Programmation fonctionnelle

M323 — Programmer de manière fonctionnelle

Jérôme Frossard

EPAI Fribourg

19 février 2026

Plan

Plan

Paradigme de programmation

La programmation impérative n'est pas la seule manière de concevoir la programmation.

Les différentes manières de concevoir la programmation sont appelées des paradigmes de programmation.

On peut distinguer deux grands paradigmes :

- Le paradigme impératif qui recouvre notamment la programmation structurée et la programmation orientée objet.
- Le paradigme déclaratif qui recouvre notamment la programmation fonctionnelle et la programmation logique.

Alors que dans le paradigme impératif, un programme décrit ce que la machine doit faire, étape par étape, pour parvenir au résultat, dans le paradigme déclaratif, un programme décrit plutôt le résultat attendu, mais pas la manière exacte d'effectuer le calcul.

Programmation fonctionnelle (I/II)

En programmation impérative, un programme est décrit par une **séquence d'instructions** (affectation, structures de contrôle, etc.) qu'une machine doit **exécuter** pour éventuellement **produire un résultat**. Il se peut que le programme ne termine pas (boucle infinie).

En programmation fonctionnelle, un programme est décrit par une **expression** qu'une machine doit **réduire** pour éventuellement parvenir à une **forme normale** correspondant au résultat. Là encore, il se peut que le programme ne termine pas (réduction infinie).

Toutefois, le fait que la réduction d'une expression corresponde à son évaluation n'est vrai que si tous les termes peuvent être remplacés par leur valeur sans que cela change le résultat. Cette propriété est appelée **transparence référentielle**.

En garantissant cette propriété, la **programmation fonctionnelle pure** permet de **raisonner** directement **sur la syntaxe (le texte)** de n'importe quelle partie du programme , sans avoir à en faire la trace comme en programmation impérative.

Programmation fonctionnelle (II/II)

Pour une variable, la transparence référentielle implique l'**immutabilité**. La valeur d'une variable est fixée lors de sa définition, puis ne change plus jamais.

Pour une fonction, la transparence référentielle implique que :

- La valeur de cette fonction ne dépend que de la valeur de ses paramètres. Celle-ci ne dépend donc pas de l'état du système (variable non locale, horloge temps réel, tampon d'entrée/sortie, etc.)
- Elle n'a pas d'effet de bord (mutation d'une variable non locale, entrée/sortie, etc.)

En programmation fonctionnelle, une telle fonction est appelée une **fonction pure**.

Tous les langages fonctionnels encouragent l'immutabilité des variables, mais seuls les langages fonctionnels purs l'imposent et interdisent les fonctions non pures.

Remarques : On peut dire qu'un programme fonctionnel décrit le résultat car si la transparence référentielle est garantie, l'expression qui décrit ce programme est sémantiquement équivalente à sa forme normale et que cette forme normale correspond au résultat. On décrit le calcul, mais on ne dit pas comment l'effectuer.

Langages et styles de programmation

Un langage de programmation, par sa syntaxe (et parfois par l'environnement d'exécution), supporte plus ou moins bien certains styles de programmation. Par exemple :

- Java ou C# supportent clairement le style impératif (structuré/orienté objet). Même avec des concepts empruntés au style fonctionnel (lambda, stream), on continue à raisonner en termes d'instructions.
- Clojure ou F#, même s'ils n'imposent pas la pureté des fonctions, supportent clairement le style fonctionnel. On raisonne en termes d'expressions à réduire, pas d'instructions à exécuter.

On peut utiliser le style orienté objet en C, ou le style fonctionnel en Java, mais la syntaxe n'offre aucune aide, et certaines choses sont impossibles (p. ex. pas d'optimisation de la récursion terminale en Java).

Certains langages, comme Kotlin ou Scala, cherchent, avec plus ou moins de succès, à supporter à la fois les styles impératif et fonctionnel, parfois en composant avec les limitations de l'environnement d'exécution (JVM, .NET CLR, etc.)

Style fonctionnel et langage fonctionnel pur

Quel que soit le langage de programmation utilisé, on peut réaliser un sous-programme dans le style fonctionnel en s'assurant que :

- Toutes les variables sont immuables.
- Toutes les fonctions sont pures (pas d'effets de bord, etc.).

Mais avec un **langage impératif** (Java, C#, etc.), un **langage fonctionnel non pur** (OCaml, F#, Clojure, etc.) ou un **langage mixte** (Kotlin, Scala, etc.), cela repose sur la **discipline des programmeurs** : rien ne garantit que ces propriétés soient appliquées, ni même qu'elles soient applicables à l'ensemble du programme (p. ex. aux opérations d'entrée/sortie).

Pour avoir la garantie de la transparence référentielle, on utilise un **langage fonctionnel pur** tel que *Haskell* ou *PureScript* (que nous allons pratiquer) qui impose ces propriétés d'**immutabilité** et de **pureté des fonctions**, garantissant ainsi la **transparence référentielle**.

Remarque : Un langage fonctionnel non pur est un langage qui supporte principalement le style fonctionnel, mais sans imposer la pureté des fonctions. Un langage mixte supporte plusieurs styles.

Structures de données persistantes

À cause de l'immutabilité, toute transformation appliquée à une variable implique la création d'une nouvelle valeur en mémoire lors de l'exécution.

C'est vrai pour les nombres, mais c'est aussi vrai pour des structures de données complexes (strings, lists, sets, maps, queues, etc.). Ajouter un élément à une liste implique (au moins en principe) de créer une copie de cette liste avec un élément supplémentaire.

Une implémentation naïve de ces structures de données conduit inévitablement à d'importants problèmes de performances et d'utilisation de la mémoire (on connaît bien le problème de la concaténation des chaînes en Java).

Les **structures de données persistantes** sont une solution à ce problème. Pour appliquer une transformation à une structure persistante, on ne crée pas une copie complète, mais une nouvelle version de cette structure en partageant autant que possible les données communes. Contrairement au `StringBuilder` de Java, on conserve ainsi l'immutabilité, tout en réduisant la duplication des données au minimum.

Plan

Réduction et forme normale

Nous avons vu qu'une expression est une construction syntaxique qui peut être évaluée pour produire une valeur. C'est toujours vrai, mais :

- En programmation impérative, l'évaluation consiste à exécuter des opérations et des appels de fonctions pour produire une valeur.
- En programmation fonctionnelle, l'évaluation consiste à **réduire** l'expression jusqu'à ce qu'elle ne puisse plus l'être. On parle alors de **forme normale**.

Grâce à la transparence référentielle, une expression et sa forme réduite sont équivalentes : elles représentent la même valeur. Par exemple, 4 + 1 et 5. De telles expressions peuvent donc être remplacées l'une par l'autre.

Le fait qu'une expression et sa forme réduite soient équivalentes ne veut pas dire que la réduction se fait sans travail. Comme il faut une machine pour exécuter les instructions d'un programme impératif, il faut une machine pour réduire (appliquer les règles de réduction) l'expression d'un programme fonctionnel. Dans les deux cas, il faut de l'énergie.

Expressions spéciales

Comme nous l'avons déjà vu, une expression peut être :

- Une valeur littérale
- Une variable
- Une opération dont les opérandes sont des expressions
- Une application de fonction dont les paramètres sont des expressions

Dans les langages fonctionnels, il existe également des expressions spéciales qui n'entrent pas dans ces catégories (ce ne sont ni des opérations ni des applications de fonction). On peut mentionner notamment :

- Expression de liaison (let-in)
- Expression conditionnelle (if-then-else)
- Expression de sélection par motif (case-of)

Expression conditionnelle (if-then-else)

L'expression if-then-else permet de choisir entre deux expressions selon la valeur d'une expression booléenne.

Contrairement à l'instruction if en programmation impérative, l'expression if-then-else :

- produit une valeur : chaque branche est une expression (pas une séquence d'instructions).
- doit toujours avoir une branche else (une expression doit toujours produire une valeur).
- n'a, par construction, pas d'effet de bord.

Exemple en PureScript :

```
absolute :: Int -> Int
absolute x = if x < 0 then -x else x
```

La réduction de `if x < 0 then -x else x` donne `-x` si `x < 0`, et `x` dans le cas contraire.

Expression de liaison (let-in)

L'expression let-in permet de définir des liaisons locales (variables ou fonctions) pour une expression. La valeur de l'expression let-in est celle de l'expression qui suit le `in`.

Syntaxe générale :

```
let <bloc de définitions>
in <expression>
```

Exemple en PureScript :

```
circleArea :: Number -> Number
circleArea r =
    let pi = 3.14159
        rr = r * r
    in pi * rr
```

Remarque : Les liaisons locales sont immuables, comme toutes les variables en programmation fonctionnelle pure, afin de garantir la transparence référentielle.

Expression de sélection par motif (case-of)

L'expression de sélection par motif (*pattern matching*) permet de sélectionner une branche selon la structure d'une valeur. Sa valeur est celle de la branche sélectionnée.

Cette expression permet de :

- Décomposer une structure de données (liste, record, type algébrique, etc.).
- Tester la forme de la valeur (constructeur de type utilisé).
- Extraire les composants de la valeur.

Exemple en PureScript :

```
describe :: Maybe Int -> String
describe m = case m of
    Nothing -> "pas de valeur"
    Just x  -> "la valeur est " <> show x -- déconstruit la valeur Maybe et lie x à sa
```

Remarque : Le compilateur vérifie que tous les cas sont couverts (exhaustivité). Si un cas est oublié, une erreur de compilation est émise.

Plan

Fonction pure

En **mathématiques**, une fonction est une relation qui associe à chaque valeur d'un **ensemble de départ** exactement une valeur d'un **ensemble d'arrivée**.

En programmation **impérative**, une fonction :

- associe un nom et une liste de paramètres à une **séquence d'instructions**.
- peut être **appelée** et les valeurs des paramètres sont passées au moment de l'appel.
- n'est pas toujours une fonction au sens mathématique (effets de bord, etc.).

En programmation **fonctionnelle** pure, une fonction :

- associe un nom et une liste de paramètres à **une seule expression**.
- peut être **appliquée** aux valeurs des paramètres (et non appelée).
- peut être remplacée par son expression en substituant les paramètres par leur valeur.
- est une fonction au sens mathématique (fonction pure).

Définition d'une fonction

Dans les langages de la famille ML (PureScript, F#, Haskell, OCaml), la définition d'une fonction a la forme suivante :

```
<nom> <param1> <param2> ... <paramN> = <expression>
```

La syntaxe est particulièrement simple : il n'y a ni parenthèses autour des paramètres formels ni virgule pour les séparer, et le « corps » est constitué d'une seule expression.

Par exemple :

```
inc x = x + 1

addThreeValues a b c = a + b + c
```

Les plus attentifs auront remarqué qu'il n'y a pas de spécification de type, alors que Purescript est statiquement typé. Cela fonctionne grâce à l'inférence de type. Voyons cela de plus près.

Type d'une fonction (I/II)

Si l'on utilise la commande `:type inc` dans le REPL de PureScript, on obtient `Int -> Int`. On reconnait la notation mathématique qui représente la relation entre l'ensemble de départ et l'ensemble d'arrivée d'une fonction.

Ce n'est bien sûr pas un hasard. Le type `Int` représente bien un ensemble de valeurs. `inc` est une fonction qui met en relation une valeur de type `Int` (l'ensemble de départ) à une seule valeur de type `Int` (l'ensemble d'arrivée).

L'**inférence de type** permet au compilateur de déterminer le type de la fonction à partir de celui de l'opérateur `+`. Cet opérateur exige que les deux opérandes soient du même type et produit une valeur de même type que celui de ces opérandes.

Puisque `1` est un littéral de type `Int`, `x` et `x + 1` sont par conséquent aussi de type `Int`. Si l'on avait écrit `1.0` (un littéral de type `Number`) à la place de `1`, le type de la fonction aurait été `Number -> Number`.

Type d'une fonction (II/II)

Pour la fonction `addThreeValues`, c'est un peu plus compliqué. Son type est également inféré à partir de l'opérateur `+`, mais la commande `:type` renvoie :

```
forall (a :: Type). Semiring a => a -> a -> a -> a
```

On distingue plusieurs parties dans ce type :

- Un quantificateur pour le type a : `forall (a :: Type).` (ou juste `forall a.`)
- Le type de la fonction proprement dit : `a -> a -> a -> a`
- Une contrainte de type exprimée à l'aide d'une classe de type : `Semiring a =>`

Le type de la fonction dit que la fonction prend trois paramètres de type a et produit une valeur de type a, mais sans préciser ce qu'est ce type a (seulement que les paramètres et la valeur produite sont de même type). C'est un exemple de **polymorphisme paramétrique** : `a` est similaire au paramètre `T` de `List<T>` en Java.

La **contrainte** dit que le type `a` doit supporter les opérations définies par la classe de type `Semiring` : `+` et `*`. Par exemple, `Int` et `Number`.

Spécification du type d'une fonction

Dans la plupart des cas, l'inférence de type permet de déterminer le type d'une fonction, mais la spécification des types permet à la programmeuse de préciser son intention.

Dans les langages de la famille ML, le type d'une fonction (ou d'une variable) est spécifié juste au-dessus de sa définition. Par exemple :

```
inc :: Int -> Int
inc x = x + 1

addThreeValues :: forall a. Semiring a => a -> a -> a -> a
addThreeValues a b c = a + b + c
```

Selon notre intention, nous pourrions décider que la fonction `addThreeValues` ne s'applique qu'à des `Int`. Son type serait alors :

```
addThreeValues :: Int -> Int -> Int -> Int
addThreeValues a b c = a + b + c
```

Liaisons, variables et fonctions

En programmation fonctionnelle, il n'y a pas vraiment de différence entre une fonction et une variable. Il s'agit dans tous les cas d'un nom lié à une valeur, c'est pourquoi on les appelle **liaisons (*binding*)**.

De plus, les **fonctions** sont des **entités de première classe**. Lorsque la fonction est définie, on peut utiliser son nom comme une variable dont la **valeur est la fonction elle-même**. Cette valeur peut alors être liée à une autre variable ou à un paramètre d'une fonction.

Par exemple :

```
inc :: Int -> Int
inc x = x + 1

inc2 :: Int -> Int
inc2 = inc
```

La variable `inc2` est maintenant une fonction au même titre que `inc`, et peut être utilisée de la même manière.

Application d'une fonction

En programmation fonctionnelle, il n'y a pas à proprement parler d'appel de fonction. Un appel suppose le **passage de contrôle** au sous-programme appelé (call), puis la restitution du contrôle à l'appelant (return). Or ce sont là des **instructions**.

En programmation fonctionnelle, une fonction est **appliquée** à une valeur. Puisque l'application d'une fonction est une expression, sa valeur est obtenue par la **réduction** de cette expression.

Par exemple, si on a la fonction :

```
inc :: Int -> Int
inc x = x + 1
```

L'expression `inc 4` est réduite de la manière suivante :

```
inc 4  ->  4 + 1  ->  5
```

On ne peut pas réduire `5`. La **forme normale** de l'expression `inc 4` est donc `5`.

Curryfication (I/II)

La manière de noter le type d'une fonction à un seul paramètre est assez intuitive. Mais pourquoi une fonction à deux paramètres se note-t-elle :

```
Int -> Int -> Int
```

plutôt que, par exemple :

```
(Int, Int) -> Int
```

Cette notation vient du fait que dans les langages de la famille ML, **toutes les fonctions** prennent **exactement un paramètre**. On dit qu'elles sont **curryfiées**.

La **curryfication** est la transformation d'une fonction à plusieurs paramètres en une fonction à un paramètre dont l'expression associée est une fonction qui prend le reste des paramètres, à laquelle on applique la même transformation.

Curryfication (II/II)

Par exemple, une fonction à quatre paramètres est transformée en une fonction à un paramètre dont la valeur est une fonction à trois paramètres. Mais cette fonction à trois paramètres est elle aussi une fonction à un paramètre dont la valeur est une fonction à deux paramètres, et ainsi de suite.

Dans la fonction suivante, les parenthèses mettent cela en évidence.

```
compoundInterest :: Number -> (Number -> (Number -> (Number -> Number)))
compoundInterest periods effectiveRate years capital =
    capital * ((1.0 + r) `pow` (periods * years) - 1.0)
    where
        r = effectiveToPeriodic periods effectiveRate
```

Comme la flèche (->) est associative à droite, la notation du type est parfaitement équivalente à la notation habituelle :

```
compoundInterest :: Number -> Number -> Number -> Number -> Number
```

Expression lambda

Une **expression lambda** est une **fonction anonyme** qui peut être considérée comme un littéral de type fonction, comme 42 est un littéral de type Int. Sa forme générale est :

```
\ <param> -> <expression>
```

Si une fonction n'est pas récursive (voir plus loin), on peut remplacer sa définition par la définition d'une variable dont la valeur est une expression lambda. Par exemple, la fonction `compoundInterest` peut être définie de la manière suivante :

```
compoundInterest :: Number -> Number -> Number -> Number -> Number
compoundInterest = \ periods -> \ effectiveRate -> \ years -> \ capital ->
    let r = effectiveToPeriodic periods effectiveRate
    in capital * ((1.0 + r) `pow` (periods * years) - 1.0)
```

Comme dans la spécification d'un type, la flèche est associative à droite.

Remarque : Le symbole \ dans les expressions lambda en PureScript (et en Haskell) évoque la lettre grecque λ (lambda) de la notation du λ-calcul de Church : λ <param>.<expression>

Application partielle

À l'inverse de la flèche (->), l'application d'une fonction associe à gauche. Une application de `compoundInterest` avec les arguments 12.0, 0.03, 5.0, 20000.0, se lit :

```
result :: Number
result = ((((compoundInterest 12.0) 0.03) 5.0) 20000.0)
```

Si on décompose l'expression, on obtient, par exemple :

```
monthlyCompound :: Number -> Number -> Number -> Number
monthlyCompound = compoundInterest 12.0

monthlyCompoundAt3Pct :: Number -> Number -> Number
monthlyCompoundAt3Pct = monthlyCompound 0.03

result :: Number
result = monthlyCompoundAt3Pct 5.0 20000.0
```

Les valeurs des variables `monthlyCompound` et `monthlyCompoundAt3Pct` sont des fonctions qui résultent d'une **application partielle**.

Plan

Pas de structure de boucle

Vous l'aurez sans doute remarqué : dans les expressions spéciales, nous avons vu une expression conditionnelle (if-then-else), mais nous n'avons pas encore parlé de structure de boucle.

Ce n'est pas un oubli. En effet, une structure de boucle nécessite la mutation d'une variable (la variable de boucle) et une instruction de saut, il ne peut donc exister de structure de boucle en programmation fonctionnelle pure.

Pourtant, dans de nombreux cas, nous avons besoin de **répéter une opération** :

- Parcourir les éléments d'une liste,
- Agréger des valeurs,
- Rechercher une valeur, etc.

En programmation fonctionnelle pure, la solution est d'utiliser la **récursion** qui consiste à **appliquer une fonction dans sa propre définition** (application récursive).

Définition d'une fonction récursive (I/II)

Comme nous venons de le dire, une fonction récursive est une fonction qui s'applique elle-même dans sa définition. Mais comme dans le cas d'une boucle, il faut s'assurer que la récursion se termine. Prenons par exemple la fonction suivante :

```
sumOfNFirstInt :: Int -> Int
sumOfNFirstInt n = n + sumOfNFirstInt (n - 1) -- application récursive
```

Essayons de réduire l'expression `sumOfNFirstInt 3` :

```
sumOfNFirstInt 3
-> 3 + sumOfNFirstInt (3 - 1)
-> 3 + 2 + sumOfNFirstInt (2 - 1)
-> 3 + 2 + 1 + sumOfNFirstInt (1 - 1)
-> 3 + 2 + 1 + 0 + sumOfNFirstInt (0 - 1)
-> 3 + 2 + 1 + 0 + (-1) + sumOfNFirstInt (-1 - 1)
-> ...
```

Cela ne s'arrête jamais. Pour éviter une **réduction infinie**, la définition doit avoir **au moins un cas de base** dont la définition ne contient **pas d'application récursive**.

Définition d'une fonction récursive (II/II)

Reprenons notre fonction. La valeur de `sumOfNFirstInt 0` n'a pas besoin d'être calculée : **c'est le cas de base**. Dans la définition de la fonction, le pattern matching permet d'exprimer cela facilement :

```
sumOfNFirstInt :: Int -> Int
sumOfNFirstInt 0 = 0 -- cas de base lorsque l'argument est 0
sumOfNFirstInt n = n + sumOfNFirstInt (n - 1) -- application récursive dans les autres cas.
```

Cette fois la réduction de `sumOfNFirstInt 3` se termine :

```
sumOfNFirstInt 3
-> 3 + sumOfNFirstInt (3 - 1)
-> 3 + 2 + sumOfNFirstInt (2 - 1)
-> 3 + 2 + 1 + sumOfNFirstInt (1 - 1)
-> 3 + 2 + 1 + 0 -- cas de base
-> 5 + 1 + 0
-> 6 + 0
-> 6
```

Mais que se passe-t-il si l'on essaie de calculer `sumOfNFirstInt 100000` dans le REPL ?

Dépassement de capacité de la pile (stack overflow)

Si l'on essaie de calculer `sumOfNFirstInt 100000` dans le REPL de spago, on obtient une erreur : stack overflow (dépassement de capacité de la pile).

Bien que la réduction d'une expression ne devrait pas poser de problème en principe, lors de l'**exécution**, le programme fonctionnel est **traduit en un programme impératif équivalent**. Les applications de fonctions deviennent alors des **appels de sous-programmes**, ce qui implique l'empilement de **stack frames**.

Pour `sumOfNFirstInt 100000`, cela signifie 100000 appels imbriqués de la fonction et donc 100000 stack frames dans la pile d'appels, ce qui dépasse la capacité de la pile : c'est le fameux **stack overflow**.

En programmation fonctionnelle, la récursion est nécessaire pour réaliser des boucles. Il est donc nécessaire de contourner cette limite d'une manière ou d'une autre. Une solution est la récursion terminale et l'**optimisation de la récursion terminale** (TCO) par le compilateur.

Récursion terminale (I/II)

Lorsqu'on réduit `sumOfNFirstInt 3`, on part du dernier entier (3) et on redescend jusqu'à 0. Les additions restent en suspens jusqu'à ce que l'on atteigne le cas de base.

Une autre approche consiste à partir du premier entier (0) et à remonter jusqu'au dernier. C'est ce qu'on fait avec une boucle `for` et un accumulateur en programmation impérative.

En programmation fonctionnelle pure, on peut réaliser cela à l'aide d'une fonction auxiliaire :

```
sumOfNFirstInt :: Int -> Int
sumOfNFirstInt n = aux n 0
    where
        aux 0 acc = acc -- cas de base
        aux i acc = aux (i - 1) (acc + i) -- cas récursif
```

La fonction auxiliaire est définie dans la **clause where** qui fonctionne comme une expression de liaison **let..in**. La liaison `aux` est locale à la définition de `sumOfNFirstInt`.

Récursion terminale (II/II)

Essayons de réduire à nouveau notre expression `sumOfNFirstInt 3` :

```
sumOfNFirstInt 3
-> aux 3 0
-> aux (3 - 1) (0 + 3)
-> aux (2 - 1) (3 + 2)
-> aux (1 - 1) (5 + 1)
-> 6 -- cas de base (la valeur de acc)
```

Dans cette réduction, on voit comment l'argument n diminue à chaque application, alors que acc augmente dans le même temps. On a l'équivalent d'une variable de boucle et d'un accumulateur mais sans mutation.

La valeur de la dernière application récursive est le résultat attendu : c'est une **récursion terminale (*tail-recursion*)**. On a une récursion terminale lorsque la valeur de l'application récursive n'est pas utilisée dans un calcul.

Optimisation de la récursion terminale

La **récursion terminale** (*tail-recursion*) permet une optimisation lors de la compilation : puisque la valeur de l'application récursive n'est pas utilisée dans un calcul, il n'est **pas nécessaire de conserver le contexte** de l'appelant.

Le compilateur peut alors **réutiliser le contexte courant** (pas d'empilement, pas d'overflow) ; la récursion devient une **boucle** : c'est la **TCO** (*Tail Call Optimization*).

Mais la TCO n'est **pas automatique**. Coder avec une récursion terminale n'est que le premier pas. Il faut encore un compilateur capable de :

- Déterminer qu'il s'agit d'une récursion terminale.
- Réaliser l'optimisation.

Si les compilateurs des **langages fonctionnel** implémentent souvent la TCO (ou quelque chose d'approchant), les compilateurs des langages impératifs (Java, C#, etc.) le font rarement (la récursion n'est pas essentielle).

Plan

Type de donnée algébrique (ADT)

Un type de données algébrique (ADT *algebraic data type*) est un type composite construit à partir de deux sortes de types :

- **Type produit (enregistrement, n-uplet)** dont l'ensemble de valeurs est le produit cartésien des ensembles de valeurs de ses parties.
- **Type somme (union disjointe)** composé de plusieurs parties munies chacune d'un constructeur, et dont l'ensemble de valeurs est celui de la partie associée au constructeur utilisé.

En Java, un type produit est défini par une classe, mais il n'y a pas de construction syntaxique spécifique pour les types somme.

On peut toutefois s'en approcher à l'aide d'une classe et des *static factory methods* pour simuler les différents constructeurs (p. ex. `Optional<T>` avec `.of()` et `.empty()`) ou, depuis la version 21, en combinant *sealed interface*, des *record* et le *pattern matching* de l'expression *switch*.

Type produit (record)

Un **type produit** est un type composite dont les valeurs sont des n-uplets de valeurs (des **enregistrements**). Le mot produit fait référence au fait que l'ensemble de valeurs du type est le produit cartésien des ensembles de valeurs des types de ses parties.

En PureScript, pour définir un enregistrement, on liste le nom et le type de ses parties entre accolades. Le mot-clé `type` permet de définir un synonyme pour un type enregistrement.

```
type Point = { x :: Number, y :: Number }

origin :: Point
origin = { x: 0.0, y: 0.0 }

distance :: Point -> Point -> Number
distance p1 p2 =
    let dx = p1.x - p2.x
        dy = p1.y - p2.y
    in Math.sqrt (dx * dx + dy * dy)
```

Type somme (union)

Un **type somme** est un type composite qui représente l'**union disjointe** (la somme) de plusieurs ensembles de valeurs. Différents **constructeurs** permettent de construire les valeurs de chacun de ces ensembles. Chaque constructeur définit une variante du type.

En PureScript, un type somme est défini avec le mot-clé `data`.

```
data Maybe a
    = Just a
    | Nothing

data Shape
    = Circle Number -- un cercle avec son rayon
    | Rectangle Number Number -- un rectangle avec largeur et hauteur
```

Dans le type `Maybe a`, le `a` est un paramètre de type comme le `T` dans `Optional<T>` en Java. Le polymorphisme paramétrique est la principale forme de polymorphisme dans les langages de la famille ML.

Pattern matching

Avec un type somme, il est souvent nécessaire de déterminer l'ensemble auquel appartient une valeur. Pour cela, on utilise généralement le **pattern matching** avec l'expression spéciale case-of ou la définition de fonction. En outre, cela permet de **déstructurer** (déconstruire) la valeur pour en extraire les différentes parties.

```
area1 :: Shape -> Number
area1 shape = case shape of
    Circle r -> pi * r * r
    Rectangle w h  -> w * h

area2 :: Shape -> Number
area2 (Circle r) = pi * r * r
area2 (Rectangle w h) = w * h
```

Le compilateur exige que les patterns (les motifs) du case-of ou de la définition soient **exhaustifs** (**qu'ils couvrent tous les cas**). On retrouve un mécanisme similaire et cette même contrainte dans l'expression `switch` avec une *sealed interface* de Java.

Plan

Définition du type List

En programmation fonctionnelle, une liste est généralement définie comme une liste chaînée (*linked list*) à l'aide d'un **type somme récursif**.

Ce type est composé d'un constructeur de liste vide et d'un constructeur qui produit une nouvelle liste en combinant un élément de tête et une liste existante. Par tradition, ce constructeur est appelé `Cons`.

En PureScript, le type liste est défini de la manière suivante :

```
data List a
    = Nil -- la liste vide
    | Cons a (List a) -- un élément suivi d'une liste
```

Là encore, le `a` est un paramètre de type comme le `T` de `List<T>` en Java. La liste est dite chaînée, car chaque nœud contient une valeur et une référence au reste de la liste.

Construction d'une liste

Pour construire une liste, on commence par associer un premier élément à une liste vide à l'aide du constructeur `Cons` et du constructeur de liste vide, `Nil` :

```
Cons 3 Nil
```

On utilise ensuite cette liste pour y associer successivement les différents éléments de notre liste avec le même constructeur :

```
Cons 2 (Cons 3 Nil)
Cons 1 (Cons 2 (Cons 3 Nil))
```

En PureScript, l'opérateur `:` est une version infixe et associative à droite de `Cons`. On peut donc construire la même liste de la manière suivante :

```
1 : 2 : 3 : Nil
```

Liste et fonction récursive (I/II)

En programmation fonctionnelle, un traitement sur les éléments d'une liste est naturellement réalisé au moyen d'une fonction récursive. Par exemple, les fonctions suivantes produisent une nouvelle liste en effectuant une opération (incrémenter et doubler) sur chacun des éléments d'une liste.

```
incElements :: List Int -> List Int
incElements Nil = Nil -- cas de base
incElements (Cons x xs) = Cons (x + 1) (incElements xs) -- cas récursif

doubleElements :: List Int ->  List Int
doubleElements Nil = Nil -- cas de base
doubleElements (Cons x xs) = Cons (x * 2) (doubleElements xs) -- cas récursif
```

Le pattern `Cons x xs` permet de déstructurer la liste : `x` est la tête (*head*) et `xs` la queue (*tail*). La tête est un élément, la queue est une liste.

Liste et fonction récursive (II/II)

On peut également réaliser des fonctions récursives qui **agrègent**, d'une manière ou d'une autre, les éléments d'une liste. Par exemple, les fonctions suivantes effectuent, respectivement, le comptage de ces éléments et la somme de leurs valeurs :

```haskell
countElements :: List Int -> Int
countElements Nil = 0 -- cas de base
countElements (Cons _ xs) = 1 + countElements xs -- cas récursif

sumElements :: List Int -> Int
sumElements Nil = 0 -- cas de base
sumElements (Cons x xs) = x + sumElements xs -- cas récursif
```

Dans la première fonction, le pattern `Cons _ xs` déstructure la liste, mais ne lie que la queue. Le caractère `_` (*underscore*) indique que la tête n'est pas utilisée.

Fonction d'ordre supérieur (I/III)

Si l'on regarde les fonctions `incElements` et `doubleElements`, on s'aperçoit que leur structure est identique et que seule l'opération diffère.

Puisqu'en programmation fonctionnelle, les fonctions sont des entités de première classe, on pourrait définir la fonction suivante :

```
applyToElements :: (Int -> Int) -> List Int -> List Int
applyToElements _ Nil = Nil -- cas de base
applyToElements f (Cons x xs) = Cons (f x) (applyToElements f xs) -- cas récursif
```

Le premier paramètre est une fonction de type `Int -> Int`. Par exemple :

```
inc :: Int -> Int
inc x = x + 1

double :: Int -> Int
double x = x * 2
```

Fonction d'ordre supérieur (II/III)

On peut ensuite utiliser l'application partielle pour reproduire nos deux fonctions :

```
incElements :: List Int -> List Int
incElements = applyToElements inc

doubleElements :: List Int ->  List Int
doubleElements = applyToElements double
```

La fonction `applyToElements` est une **fonction d'ordre supérieur (*higher order function*)**, c'est-à-dire une fonction qui s'applique à une fonction ou qui produit une fonction.

Appliquons la même idée aux fonctions `countElements` et `sumElements` :

```
aggregateElements :: (Int -> Int -> Int) -> Int -> List Int -> Int
aggregateElements _ acc Nil = acc -- cas de base
aggregateElements f acc (Cons x xs) =
    aggregateElements f (f acc x) xs -- cas récursif
```

Fonction d'ordre supérieur (III/III)

Les paramètres de `aggregateElements` sont une fonction d'agrégation qui prend la valeur de l'accumulateur et la valeur de l'élément, la valeur initiale de l'accumulateur et une liste.

Si on définit les fonctions d'agrégation suivantes :

```
count :: Int -> Int -> Int
count acc _ = acc + 1

sum :: Int -> Int -> Int
sum acc x = acc + x
```

On peut redéfinir nos fonctions `countElements` et `sumElements` :

```
countElements :: List Int -> Int
countElements = aggregateElements count 0

sumElements :: List Int -> Int
sumElements = aggregateElements sum 0
```

Fonction d'ordre supérieur et expression lambda

Les fonctions `inc`, `double`, `sum` et `count` sont très simples et leur définition est inutilement verbeuse. Lorsque c'est le cas, on peut utiliser des expressions lambda.

La définition de nos fonctions devient :

```
incElements :: List Int -> List Int
incElements = applyToElements (\ x -> x + 1)

doubleElements :: List Int ->  List Int
doubleElements = applyToElements  (\ x -> x * 2)

countElements :: List Int -> Int
countElements = aggregateElements (\ acc -> \ _ -> acc + 1) 0

sumElements :: List Int -> Int
sumElements = aggregateElements (\ acc -> \ x -> acc + x) 0
```

Les fonctions `applyToElements` et `aggregateElements` sont des implémentations naïves et spécialisées de deux fonctions essentielles : **map** et **fold**.

Fonction map

La fonction `map` appliquée à une liste, **transforme chaque élément** et produit une nouvelle **liste de même forme** (même taille, même ordre) avec les éléments transformés.

Dans la signature ci-dessous, le type de la **fonction de transformation** (`a -> b`) indique que le type de l'élément transformé (`b`) peut être de n'importe quel type (`forall b.` sans contraintes) et que ce type est indépendant de celui de l'élément original (`a`).

```
map :: forall a b. (a -> b) -> List a -> List b
```

Pour illustrer cela, définissons la liste d'entiers suivante :

```
myList = 1 : 2 : 3 : 4 : 5 : 6 : Nil
```

Puis appliquons-lui, par exemple, les transformations suivantes :

```
map (\x -> x * 2) myList -- (2 : 4 : 6 : 8 : 10 : 12 : Nil) -- liste d'entiers

map show myList -- ("1" : "2" : "3" : "4" : "5" : "6" : Nil) -- liste de chaînes
```

Fonction fold (I/III)

En programmation fonctionnelle, la fonction **fold** (replier, incorporer) est une fonction d'ordre supérieur qui permet de combiner les valeurs d'une structure pour les **agréger en une valeur unique**, possiblement une nouvelle structure.

Cette fonction a généralement les paramètres suivants :

- Une fonction d'agrégation (qui permet de combiner les éléments) dont les paramètres sont la valeur d'un élément et un **accumulateur**.
- La valeur initiale de l'accumulateur
- La structure à folder (typiquement une liste)

Le fold est réalisé en appliquant la fonction d'agrégation de manière systématique à la valeur de chaque élément de la structure et à la valeur de l'accumulateur. Lorsque le fold est terminé, le résultat est la valeur de l'accumulateur.

La fonction fold est une **primitive fondamentale** qui permet de réaliser la plupart des opérations sur les structures de données (somme, filtrage, transformation, etc.).

Fonction fold (II/III)

Il existe deux versions de la fonction fold : **`foldl`** et **`foldr`** (pour ***fold left*** et ***fold right***). Gauche et droite indiquent la manière dont le fold associe les valeurs des éléments (et la valeur initiale de l'accumulateur) lors de l'application de la fonction d'agrégation.

Pour illustrer cela, prenons la liste suivante :

```
1 : 2 : 3 : 4 : 5 : 6 : Nil
```

Avec l'opération + et la valeur initiale 0, l'association des éléments se fait de la manière suivante :

```
((((((0 + 1) + 2) + 3) + 4) + 5) + 6) -- fold left : association à gauche
(1 + (2 + (3 + (4 + (5 + (6 + 0)))))) -- fold right : association à droite
```

Remarque : En principe, les deux versions ont la même puissance. En pratique, le choix dépend de considérations de performance, de considérations techniques (évaluation stricte ou paresseuse) ou encore du type d'opération à réaliser et de la forme du résultat. Tout cela nous mènerait hors du cadre de ce module.

Fonction fold (III/III)

En PureScript, les fonctions **`foldl`** et **`foldr`** sont définies de la manière suivante :

```
foldl :: forall f a b. Foldable f => (b -> a -> b) -> b -> f a -> b
```

```
foldr :: forall f a b. Foldable f => (a -> b -> b) -> b -> f a -> b
```

On observe que la signature de la fonction d'agrégation n'est pas la même dans les deux fonctions. Dans ces signatures, le type b est le type de l'accumulateur ; sa position reflète la manière dont fold fait l'association :

- **à gauche** de la valeur de l'élément pour **`foldl`** : `(\ acc x -> ...)`
- **à droite** de la valeur de l'élément pour **`foldr`** : `(\ x acc -> ...)`

Attention : Gauche et droite indiquent bien la manière dont le fold associe les éléments, pas la manière dont il visite les éléments de la structure. Pour une liste, les éléments sont toujours visités de la tête vers la queue ; dans le cas d'une liste chaînée, c'est d'ailleurs le seul moyen.

Fold spécialisé — filter

La fonction fold est très versatile. Toutefois, pour les opérations courantes, on trouve généralement des fonctions spécialisées, potentiellement plus efficaces.

La fonction **`filter`** est une fonction d'ordre supérieur qui permet de sélectionner les éléments d'une liste satisfaisant un **prédicat** (une fonction `a -> Boolean`).

```
filter :: forall a. (a -> Boolean) -> List a -> List a
```

Par exemple, si on définit le prédicat suivant :

```
isEven :: Int -> Boolean
isEven n = n `mod` 2 == 0
```

On peut l'utiliser pour ne garder que les nombres pairs d'une liste :

```
filter isEven (1 : 2 : 3 : 4 : 5 : 6 : Nil) --  (2 : 4 : 6 : Nil)
```

Remarque : Dans la fonction `isEven`, les backtick autour de la fonction `mod` permettent d'utiliser cette fonction binaire (à deux paramètres) comme un opérateur infixe.

Fold spécialisé — autres exemples

La fonction **`length`** calcule la longueur d'une liste.

```
length :: forall a b f. Foldable f => Semiring b => f a -> b
```

La fonction **`sum`** calcule la somme des valeurs des éléments d'une liste.

```
sum :: forall f a. Foldable f => Semiring a => f a -> a
```

La fonction **`find`** recherche le **premier élément** satisfaisant un prédicat. Le type `Maybe a` exprime le fait que la recherche peut être infructueuse.

```
find :: forall f a. Foldable f => (a -> Boolean) -> f a -> Maybe a
```

La fonction **`any`** détermine si **au moins un** élément de la liste satisfait un prédicat.

```
any :: forall f b a. Foldable f => HeytingAlgebra b => (a -> b) -> f a -> b
```

Remarque : Dans la signature de **`any`**, la contrainte `HeytingAlgebra b` indique que le type **`b`** doit supporter, entre autres, les opérateurs **`&&`** et **`||`**. Le type Boolean remplit cette condition.

Pipeline (map-filter-reduce)

L'**opérateur d'application inversée** (`#` en PureScript) permet de former un **pipeline d'opérations**, comme le tube (`|`) en Bash ou en PowerShell.

Par exemple, dans l'expression suivante, `myList` est passée à la fonction `filter isEven` (application partielle), le résultat de `filter` est passé à la fonction `map (\ x -> x * 2)`, et le résultat de `map` à la fonction `sum`.

```
myList # filter isEven # map (\ x -> x * 2) # sum -- 24
```

Cette manière d'écrire est généralement plus intuitive et plus facile à lire que l'application classique avec ou sans l'**opérateur d'application** (`$`) qui permet d'éviter la prolifération des parenthèses :

```
sum (map (\x -> x * 2) (filter isEven myList)) -- application classique

sum $ map (\x -> x * 2) $ filter isEven myList -- application avec l'opérateur $
```

Plan

Retour sur la fonction map

Nous avons vu que la fonction **map** permet de transformer les éléments d'une liste en préservant la taille et l'ordre des éléments de la liste originale. Mais cette fonction peut s'appliquer à n'importe quelle structure qui « contient » une valeur (ou une multiplicité de valeurs dans le cas d'une collection).

Plus précisément, la fonction **map** s'applique à n'importe quel type qui fournit un **contexte** (le contexte est rarement un contenant) pour une valeur. Par exemple :

- `List a`, `Seq a`, `Array a` : le contexte est la collection elle-même, la transformation s'applique aux éléments.
- `Maybe a` : le contexte est la possibilité de l'absence de valeur, la transformation s'applique à `a`, si `Just a`, sinon `Nothing` est conservé.
- `Either a b` : le contexte est la possibilité d'une erreur, la transformation s'applique à `b`, si `Right b`, sinon `Left a` est conservé.

Foncteur — Un type qui supporte map

La signature de la fonction **map** est la suivante :

```
map :: forall f a b. Functor f => (a -> b) -> f a -> f b
```

Cette signature dit que **map** applique la fonction de transformation de type `a -> b` à la partie de type `a` d'une valeur de type `f a` (p. ex. `Maybe a`), et produit une valeur de type `f b` en conservant la structure du contexte `f`.

Il est important de noter que le contexte original n'est pas modifié, il est reconstruit avec les valeurs transformées. Par exemple :

- Si `f a` est une `List a`, `f b` est une nouvelle `List b`.
- Si `f a` est `Maybe a`, `f b` est un nouveau `Maybe b`.

La contrainte dit qu'il doit y avoir une **instance de classe de type** `Functor` pour le type `f`.

Dans le cadre de ce module, un **foncteur** est un type paramétré qui **fournit un contexte** (p. ex. `Maybe a`) et qui **supporte l'opérateur map**.

Foncteur — Maybe

Nous avons vu l'intérêt de **map** pour effectuer une opération sur les éléments d'une liste. Voyons maintenant comment cette fonction peut s'utiliser avec `Maybe a` et `Either a`.

Commençons par `Maybe a`. Si j'applique `map (\ x -> x * 2)` à `Just 5`, la fonction applique la transformation `(\ x -> x * 2)` à la valeur `5` et produit un nouveau contexte pour le résultat : `Just 10`.

```
map (\ x -> x * 2) (Just 5) -- (Just 10)

(\ x -> x * 2) <$> Just 5 -- (Just 10), même résultat avec l'opérateur infixe <$>
```

Si j'applique la même fonction à `Nothing`, j'obtiens `Nothing` :

```
map (\ x -> x * 2) Nothing -- (Nothing)

(\ x -> x * 2) <$>  Nothing -- (Nothing)
```

Foncteur — Either

Le type `Either a b` représente le plus souvent un calcul qui peut échouer. Le constructeur `Right b` est utilisé lorsque le calcul est effectué avec succès, le constructeur `Left a`, dans le cas contraire.

Si j'applique `map (\ x -> x * 2)` à `Right 5`, la fonction applique la transformation `(\ x -> x * 2)` à la valeur `5` et produit un nouveau contexte pour le résultat : `Right 10`.

```
map (\ x -> x * 2) (Right 5 :: Either String Int) -- (Right 10)
(\ x -> x * 2) <$> (Right 5 :: Either String Int) -- (Right 10)
```

La fonction `map` ne transforme que la partie droite. Si je l'applique à `Left "Echec!"`, j'obtiens `Left "Echec!"` :

```
map (\ x -> x * 2) (Left "Echec!" :: Either String Int) -- (Left "Echec!")
(\ x -> x * 2) <$> (Left "Echec!" :: Either String Int) -- (Left "Echec!")
```

Force et limite de la fonction map (I/II)

Comme on l'a vu, la fonction `map` permet d'appliquer une **transformation** à une valeur **indépendamment de son contexte**. La fonction de transformation ne sait rien du contexte et peut être appliquée aussi bien aux éléments d'une structure de type `List a`, `Seq a` ou `Array a`, qu'à une valeur de type `Maybe a` ou `Either a b`.

Mais que se passe-t-il si la fonction de transformation utilise une fonction comme `safeDivide`, ci-dessous, qui produit une valeur de type `Maybe a` ?

```
safeDivide :: Int -> Int -> Maybe Int
safeDivide _ 0 = Nothing
safeDivide a b = Just (a / b)
```

Par exemple :

```
map (\ x -> safeDivide 10 x) (Just 5) -- ???
```

Force et limite de la fonction map (II/II)

L'appliquation de `safeDivide 10` (application partielle) à la valeur `5` de `Just 5`, peut s'écire :

```
Just (safeDivide 10 5)
```

La garantie de transparence référentielle nous permet de remplacer l'expression (safeDivide 10 5) par sa valeur. Cela nous donne :

```
Just (Just 2) -- :type -> Maybe (Maybe Int)
```

Essayons maintenant d'appliquer un autre transformation à ce résultat, par exemple :

```
map (\x -> x * 3) (Just (Just 2))
```

Que se passe-t-il ?

Plan

Monade — Un foncteur qui supporte bind et pure (I/III)

L'application de `map (\x -> x * 3)` à la valeur `(Just 2)` de `Just (Just 2)` provoque une erreur de compilation : on essaie d'appliquer une fonction de type `Int -> Int` à une valeur de type `Maybe Int`.

La fonction **`bind`** (PureScript et Haskell), ou **`flatMap`** (Java, JavaScript, C#, etc.), permet de résoudre ce problème en « aplatissant » le résultat (**`>>=`** est l'opérateur infixe **`bind`**) :

```
bind (Just 5) (\ x -> safeDivide 10 x) -- (Just 2)

Just 5 >>= \ x -> safeDivide 10 x -- (Just 2)
```

On peut également utiliser la fonction avec une liste. Par exemple :

```
bind (10 : 20 : 30 : Nil) (\ x -> map (\ y -> y + x) (1 : 2 : Nil))

10 : 20 : 30 : Nil >>= \ x -> (\ y -> y + x) <$> (1 : 2 : Nil)
```

Remarque : Utilisez le REPL pour évaluer ces expressions et essayer d'appliquer la même transformation à la même liste avec la fonction **`map`** pour observer la différence.

Monade — Un foncteur qui supporte bind et pure (II/III)

La signature de la fonction `bind` est la suivante :

```
bind :: forall m b a. Monad m => m a -> (a -> m b) -> m b
```

Cette signature est proche de celle de `map` avec deux différences notables. Voyons d'abord les similarités. La fonction `bind` applique une **fonction de transformation** à la partie de type `a` d'une valeur de type `m` et la fonction produit une valeur de type `m b` en conservant la structure du contexte `m`.

Voyons maintenant les différences. D'abord, l'ordre des paramètres : la fonction est en deuxième position. Ensuite, le type de la valeur produite par la fonction de transformation : ce n'est plus une valeur de type `b`, mais une valeur de type `m b`. Enfin la contrainte : m doit être une **monade**.

Pour l'instant, une **monade** est un **foncteur qui supporte l'opération** `bind`.

Monade — Un foncteur qui supporte bind et pure (III/III)

Avec la fonction **bind**, la fonction de transformation doit produire une valeur de type **m b**. C'est ce qui permet l'aplatissement et l'enchaînement :

```
Just 5 >>= \ x -> safeDivide 10 x >>= \ y -> Just (y * 3) -- (Just 6)
```

Cet enchaînement fonctionne, mais si l'on regarde la deuxième transformation, on peut se dire qu'il est un peu étrange d'utiliser le type **Maybe a** pour un calcul qui ne peut pas échouer.

Pour éviter cela, on utilise plutôt la fonction **pure** (ou **return**) qui est, avec **bind**, la deuxième opération que doit supporter une monade. Cette fonction permet d'encapsuler une valeur dans le contexte offert par la monade :

```
Just 5 >>= \ x -> safeDivide 10 x >>= \ x -> pure (x * 3) -- (Just 6)
```

```
Just 5 >>= \ x -> safeDivide 10 x >>= (pure <<< (\ x -> x * 3)) -- (Just 6)
```

Remarque : En PureScript, les opérateurs **<<<** et **>>>**permet de composer des fonctions : `(f <<< g) x = f (g x)` et `(f >>> g) x = g (f x)`.

Monade — Bloc do (I/II)

L'une des choses les plus importantes à retenir de la notion de monade est qu'une **monade** permet d'**enchaîner des calculs (computations) dans un contexte** avec la fonction `bind`.

Cet enchaînement a pour effet de séquentialiser la réduction des expressions puisque chaque calcul dépend du précédent. En PureScript (et en Haskell), le bloc `do` permet d'écrire un tel enchaînement comme une séquence. Par exemple, les deux expressions suivantes sont équivalentes :

```
resBind :: Maybe Int
resBind =
    pure 5 >>= \x ->
        safeDivide 10 x >>= \y ->
            let res = y * 3
            in pure res
```

```
resDo :: Maybe Int
resDo = do
    x <- pure 5
    y <- safeDivide 10 x
    let res = y * 3
    pure res
```

Il est important de noter que le **bloc `do`** est du **sucre syntaxique** pour l'enchaînement d'opérateurs bind (`>>=`), pas une séquence dans le sens de la programmation impérative.

Monade — Bloc do (II/II)

À ce stade, un bloc `do` peut contenir quatre sortes d'expressions :

- Des liaisons monadiques avec l'opérateur `<-` qui permettent d'extraire une valeur de son contexte.

```
x <- pure 5 -- puisque la monade est un Maybe a, x <- Just 5   -->  x vaut 5
y <- safeDivide 10 x -- y <- safeDivide 10 5  -->  y <- Just 2  -->  y vaut 2
```

- Des expressions de liaison `let` sans le `in` (c'est implicitement le reste du bloc `do`).

```
let res = y * 3
```

- Des expressions conditionnelles (if-then-else et case-of).
- Une expression monadique sans le `<-` à la toute fin du bloc (la valeur du bloc).

Remarque : Parfois, la valeur d'une liaison monadique peut être ignorée. On écrit alors `_ <- <expr>` ou `<expr>`. C'est le cas par exemple avec une monade Either, quand on utilise son mécanisme de court-circuit pour faire une validation, ou avec une monade Effect, pour une opération qui décrit un effet de bord (p. ex. `log "Hello world!"`).

Monade d'effet (I/II)

Nous avons vu qu'en **programmation fonctionnelle pure**, une **fonction doit être pure** :

- Ne pas avoir d'effet de bord.
- Ne pas dépendre d'un état mutable non local.

Mais nous avons également vu que pour interagir avec un·e utilisateur·rice, un programme doit décrire une séquence d'opérations d'entrée/sortie qui ont :

- un effet de bord (affichage, mutation d'un buffer d'entrée, etc.).
- une dépendance à des états mutables (état du périphérique, utilisateur·rice, etc.).

Ni la séquence ni les propriétés des opérations d'entrée/sortie ne sont a priori compatibles avec la programmation fonctionnelle pure.

Toutefois, en partant de l'hypothèse que PureScript et Haskell permettent de réaliser des programmes interactifs, on en conclut qu'il doit exister un moyen de gérer les interactions (et donc les opérations d'entrée/sortie) en programmation fonctionnelle pure.

Monade d'effet (II/II)

L'un de ces moyens est d'utiliser une monade d'effet, par exemple, la monade `Effect` en PureScript ou la `IO` en Haskell.

Comme nous l'avons vu, une monade permet d'enchaîner des opérations dans le contexte de la monade avec la fonction `bind`. Si la réduction d'une opération dépend de celle de la précédente, on crée une séquence.

Le problème de la séquence est réglé, mais qu'en est-il des effets de bord.

L'idée est qu'en PureScript ou en Haskell, la réduction de la fonction `main` ne produit pas d'effet, mais une valeur de type `Effect Unit` ou `IO Unit` qui décrit les opérations d'entrée/sortie à effectuer.

En d'autres termes, la réduction de l'expression de la fonction `main` produit une sorte de programme dont l'exécution par le runtime produit un effet. D'un point de vue conceptuel, le programme fonctionnel est donc bien sans effet de bord.