3 1)} \\ & = & \texttt{(loop 2 5)} \\ & = & \texttt{(loop 1 25)} \\ & = & \texttt{(loop 0 125)} \\ & = & 125 \end{array}$$

L'accumulateur contient à chaque appel récursif une valeur correspondant au résultat des multiplications mises en attente dans la version non terminale. Lorsqu'il n'y a plus d'appel récursif, la valeur de l'accumulateur correspond bien à toutes les multiplications qu'il fallait effectuer. En général, on a que `(loop` n r`)` a pour valeur $5^n \times r$. Puisqu'à l'appel initial de `loop`, on donne à r la valeur 1, on obtient bien que `(loop` n `1)` a pour valeur $5^n \times 1$, soit 5^n.

Attention : nous ne pouvons pas tirer de l'exemple que nous donnons de transformation d'une définition récursive en récursive terminal un principe général. Il n'y a pas de méthode universelle, pas d'algorithme, pour obtenir une version récursive terminale qui maîtrise la consommation mémoire. Dans certains cas, la transformation peut même devenir inutilement complexe.

Fonctionnelle On peut définir une fonction qui réalise une «boucle récursive» générique contrôlée par une valeur entière. Étant donné une fonction f et une valeur a et un entier n, elle donnera la valeur de $\underbrace{(f \ldots (f}_{n\ fois}\ a) \ldots)$ – ce que l'on note aussi $f^n(a)$. On peut définir cette itération d'applications, récursivement, en fonction de n :

$$\begin{array}{lcll} f^n(a) & = & a & \text{si } n = 0 \\ & = & f(f^{n-1}(a)) & \text{sinon} \end{array}$$

Ce qui donne la définition Ocaml :

```
let rec iter (n:int) (f: 'a -> 'a) (a:'a) : 'a =
  if (n > 0) then (f (iter (n-1) f a))
  else a
```

On peut aussi poser les équations récursives suivantes :

$$\begin{array}{lcll} f^n(a) & = & a & \text{si } n = 0 \\ & = & f^{n-1}(f(a)) & \text{sinon} \end{array}$$

Ce qui donne la définition Ocaml :

```
let rec iter (n:int) (f: 'a -> 'a) (a:'a) : 'a =
  if (n > 0) then (iter (n-1) f (f a))
  else a
```

qui est récursive terminale.

La fonction `iter` a deux particularités :

- la fonction `iter` est de type : `int -> ('a -> 'a) -> 'a -> 'a`. C'est une fonction polymorphe ;
- c'est une *fonction d'ordre supérieure*, appelée aussi *fonctionnelle*, car l'un de ses arguments est lui même une fonction (`f: 'a -> 'a`).

Cette fonction permet de donner une définition compacte de l'élévation à la puissance. En effet, pour obtenir la valeur de «x à la puissance n», il faut itérer n fois la fonction «multiplier par x» sur la valeur neutre 1.

En utilisant une expression fonctionnelle comme argument de `iter`, on peut poser :

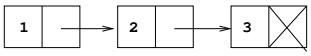
```
let pow (x:int) (n:int) : int =
  if (n < 0) then raise (Invalid_argument "pow")
  else (iter n (fun r -> x * r) 1)
```

où l'expression fonctionnelle `(fun r -> x * r)` représente l'opération «multiplier par x» ; elle est de type `int -> int`. Notez que cette définition de `pow` n'est pas récursive, comme ne l'était pas la définition qui utilise la fonction récursive locale `loop`. Ici, c'est la fonction `iter` qui joue le rôle que jouait `loop`.

3 Structures linéaires : les listes

Les listes sont des *structures linéaires dynamiques*. Dans les langages de programmation, les chaînes de caractères ou les tableaux sont des structures *statiques* : une fois créées, leur taille ne change plus. Pour ajouter un élément dans un tableau, il faut en créer un nouveaux pour y recopier les éléments de l'ancien, plus celui que l'on veut ajouter. L'ajout d'un élément dans une liste sera beaucoup plus économique ; si l'on ajoute en début de liste. C'est ce que l'on fait dans le langage C en créant des *structures allouées*[7] *et chaînées dynamiquement*.

On représente schématiquement de telles structures par des enchaînements (à l'aide de flèches) de cellule mémoires contenant les valeurs de la liste. Par exemple, la liste chaînée contenant les entiers 1, 2 et 3 peut se dessiner ainsi :



Dans le style fonctionnel de programmation, il est possible de construire et d'explorer de telles structures uniquement en écrivant des expressions. Les opérations concrètes d'allocation et de chaînage sont prises en charge par le mécanisme d'évaluation.

3.1 Construction et typage des listes

Type des listes Les structures prédéfinies de listes en OCaml peuvent contenir des valeur appartenant à n'importe quel type. Si une liste contient des valeurs de types t, le type de la liste elle-même s'écrit `(`t `list)`. La liste dessinée ci-dessus, qui contient des valeurs entières (type `int`) est de type `(int list)`. On dit que le type des listes est *paramétré* par le type des valeurs qu'elles contiennent. Pour prendre un autre exemple : le type d'une expression dont la valeur est une liste de couples d'entiers et de chaînes de caractères s'écrit `(int * string) list`. Il arrive que l'on ne sache pas déterminer le type des valeurs contenues dans une liste : par exemple, la liste vide. Dans ce cas, on utilise la notation des type polymorphes : `('a list)` la variable de type `'a` désigne le type (indéterminé) des valeurs contenues dans la liste. On peut également avoir des types d'éléments partiellement déterminés. Par exemple le type d'une expression dont la valeur est une liste de couples sans que l'on connaisse le type des valeurs des constituants des couples s'écrit `('a * 'b) list`.

Par définition, toutes les valeurs contenues dans une liste doivent avoir le même type : par exemple, il n'y a pas de liste qui contienne à la fois des entiers (type `int`) et des chaînes de caractères (type `string`).

Syntaxe : constructeurs Les *constructeurs* des listes sont les opérations primitives[8] qui sont à la base de la construction de toutes les structures de listes. Ces sont eux qui réalisent les opérations d'allocation et de chaînage évoquées plus haut. Ils suffisent à obtenir toutes les structures de listes souhaitées. C'est à dire que

7. avec `malloc`
8. Dans les langages de programmations, les opérations dites *primitives* sont celles qui ne peuvent être définies dans le langage lui-même.

pour toute expression e de type `(`t `list)`, il est possible d'écrire une expression e' en utilisant uniquement des valeurs de t [9] et les constructeurs des listes telle que e'=e.

Le type des listes possède deux constructeurs. Le premier est une constante : la liste vide que l'on note `[]`. Le second est l'opérateur qui permet d'ajouter un élément en tête de liste. Il est symbolisé par `::`, en notation infixe, que l'on prononce *«conse»* [10].

Pour construire la liste contenant (dans cet ordre) les entiers 1, 2 et 3, que nous avons prise en exemple ci-dessus, on commence par prendre la liste vide, puis on y ajoute 3. Ce qui donne la liste contenant uniquement l'entier 3. En OCaml, ces opérations sont réalisées par l'évaluation de l'expression `3::[]`. À cette liste, on ajoute l'entier 2. C'est ce que donne l'évaluation de l'expression `2::(3::[])`. Enfin, la liste entière est obtenue par évaluation de `1::(2::(3::[]))`.

Avec le même constructeur `::` et la même liste vide `[]`, on peut aussi construire la liste qui contient le booléen `true` en écrivant `(true::[])`. On obtient dans ce cas une valeur de type `bool list`, une liste de booléens.

Par contrainte de typage, une liste ne peut contenir des valeurs appartenant à des types différents : par exemple, `(true::(42::[]))` n'est pas une expression correctement typée ; ça n'est donc pas une expression du tout en OCaml. Par respect de cette contrainte, le constructeur `[]` est de type `'a list` et le constructeur `::` est de type `'a -> ('a list) -> ('a list)`. Notez que ce sont des *types polymorphes*.

Enfin, les deux constructeurs `[]` et `::` peuvent être utilisés pour écrire des *motifs de filtrage* pour les listes. Cette possibilité est justifiée par le fait que toute valeur de type `(`t `list)` est égale à la valeur d'une expression qui n'utilise (pour construire la liste) que les constructeurs `[]` et `::`. C'est-à-dire que toute liste a une valeur égale à une liste que l'on peut écrire *sous la forme* `[]`, lorsque la valeur est la liste vide, ou *sous la forme* `x::xs` sinon. En effet, dans ce dernier cas, la valeur est nécessairement obtenu par ajout d'un élément `x` en début d'une certaine liste `xs`. On dira que toute liste *est de la forme* `[]` ou `x::xs`.

Abréviations Il y a potentiellement une infinité de listes. Dune part parce que, en théorie, une liste peut contenir autant d'éléments que l'on désire. D'autre part car on peut construire des listes contenant des valeurs aussi complexes que l'on désire.

Par exemple, on a,
les listes de booléens `(true::[])`, `(true::(false:[]))`, `(true::(false::(false::[])))`,... ;
les listes d'entiers `(1::[])`, `(1::(2::[]))`, ... ;
les listes de chaînes de caractères : `("hello"::[])`, `("hello"::("world"::[]))`, ... ;
les listes de couples d'entiers : `[]`, `((0,0)::[])`, `((0,0)::((0,1)::[]))`, `((0,0)::((0,1)::((1,0)::[])))`,... ;
les listes de listes d'entiers : `([]::[])`, `((1::[])::[])`, `((1:[])::((1::(2::[]))::[]))`, ... ;
etc.

Les types respectifs de ces listes sont `(bool list)`, `(int list)`, `(string list)`, `((int*int) list-` et `((int list) list)`.

Le langage OCaml autorise à ne pas mentionner toutes les parenthèses de l'application du constructeur `::`. Par exemple, on peut écrire `1::2::[]` à la place de `(1::(2::[]))`. Attention toutefois aux listes dont les éléments sont complexes : il n'est pas possible de supprimer toutes les parenthèses de `((1::[])::[])` en écrivant `1::[]::[]` ; cette expression est mal interprétée ; il faut écrire `(1::[])::[]`.

Enfin, on peut encore abréger l'écriture des expressions pour les listes en utilisant une notation assez standard pour les structures linéaires dans les langages de programmation en notant les éléments de la listes entre crochets (`[` et `]`) et en les séparant par des point-virgules (`;`). Le tableau ci-dessous donne la correspondance

9. Plus exactement, des expressions désignant des valeurs de t.
10. En référence au langage LISP où cet opérateur a été initialement introduit.

entre les expressions complètement parenthésées de nos exemple, leur équivalent avec moins de parenthèse et leur équivalant avec la notation «entre crochets».

`(true::[])`	`true::[]`	`[true]`
`(true::(false::[]))`	`true::false::[]`	`[true; false]`
`(true::(false::(false::[])))`	`true::false::false::[]`	`[true; false; false]`
`((0,0)::((0,1)::[]))`	`(0,0::(0,1)::[]`	`[(0,0); (0,1)]`
`([]::[])`	`[]::[]`	`[[]]`
`((1::[])::[])`	`(1::[])::[]`	`[[1]]`
`((1:[])::((1::(2::[]))::[]))`	`(1::[])::(1::2::[])::[]`	`[[1]; [1;2]]`

On peut mélanger les abréviations. Par exemple on écrira `1::[2]` comme abréviation de `(1::(2::[]))`.

Par abus de langage, nous pourrons utiliser la notation des listes «entre crochets» comme «valeur» d'une expression de type liste. Par exemple, nous nous autoriserons à dire que la valeur de l'expression `(1+1)::(2+1)::(3+1)::[]` est la liste `[2;3;4]`.

3.2 Listes et filtrage

La structure de contrôle `match-with` qui nous avait permis de distinguer entre plusieurs cas de valeurs des booléens ou à destructurer des n-uplets peut également servir à distinguer les *cas de construction* des listes.

Par exemple, on peut l'utiliser pour définir une fonction booléenne valant `true` si et seulement si son argument est une liste vide. C'est-à-dire, la fonction que nous appelerons `is_empty` et qui répond à la spécification équationnelle suivante

$$\begin{array}{rcll} \texttt{(is\_empty xs)} & = & \texttt{true} & \text{si } \texttt{xs=[]} \\ & = & \texttt{false} & \text{sinon} \end{array}$$

Sachant que la valeur de toute liste est de la forme `[]` ou `x::xs`, cette donnée équationnelle est équivalente à :

$$\begin{array}{rcl} \texttt{(is\_empty [])} & = & \texttt{true} \\ \texttt{(is\_empty (x::xs))} & = & \texttt{false} \end{array}$$

Puisque `[]` et `x::xs` sont des motifs, cela permet l'utilisation du filtrage pour définir `is_empty` :

```
let is_empty (xs : 'a list) : bool =
  match xs with
    [] -> true
  | x::xs' -> false
```

Puisque ni la variable de motif `x`, ni la variable de motif `xs` ne sont utilisées, on peut ici ne pas exprimer le deuxième cas, en utilisant le *motif universel* noté `_` :

```
let is_empty (xs : 'a list) : bool =
  match xs with
    [] -> true
  | _ -> false
```

On peut voir ici l'utilisation du motif universel `_` comme un simple «sinon», «dans tous les autres cas». Pour le processus d'évaluation, le motif universsel `_` est traité (presque) comme une variable : toute expression satisfait le motif universelle ; mais aucune liaison entre `_` et la valeur de l'expression n'est créée. Cela permet d'éviter la consommation mémoire due à la liaison lorsque cela n'est pas nécessaire.

Attention : le motif universel doit être utilisé à la bonne place, c'est-à-dire, en seconde position. La définition suivante n'est pas correcte :

```
let is_empty (xs : 'a list) : bool =
  match xs with
    _ -> false
  | [] -> true
```

car alors toute application de `is_empty` donnerait la valeur `false`.

La définition suivante complète la définition 2 :
Définition 3 (Motif II)**.**

- `_` est un motif.
- les constantes sont des motifs ;
- les variables sont des motifs ;
- si m_1, ..., m_k sont des motifs alors `(`m_1`,...,`m_k`)` est un motif composé ;
- `[]` est un motif
  et si m_1 et m_2 sont des motifs alors m_1`::`m_2 est un motif composé.
- dans un motif composé, les variables de motif n'ont qu'une seule occurrence.

Considérons un autre exemple de fonction sur des listes d'entiers : la fonction qui «remplace» les deux premiers éléments d'une liste par leur somme. Appelons `pop` cette fonction. On aura, par exemple que `(pop (2::4::6::[]))` est égal à `(2+4)::6::[]`, c-à-d, `6::6::[]`. Si la liste contient moins de deux éléments, le résultat de l'application de la fonction `pop` est la liste «inchangée». Nous donnons de cette fonction une définition en terme de filtrage. Pour cela, il faut déterminer quels motifs de filtrage utiliser.

Il y a deux formes de listes qui contiennent moins de deux éléments et que l'on peut exprimer par les motifs `[]` (liste vide) et `x::[]` (liste qui contient exactement un seul élément). La forme des listes qui contiennent au moins deux éléments s'exprime par le motif `x1::x2::xs`. On peut alors poser que

```
(pop [])           =  []
(pop (x::[]))      =  x::[]
(pop (x1::x2::xs)) =  (x1+x2)::xs
```

Cette spécification équationnelle nous donne immédiatement une définition de `pop` en OCaml :

```
let pop (xs: int list) : (int list) =
  match xs with
    [] -> []
  | x::[] -> x::[]
  | x1::x2::xs -> (x1+x2)::xs
```

> Il faut noter ici le caractère fonctionnel de cette définition : on ne *modifie* pas la liste passé en argument, mais on *construit une nouvelle liste* à partir des composants de la liste donnée en argument.

3.3 Définition récursive

Les listes sont des structures linéaires comme le sont les tableaux. Dans la plupart des cas, le traitement d'une telle structure réclame un processus itératif. Nous avons vu au chapitre 2 comment les définitions récursives donnent le moyen de réaliser des processus itératif en programmation fonctionnelle. C'est un moyen similaire que nous utiliserons pour le traitement itératif des listes en combinaison avec le mécanisme de filtrage.

Récurence structurelle Illustrons la mise en œuvre de ce moyen sur l'exemple du calcul de la somme des entiers contenus dans une liste.

Schématiquement, la somme des valeurs d'une liste de la forme n_1`::`n_2`::...::`n_k`::[]` est égale à la valeur de l'expression n_1`+`n_2`+...+`n_k. Comme nous l'avions fait avec le calcul de l'élévation à la puissance, on décompose cette somme en n_1 `+ (`n_2`+...+`n_k`)` où `(`n_2`+...+`n_k`)` est la somme des valeurs contenues dans la liste n_2`::...::`n_k`::[]` qui est la suite de la liste n_1`::`n_2`::...::`n_k`::[]`.

De manière générale, pour toute liste dont la première valeur est `n` et la suite `ns`, c'est-à-dire, pour toute liste de la forme `(n::ns)`, cette remarque nous permet de poser que

```
(sum (n::ns))  =  n + (sum ns)
```

où `sum` est le nom que nous donnons à la fonction de calcul de la somme des valeurs contenues dans une liste.

Pour obtenir une définition générale, il faut considérer le cas des listes «sans suite», c'est-à-dire, le cas de la liste vide. On peut poser par convention que dans ce cas le résultat est l'entier 0.

```
(sum [])  =  0
```

Ce qui est cohérent avec ce que nous attendons puisque

$$\begin{array}{rcl} \texttt{(sum (}n_1\texttt{::}n_2\texttt{::...::}n_k\texttt{::[]))} & = & n_1\texttt{+}n_2\texttt{+...+}n_k \\ & = & n_1\texttt{+}n_2\texttt{+...+}n_k\texttt{+0} \\ & = & n_1\texttt{+}n_2\texttt{+...+}n_k\texttt{+(sum [])} \end{array}$$

En résumé, on peut donner une spécification équationnelle par cas de construction de la somme des éléments d'une liste :

```
(sum [])       =  0
(sum (n::ns))  =  n + (sum ns)
```

On en déduit immédiatement la définition récursive :

```
let rec sum (ns:int list) : int =
  match ns with
    [] -> 0
  | n::ns' -> n + (sum ns')
```

On parle ici de définition par *récurrence structurelle* puisque

1. la fonction est définie selon la structure de la liste (cas de constructions)
2. l'appel récursif se fait sur une sous-liste de la liste passée en argument (la suite de la liste)

Notez que cette définition récursive est correcte puisque la sous-liste à laquelle s'applique l'appel récursif est de taille plus petite (au sens de sa longueur) que la liste traitée. Cela donne la même garantie de terminaison que l'appel récursif sur $n-1$ (pour n strictement positif).

On peut vérifier, dans notre modèle d'évaluation équationnel, que la définition OCaml de `sum` satisfait la spécification équationnelle :

```
(sum [])       =  (match [] with [] -> 0 | ...)
               =  0
(sum (x::xs))  =  (match (x::xs) with [] -> 0 | n::ns' -> n + (sum ns')
               =  x + (sum xs)
```

On peut donc se fier aux équations pour suivre l'évaluation d'une application de `sum` :

```
(sum (5::4::3::[]))  =  5 + (sum (4::3::[]))
                     =  5 + (4 + (sum (3::[])))
                     =  5 + (4 + (3 + (sum [])))
                     =  5 + (4 + (3 + 0))
                     =  5 + (4 + 3)
                     =  5 + 7
                     =  12
```

Récurence terminal Le concept de récurrence terminal s'applique également aux fonctions définies par récurrence structurelle sur les listes (nous dirons plus simplement «par récurrence sur les listes»). C'est le même principe que celui mis en œuvre pour les entiers.

Par exemple, voici une définition utilisant la récurrence terminale pour la fonction `sum`

```
let sum (ns:int list) : int =
  let rec loop ns r =
    match ns with
      [] -> r
    | n::ns' -> (loop ns' (n+r))
  in
    (loop ns 0)
```

> Dans la suite de ce paragraphe nous utiliserons l'abréviation `[x1; x2; ...; xn]` pour écrire une liste de la forme `x1::x2::...::xn::[]`. Mais n'oubliez pas que ce n'est qu'une abréviation qui masque l'application du constructeur `::`.

Fonction standard : la concaténation La fonction de *concaténation* sur les listes consiste à mettre *«bout-à-bout»* deux listes. Appelons `app` (pour *append*) cette fonction. On peut la caractériser par les *schémas d'équations* suivants :

```
(app [] [y1; ..; ym])                 =  [y1; ..; ym]
(app [x1; x2; ..; xn] [y1; ..; ym])   =  [x1; x2; ..; xn; y1; ..; ym]
```

Appelons `ys` la liste notée `[y1; ..; ym]`. La première équation dit que : pour toute liste `ys`, l'expression `(app [] ys)` a pour valeur celle de `ys` :

```
(app [] ys)  =  ys
```

Pour la seconde équation : rappelons que la liste notée `[x1; x2; ..; xn]` est égale à `x1::[x2; ..; xn]` ; de même, la liste notée `[x1; x2; ..; xn; y1; ..; ym]` est égale à `x1::[x2; ..; xn; y1; ..; ym]`. La seconde équation peut donc se réécrire en :

```
(app (x1::[x2; ..; xn]) [y1; ..; ym])  =  x1::[x2; ..; xn; y1; ..; ym]
```

Appelons `zs` la liste notée `[x2; ..; xn]` (attention : elle commence avec `x2`). Ici, intervient la remarque essentielle : la liste notée `[x2; ..; xn; y1; ..; ym]` correspond à la concaténation des listes `[x2; ..; xm]` et `[y1; ..; ym]`, c'est-à-dire à la valeur de l'expression `(app [x2; ..; xm] [y1; ..; ym])`. Ainsi, en utilisant les noms `zs` et `ys`, la seconde équation s'écrit :

```
(app (x1::zs) ys)  =  x1::(app zs ys)
```

En résumé, la concaténation est définie par les deux équations

$$\begin{cases} \texttt{(app [] ys)} &= \texttt{ys} \\ \texttt{(app (x::zs) ys)} &= \texttt{x::(app zs ys)} \end{cases}$$

Ces deux équations nous donnent une définition par cas de constructeur (sur le premier argument) de la fonction de concaténation des listes. C'est une définition récursive puisque la fonction définie apparaît à gauche et à droite dans la seconde équation. En utilisant le filtrage et les définitions récursives, on peut transcrire dans notre langage la définition de la concaténation des listes de la manières suivante :

```
let rec app (xs:'a list) (ys:'a list) : ('a list) =
match xs with
    [] -> ys
    | x::zs -> x :: (app zs ys)
```

Exemple (schématique) d'application :

```
(app [x1; x2; x3] [y1; y2; y3; y4])  =  x1::(app [x2;x3] [y1; y2; y3; y4])
                                     =  x1::x2::(app [x3] [y1; y2; y3; y4])
                                     =  x1::x2::x3::(app [] [y1; y2; y3; y4])
                                     =  x1::x2::x3::[y1; y2; y3; y4]
```

L'opération de concaténation est prédéfinie dans le langage OCAML par l'opérateur infixe noté `@` : l'expression `[x1; ..; xn]@[y1; ..; ym]` a la même valeur que celle de `(app [x1; ..; xn] [y1; ..; ym])`.

La concaténation permet d'ajouter un élément en fin de liste. C'est-à-dire de définir la fonction `snoc`, de type `'a list -> 'a -> 'a list` telle que (schématiquement) : `(snoc [x1; ...; xn] y)` = `[x1; ...; xn; y]` Pour obtenir la définition de `snoc`, il suffit de remarquer que `[[x1; ...; xn; y]`=`[x1; ...; xn]@[y]`.

```
let snoc (xs:'a list) (x:'a) = xs@[x]
```

Mais attention, contrairement à la primitive d'ajout en tête de liste, la fonction d'ajout en fin de liste a un coût dépendant de la longueur de la liste à laquelle on ajoute un élément. Nous y reviendrons.

> L'opérateur de *concaténation* de liste est de type `'a list -> 'a list -> 'a list`. **Ce n'est pas le type de l'opérateur `::`.**

Remarque : bien qu'il soit toujours possible de mettre une fonction récursive sous forme récursive terminale, nous verrons au travers de nos exemples que lorsque la fonction doit calculer une structure complexe (par exemple ici une liste), l'application sans réfléchir de la "recette" proposée ci-avant ne produit pas toujours une fonction ayant le même comportement. Par exemple la fonction :

```
let badapp (xs:'a list) (ys:'a list) : ('a list) =
  let rec loop xs r =
    match xs with
    [] -> r
    | x::zs -> (loop zs (x::r))
  in
  loop xs ys
```

Nous avons pris `ys` comme valeur initiale de l'accumulateur car nous aurons qu'alors `(badapp [] ys)`=`ys`. Mais, en général, la fonction `badapp` appliquée aux listes `[x1; x2; x3] [y1; y2; y3; y4]` produit la liste `[x3; x2; x1; y1; y2; y3; y4]`. Ce n'est pas du tout la fonction de concaténation des listes. Nous reviendrons sur ce point.

Le module `List` de la bibliothèque standard contient un nombre important de fonctions utilitaires sur les listes. Elles sont en général de type polymorphe sur les éléments des listes. Citons :

`List.length : 'a list -> int` qui donne le nombre d'éléments de son argument, sa *longueur*.

`List.mem : 'a -> 'a list -> bool` qui donne la valeur `true` si et seulement si son premier argument appartient à la liste donnée en second argument.

`List.nth : 'a list -> int -> 'a` telle que `(nth xs i)` donne l'élémént en `i`-ème position dans la liste `xs`, s'il existe. Le premier élément est à la position 0. On obtient l'exception `Failure "nth"` si `i` est supérieur ou égal à la longueur de `xs` ou l'exception `Invalid_argument "List.nth"` si `i` est négatif.

`List.rev : 'a list -> 'a list` telle que `(List.rev [x1; ..; xn])` donne la liste `[xn; ..; x1]`.

etc.

À titre d'exemples, voici les définitions de ces fonctions.

La fonction `length` : calculer la longueur d'une liste, c'est compter son nombre d'éléments. Si l'on considère les deux cas de construction des listes, on a :

— la liste `[]` ne contient aucun élément ;

— la liste `x::xs` contient un élément de plus que la liste `xs`.

La fonction `length` satisfait donc les deux équations :

$$\begin{cases} \texttt{(length [])} & = & 0 \\ \texttt{(length (x::xs))} & = & 1 + \texttt{(length xs)} \end{cases}$$

D'où la définition :

```
let rec length (xs : 'a list) : int =
   match xs with
     [] -> 0
   | _::xs -> 1+(length xs)
```

Comme on n'a pas eu besoin ici de la valeur du premier élément, on a utilisé le motif universel (`_`).

Cette fonction admet la définition récursive terminale suivante :

```
let length (xs : 'a list) : int =
  let rec loop (xs : 'a list) (r:int) : int =
    match xs with
      [] -> r
    | _::xs -> (loop xs (r+1))
  in
    (loop xs 0)
```

La fonction `mem` : pour déterminer si un élément `z` appartient ou non à une liste, on raisonne par cas de construction de la liste :

— si la liste est vide, alors `z` n'appartient pas à la liste ;

— si la liste est de la forme `x::xs` alors, il y a deux possibilités :

  — le premier élément de la liste est égal à `z` et donc `z` appartient à `x::xs`

  — `z` appartient à la liste `xs`

On peut donc poser que `mem` satisfait les *équations conditionnelles* suivantes :

$$\begin{cases} \texttt{(mem z [])} & = & \texttt{false} & \\ \texttt{(mem z (x::xs))} & = & \texttt{true} & \text{si } \texttt{x} = \texttt{z} \\ \texttt{(mem z (x::xs))} & = & \texttt{(mem z xs)} & \text{sinon} \end{cases}$$

D'où la définition :

```
let rec mem (z:'a) (xs:'a list) : bool =
  match xs with
    [] -> false
  | x::xs -> if (x=z) then true else (mem z xs)
```

On peut également dire que `z` est un élément de `x::xs` si et seulement si `x`=`z` ou `z` est un élément de `xs`. Ce qui donne la définition :

```
let rec mem (z:'a) (xs:'a list) : bool =
  match xs with
    [] -> false
  | x::xs -> (x=z) || (mem z xs)
```

La fonction (partielle) `nth` : quoique de réalisation assez simple, l'élaboration de cette fonction réclame un peu d'attention.

Intuitivement, l'application `(nth [x0; ..; xn] i)` a pour valeur `xi`, si `i` est compris entre 0 et n. Pour `i` négatif ou strictement supérieur à n, l'application n'a pas de valeur ; de même, `(nth [] i)` n'a pas de valeur.

On peut remarquer que :

- l'indice du premier élément d'une liste est 0, donc l'application `(nth [x0; ..; xn] 0)` a pour valeur `x0` ;
- si l'indice d'un élément dans la liste `[x1; ..; xn]` est i alors, il est $i+1$ dans la liste `[x0; x1; ..; xn]` ; réciproquement, si un élément est d'indice $i+1$ dans la liste `[x0; x1; ..; xn]`, il est d'indice i dans la liste `[x1; ..; xn]`. Donc, si $i > 0$ est l'indice d'un élément dans `[x0; x1; ..; xn]`, son indice est $i-1$ dans `[x1; ..; xn]`.

De ces remarques, on déduit les équations conditionnelles suivantes

`(nth (x::xs) i)`	=	`x`	si `i`=0
`(nth (x::xs) i)`	=	`(nth xs (i-1))`	sinon

On déduit de cette analyse la définition suivante qui déclenche une exception lorsque la valeur n'est pas définie :

```
let rec nth (xs:'a list) (i:int) : 'a =
  match xs with
    [] -> raise (Failure "nth")
  | x::xs -> if (i=0) then x else (nth xs (i-1))
```

Si l'on regarde la spécification de la fonction `List.nth`, il y a en fait deux cas où la fonction n'a pas de valeur : le cas où l'indice est supérieur ou égal à la longueur de la liste et le cas où l'indice est négatif. Le premier cas est signalé par l'exception `(Failure "nth")` ; le second, par l'exception `(Invalid_argument "List.nth")`.

On peut intercepter le second cas avant d'activer la recherche récursive par l'application d'une fonction locale, comme nous l'avions fait, par exemple, pour la fonction `pow` (2). Cela donne :

```
let nth (xs:'a list) (i:int) : 'a =
  let rec loop (xs:'a list) (i:int) : 'a =
    match xs with
      [] -> raise (Failure "nth")
    | x::xs -> if (i=0) then x else (loop xs (i-1))
  in
    if (i < 0) then raise (Invalid_argument "List.nth")
    else (loop xs i)
```

Le déclenchement de l'exception `(Failure "nth")` est cohérent avec la spécification : « On obtient l'exception `Failure "nth"` si `i` est supérieur ou égal à la longueur de `xs` ». En effet si, `i` étant positif, la succession d'appels récursifs atteint la liste vide sans avoir annulé `i`, puisque l'indice `i` est décrémenté de 1 à chaque appel récursif, c'est que l'indice était supérieur à la longueur de la liste.

La fonction `rev` : schématiquement, l'application `(rev [x0; x1; ..; xn])` a pour valeur `[xn; ..; x1; x0]`. On peut remarquer que :

- `(rev [])` a pour valeur `[]` ;
- la liste `[xn; ..; x1]` est la valeur de `(rev [x1; ..; xn])` ;
- la liste `[xn; ..; x1; x0]` est la liste obtenue en ajoutant `x0` à la fin de la liste `[xn; ..; x1]`, c'est-à-dire, la valeur de la concaténation de `[xn; ..; x1]` et `[x0]`, c'est-à-dire, la valeur de la concaténation des listes `(rev [x1; ..; xn])` et `[x0]`.

On peut donc poser les deux équations suivantes :

$$\begin{cases} \texttt{(rev [])} & = & \texttt{[]} \\ \texttt{(rev (x::xs))} & = & \texttt{(rev xs) @ (x::[])} \end{cases}$$

Cela donnerait la définition suivante :

```
let rec rev (xs:'a list) : 'a list =
  match xs with
    [] -> []
  | x::xs -> (rev xs) @ [x]
```

Cette définition n'est toutefois pas très satisfaisante. Illustrons le en déroulant les étapes d'évaluation de `(rev [x1; x2; x3; x4])` :

```
(1)   (rev [x1; x2; x3; x4])  =  (rev [x2; x3; x4]) @ (x1::[])
(2)                           =  ((rev [x3; x4]) @ (x2::[])) @ (x1::[])
(3)                           =  (((rev (x4::[])) @ (x3::[])) @ (x2::[])) @ (x1::[])
(4)                           =  ((((rev []) @ (x4::[])) @ (x3::[])) @ (x2::[])) @ (x1::[])
(5)                           =  (((([] @ (x4::[])) @ (x3::[])) @ (x2::[])) @ (x1::[])
(6)                           =  (((x4::[]) @ (x3::[])) @ (x2::[])) @ (x1::[])
(7)                           =  ((x4::([] @ (x3::[]))) @ (x2::[])) @ (x1::[])
(8)                           =  ((x4::(x3::[])) @ (x2::[])) @ (x1::[])
(9)                           =  (x4::((x3::[]) @ (x2::[]))) @ (x1::[])
(10)                          =  (x4::x3::([] @ (x2::[]))) @ (x1::[])
(11)                          =  (x4::x3::(x2::[])) @ (x1::[])
(12)                          =  x4::((x3::(x2::[])) @ (x1::[]))
(13)                          =  x4::x3::((x2::[]) @ (x1::[]))
(14)                          =  x4::x3::x2::([] @ (x1::[]))
(15)                          =  x4::x3::x2::(x1::[])
```

Les étapes (1) à (5) correspondent au développement de la fonction `rev` en terme de concaténations. Les étapes (6) à (15) sont les étapes d'évaluation des différentes concaténations mises en attente. C'est ici que le bât blesse : chaque évaluation d'une concaténation reparcours des valeurs déjà envisagées ; par exemple, `x4` est repris 3 fois avant d'arriver à sa place définitive.

Avec un peu de réflexion, on peut faire bien mieux. Souvenons nous de notre malheureuse tentative de donner une définition récursive terminale de la concaténation.

```
let badapp (xs:'a list) (ys:'a list) : ('a list) =
  let rec loop xs r =
    match xs with
    [] -> r
```

```
  | x::zs -> (loop zs (x::r))
in
loop xs ys
```

On peut remarquer que `(badapp [x1;...;xn] [])`=`[xn;...;x1]` qui est la valeur attendue de `(rev [x1;...;xn])`.

Sans utiliser la fonction `badapp`, on peut définir diretement une version récursive terminale de la fonction `rev` :

```
let rev (xs:'a list) : 'a list =
  let rec loop (xs:'a list) (r:'a list) =
    match xs with
      [] -> r
    | x::xs -> (loop xs (x::r))
  in
    (loop xs [])
```

`(rev [x1;x2;x3;x4])` =

On obtient alors une fonction ayant de bien meilleures performances que notre version précédente :

Rappelons que `x4::x3::x2::x1::[]` est l'écriture développée de la forme abrégée `[x4;x3;x2;x1]`.

Schémas d'itération Le module `List` contient également nombre de fonctionnelles qui correspondent à des schémas génériques de traitement ou d'exploration de listes.

Schéma d'application
`List.map : ('a -> 'b) -> 'a list -> 'b list` telle que `(List.map f [x1; ..; xn])` a pour valeur la liste `[(f x1); ..; (f xn)]`.

Schéma de filtrage
`List.filter : ('a -> bool) -> 'a list -> 'a list` telle que `(List.filter p xs)` donne la liste des éléments `xi` de `xs` pour lesquels `(p xi)` vaut `true`.

Schémas d'accumulation
`List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` telle que `(List.fold_right f [x1; ..; xn] a)` donne la valeur de l'expression `(f x1 .. (f xn a)..)`.
`List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` telle que `(List.fold_left f a [x1; ..; xn])` donne la valeur de l'expression `(f (.. (f a x1) ..) xn)`.

À titre d'exemples, voici les définitions de ces itérateurs :

```
let rec map (f : 'a -> 'b) (xs : 'a list) : ('b list) =
  match xs with
    [] -> []
 | x::xs -> (f x)::(map f xs)
let rec filter (p : 'a -> bool) (xs : 'a list) : ('a list) =
  match xs with
    [] -> []
  | x::xs -> if (p x) then x::(filter p xs)
             else (filter p xs)
let rec fold_right (f : 'a -> 'b -> 'b) (xs : 'a list) (b : 'b) : 'b =
  match xs with
    [] -> b
```

```
  | x::xs -> (f x (fold_right f xs b))
let rec fold_left (f : 'a -> 'b -> 'a) (a : 'a) (xs : 'b list) : 'a =
  match xs with
    [] -> a
  | x::xs -> (fold_left f (f a x) xs)
```

Notez que `fold_left` est récursive terminale.

Récursion terminale Très souvent avec les fonctions récursives qui construisent des listes on n'obtient pas directement une version récursive terminale. Nous avons déjà observé cela avec la fonction de concaténation. Nous pouvons à nouveau l'observer avec, par exemple, une tentative d'écrire une version récursive terminale de `map` qui s'appuie sur l'emploi d'une fonction récursive locale avec accumulateur :

```
let map (f:'a -> 'b) (xs:'a list) : ('b list) =
  let rec loop (xs:'a list) (r:'b list) =
     match xs with
       [] -> r
     | x::xs -> (loop xs ((f x)::r))
  in
    (loop xs [])
```

Cette définition ne convient pas : `(map f [x1; ...; xn])` donne la liste `[(f xn); ...; (f x1)]`. L'ordre attendu des éléments est inversé. Qu'à cela ne tienne, pour rétablir l'ordre voulu, il suffit d'inverser la liste résultat de `loop`.

```
let map (f:'a -> 'b) (xs:'a list) : ('b list) =
  let rec loop (xs:'a list) (r:'b list) =
     match xs with
       [] -> (List.rev r)
     | x::xs -> (loop xs ((f x)::r))
  in
    (loop xs [])
```

Ce sera notre manière standard de donner des version récursives terminales de fonctions construisant une liste. Le prix à payer est qu'il faut un parcours de l'argument, puis un parcours de la liste construite par la fonction locale. On dit que l'on obtient le résultat en *2 passes*.

Un itérateur pour les gouverner tous On peut (re)définir les itérateurs `List.map` et `List.filter` comme cas d'application de `List.fold_right`.

La fonction itérée par `List.map` est simplement le constructeur *cons* :

```
let map (f : 'a -> 'b) (xs : 'a list) : 'b list =
   List.fold_right (fun x r -> (f x)::r) xs []
```

La fonction itérée par `List.filter` sélectionne les éléments à retenir pour le résultat final :

```
let filter (p : 'a -> bool) (xs : 'a list) : 'a list =
  List.fold_right (fun x r ->  if (p x) then x::r else r) xs []
```

L'itérateur `List.fold_right` peut permettre une expression très compacte de certaines fonctions. Par exemple, on obtient la fonction de recherche de l'élément maximal d'une liste de la manière suivante :

```
let find_max (xs : 'a list) : 'a =
  match xs with
    [] -> raise (Invalid_argument "find_max")
  | x::xs -> List.fold_right max xs x
```

Notez que `find_max` est une fonction partielle, non définie pour la liste vide.

On peut comparer cette définition à une définition plus directe :

```
let find_max (xs : 'a list) : 'a =
  let rec loop (xs : 'a list) (r : 'a) =
    match xs with
      [] -> r
    | x::xs -> (loop xs (max x r))
  in
    match xs with
      [] -> raise (Invalid_argument "find_max")
    | x::xs -> (loop xs x)
```

On peut noter que la fonction locale `loop` est terminale récursive. Cela indique que, dans ce cas, on peut tout aussi bien utiliser l'itérateur `List.fold_left` :

```
let find_max (xs : 'a list) : 'a =
  match xs with
    [] -> raise (Invalid_argument "find_max")
  | x::xs -> List.fold_left max x xs
```

L'utilisation de `List.fold_right` ou `List.fold_left` est indifférent dès lors que la fonction itérée est associative et commutative. On peut alors préfèrer `List.fold_left` qui est récursive terminale.

`List.fold_left` accumule *«à l'envers»* : on peut définir la fonction `rev` directement avec `fold_left`, version récursive terminale :

```
let rev (xs:'a list) : ('a list) =
  List.fold_left (fun r x -> x::r) [] xs
```

Pour avoir un `map` récursif terminal en utilisant `List.fold_left`. Mais, comme observé ci-dessus, il faut *2 passes* :

```
let map (f:'a -> 'b) (xs:'a list) : ('b list) =
  List.fold_left (fun r x -> x::r) []
      (List.fold_left (fun r x -> (f x)::r) [] xs)
```

Itération non bornée Parmi les fonctionnelles fournies par la bibliothèque standard, il y a celle-ci :

`find : ('a -> bool) -> 'a list -> 'a` telle que `(find p xs)` donne le premier élément `x`de `xs` pour lequel `(p x)` prend la valeur `true`, s'il existe, et déclenche l'exception `Not_found`, sinon.

On pourrait définir cette fonction en utilisant l'itérateur `List.filter` : en effet, le premier élément de `xs` pour lequel `p` prend la valeur `true` et le premier élément du résultat de `(List.filter p xs)`, si ce résultat n'est pas la liste vide. On aurait :

```
let find (p : 'a -> bool) (xs : 'a list) : 'a =
  match (List.filter p xs) with
    [] -> raise Not_found
  | x::_ -> x
```

Notez comment ici, on analyse le résultat de l'application d'une fonction et non plus simplement la forme d'un argument.

Toutefois, cette manière de faire n'est pas très optimale. En effet, `List.filter` explore toute la liste pour construire un résultat dont on ne conserve que le premier élément. Il eût été plus optimal d'arrêter la recherche dès que le premier `x` de `xs` pour lequel `p` prend la valeur `true` a été trouvé. Ce qui donne la définition :

```
let rec find (p : 'a -> bool) (xs : 'a list) : 'a =
  match xs with
    [] -> raise Not_found
  | x::xs -> if (p x) then x else (find p xs)
```

Il existe deux *conditions d'arrêt* pour cette fonction :

1. La liste est vide, au quel cas la recherche a échoué.
2. La fonction `p` prend la valeur `true` pour le premier élément de la liste, auquel cas, ce premier élément est le résultat recherché.

De surcroît, cette définition est maintenant récursive terminale.

Les itérateurs `List.map` et `List.filter` implémentent des *itération bornées* sur les listes. Le nombre d'étapes d'itérations (*i.e.* le nombre d'appels récursifs) de l'application de ces itérateurs est égale à la taille des listes traitées.

En revanche, les fonctions de recherche d'une valeur dans une structure sont des exemples d'itérations *non bornées*. C'est-à-dire, d'itérations qui n'ont pas nécessairement besoin d'explorer la totalité des structures traitées pour produire un résultat.

Les exceptions comme contrôle Les itérateurs `fold_right` ou `fold_left` permettent de définir un grand nombre de fonctions définies par itération, c'est-à-dire par récurrence, sur une liste (voir les exemples de `map`, `filter`, `find_max`, etc.) Toutefois ces itérateurs réalisent des itération bornées. Il semblerait donc que l'on n'ait aucun intérêt à les utiliser pour des fonctions ne réclamant qu'une utilisation bornée.

Considérons l'exemple de la fonction `exists` de type `(p:'a -> bool) -> ('a list) -> bool` qui donne `true` si la liste donnée en argument contient un élément `x` tel que `(p x)` vaut `true` et qui donne `false` sinon. Cette fonction peut être réalisée par une itération non bornée :

```
let rec exists (p:'a -> bool) (xs:'a list) : bool =
  match xs with
    [] -> false
  | x::xs -> if (p x) then true else (exists p xs)
```

Notons que cette définitions est équivalente à

```
let rec exists (p:'a -> bool) (xs:'a list) : bool =
  match xs with
    [] -> false
  | x::xs -> (p x) || (exists p xs)
```

En effet, il existe un `xi` dans `[x1; ...; xn]` tel que `(p xi)` vaut `true` si et seulement si `(p x1) || ...|| (p xn)` vaut `true`. Notez que cette définition possède toujours la propriété d'être une itération non bornée car (comme nous l'avons souligné en 1, page 7) l'opérateur de disjonction n'évalue pas son second argument lorsque le premier vaut `true`. Ici, il n'y a pas d'appel récursif à `exists` dès le premier `x` de `xs` tel que `(p x)` vaut `true`.

Avec cette nouvelle définition, on serait tenté d'utiliser un itérateur pour définir `exists` par itération de la disjonction `||` :

```
let exists (p:'a -> bool) (xs:'a list) : bool =
  List.fold_left (fun r x -> r || (p x)) false xs
```

Toutefois, malgré la propriété d'évaluation de la disjonction, cette définition réalise une itération bornée : toute la liste est parcourue avant que le résultat de `fold_left` ne soit donné (à vérifier en exercice).

Pour obtenir une itération non bornée il faut être capable *d'interrompre* le fil ordinaire de l'évaluation de `List.fold_left`. En programmation impérative, on peut réaliser cela avec l'instruction `return` qui peut interrompre l'évaluation d'une fonction même à l'intérieur d'une boucle pour délivrer un résultat. Si en style fonctionnel on ne dispose pas de `return`, on dispose d'un autre moyen d'interrompre le fil d'une évaluation : le déclenchement d'une exception.

Le langage OCaml fournit des exceptions prédéfinies mais il permet également au programmeur de définir ses propres exceptions avec le mot clé `exception`. Pour notre propos, définissons donc :

```
exception Found
```

Si la fonction itérée par `List.fold_left` déclenche l'exception `Found` dès qu'elle trouve un élément `x` de la liste tel que `(p x)` vaut `true` alors on a gagné : l'évaluation de `List.fold_left` est interrompue et on peut récupérer cette exception pout donner le résultat `true`. Voici un exemple de mise en œuvre de cette possibilité :

```
let exists (p:'a -> bool) (xs:'a list) : bool =
  try
    List.fold_left (fun r x -> if (p x) then (raise Found) else r) false xs
  with
    Found -> true
```

Si la liste `xs` contient un élément `x` tel que `(p x)` vaut `true` alors l'exception `Found` est déclenchée, elle est rattrappée par le `with Found` et la valeur de l'application est `true` ; sinon, `List.fold_left` va à son terme et donne la valeur de `r` qui a été initialisée à `false` et qui reste avec cette valeur tout au long de l'évaluation de `List.fold_left`.