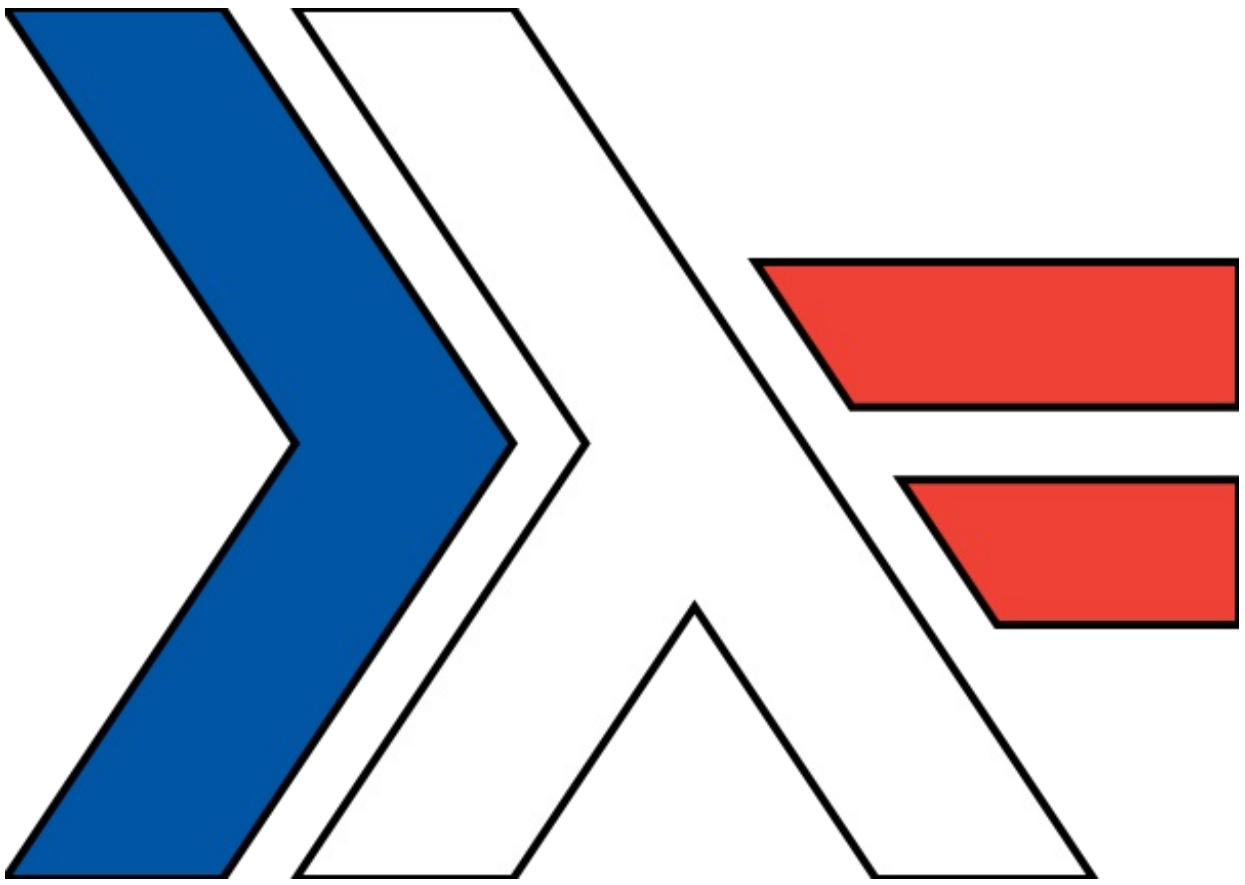
Apprendre Haskell vous fera le plus grand bien !



http://lyah.haskell.fr

http://haskell.fr

Table Of Contents

Introduction

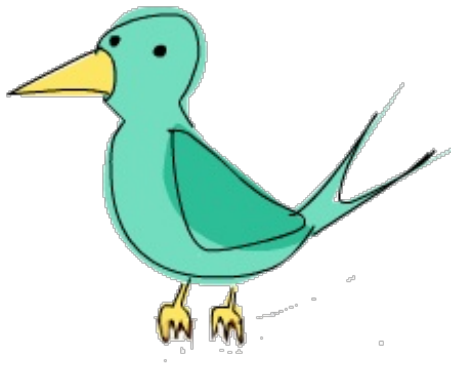
Table des matières

Démarrons ➔

À propos de ce tutoriel

Bienvenue dans **Apprendre Haskell vous fera le plus grand bien !** Si vous lisez ceci, il est possible que vous souhaitiez apprendre Haskell. Vous êtes au bon endroit, mais discutons un peu de ce tutoriel avant tout.

J'ai décidé d'écrire ce tutoriel parce que je voulais consolider ma propre connaissance d'Haskell, et parce que je pensais pouvoir aider les débutants à l'apprendre selon mon point de vue. Il y a plusieurs tutoriels traitant de Haskell qui traînent sur Internet. Quand je débutais en Haskell, je n'ai pas appris que d'une seule source. Ma méthode d'apprentissage consistait à lire plusieurs tutoriels et articles, car chacun d'eux expliquait d'une manière un peu différente de l'autre. En confrontant plusieurs ressources, j'ai pu assembler les pièces du puzzle et tout s'est mis à sa place. Ceci est une tentative visant à produire une ressource de plus afin d'apprendre Haskell, de sorte que vous ayez plus de chance d'en trouver une que vous appréciez.



Ce tutoriel vise les personnes ayant de l'expérience dans un langage de programmation impérative (C, C++, Java, Python, …) mais qui n'ont jamais programmé dans un langage de programmation fonctionnel auparavant (Haskell, ML, OCaml, …). Bien que je me doute que même sans vraie expérience en programmation, une chouette personne comme vous pourra suivre et apprendre Haskell.

Le canal #haskell sur le réseau freenode est un bon endroit pour poser des questions si vous êtes bloqué (NdT: Le canal #haskell-fr du même réseau vous permettra de poser vos questions en français). Les gens y sont extrêmement gentils, patients et compréhensifs envers les débutants.

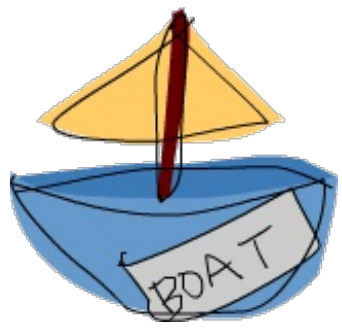
J'ai échoué dans mon apprentissage de Haskell à peu près deux fois avant d'arriver à le saisir, car cela me semblait trop bizarre et je ne comprenais pas. Mais alors, j'ai eu le déclic, et après avoir passé cet obstacle initial, le reste est venu plutôt facilement. Ce que j'essaie de vous dire : Haskell est génial, et si vous êtes intéressés par la programmation, vous devriez l'apprendre même si cela semble bizarre au début. Apprendre Haskell est très similaire à l'apprentissage de la programmation la première fois - c'est fun ! Ça vous force à penser différemment, ce qui nous amène à la section suivante…

Donc, qu'est-ce qu'Haskell ?

Haskell est un **langage de programmation fonctionnel pur**. Dans les langages impératifs, vous effectuez des choses en donnant à l'ordinateur une séquence de tâches, et il les exécute. Lors de cette exécution, il peut changer d'état. Par exemple, vous affectez à une variable `a` la valeur 5, puis vous faites quelque chose, puis lui affectez une autre valeur. Vous avez des structures de contrôle de flot pour répéter des actions plusieurs fois. Dans un langage de programmation fonctionnel pur, vous n'indiquez pas à l'ordinateur quoi faire, mais plutôt ce que les choses *sont*. La factorielle d'un nombre est le produit de tous les nombres de 1 à celui-ci, la somme d'une liste de nombres est égale au premier de ceux-ci additionné à la somme des autres, etc. Vous exprimez ceci sous la forme de fonctions. Également, vous n'affectez pas une valeur à une variable, puis une autre valeur plus tard. Si vous dites que `a` est égal à 5, vous ne pouvez pas dire qu'il est égal à autre chose plus tard, car vous venez de dire qu'il était égal à 5. Quoi, vous êtes un menteur ? Donc, dans les langages fonctionnels purs, une fonction n'a pas d'effets de bord. La seule chose qu'une fonction puisse faire, c'est calculer quelque chose et retourner le résultat. Au départ, cela semble plutôt limitant, mais il s'avère que ça a de très intéressantes conséquences : si une fonction est appelée deux fois, avec les mêmes paramètres, alors il est garanti qu'elle renverra le même résultat. On appelle ceci la transparence référentielle, et cela permet non seulement au compilateur de raisonner à propos du comportement du programme, mais également à vous de déduire facilement (et même de prouver) qu'une fonction est correcte, puis de construire des fonctions plus complexes en collant de simples fonctions l'une à l'autre.

Haskell est **paresseux**. Cela signifie qu'à moins de lui demander explicitement, Haskell n'exécutera pas les fonctions et ne calculera pas les choses tant qu'il n'est pas forcé de nous montrer les résultats. Cela va bien de pair avec la transparence référentielle et permet de penser ses programmes comme des **transformations sur des données**. Cela permet aussi des choses cool comme des structures de données infinies. Supposons que vous ayez une liste de nombres immuable `xs = [1, 2, 3, 4, 5, 6, 7, 8]` et une fonction `doubleMe` qui multiplie chaque élément par 2 et retourne la nouvelle liste. Si nous voulions multiplier notre liste par 8 dans un langage impératif en faisant `doubleMe(doubleMe(doubleMe(xs)))`, il traverserait probablement la liste une fois, ferait une copie, et la retournerait. Puis, il traverserait cette liste encore deux fois et retournerait le résultat. Dans un langage paresseux, appeler `doubleMe` sur une liste sans la forcer à s'afficher résulte en gros à ce que le programme vous dise "Ouais ouais, je le ferai plus tard !". Mais dès que vous voulez voir le résultat, le premier `doubleMe` dit au deuxième qu'il a besoin du résultat, maintenant ! Le deuxième le dit à son tour au troisième, et le troisième, avec réticence, renvoie un 1 doublé, qui est donc un 2. Le deuxième reçoit cela, et renvoie un 4 au premier. Le premier voit cela, et vous répond que le premier élément est un 8. Ainsi, la liste n'est traversée qu'une fois, et seulement quand vous en avez vraiment

besoin. De cette façon, lorsque vous voulez quelque chose d'un langage paresseux, vous pouvez juste prendre les données initiales, et efficacement les transformer de toutes les façons possibles jusqu'à arriver à vos fins.



Haskell est **typé statiquement**. Lorsque vous compilez votre programme, le compilateur sait quel bout de code est un nombre, lequel est une chaîne de caractères, etc. Cela veut dire que beaucoup d'erreurs potentielles sont détectées à la compilation. Si vous essayez de sommer un nombre et une chaîne de caractères, le compilateur vous criera dessus. Haskell utilise un système de types très bon qui bénéficie de l'**inférence de types**. Cela signifie que vous n'avez pas à marquer explicitement chaque bout de code avec un type, car le système de types peut intelligemment comprendre beaucoup de choses. Si vous dites `a = 5 + 4`, pas besoin d'indiquer à Haskell que `a` est un nombre, il peut s'en rendre compte tout seul. L'inférence de types permet également à votre code d'être plus général. Si une fonction que vous faites prend deux paramètres et les somme l'un à l'autre, et que vous ne précisez rien sur leur type, la fonction marchera sur n'importe quelle paire de paramètres qui se comportent comme des nombres.

Haskell est **élégant et concis**. Grâce à son usage de concepts de haut niveau, les programmes Haskell sont généralement plus courts que leurs équivalents impératifs. Et des programmes plus courts sont plus simples à maintenir et ont moins de bogues.

Haskell a été créé par des **personnes très intelligentes** (ayant un doctorat). Le travail sur Haskell a débuté en 1987 lorsqu'un comité de chercheurs s'est rassemblé pour concevoir un langage révolutionnaire. En 2003, le rapport Haskell fut publié, et définit une version stable du langage.

Ce dont vous avez besoin avant de plonger

Un éditeur de texte, et un compilateur Haskell. Vous avez probablement déjà un éditeur de texte favori, donc ne perdons pas de temps avec ça. Pour les besoins de ce tutoriel, nous utiliserons GHC, le compilateur Haskell le plus utilisé. Le meilleur moyen de démarrer est de télécharger la plate-forme Haskell, qui contient simplement un Haskell prêt à tourner.

GHC peut prendre un script Haskell (généralement avec l'extension .hs) et le compiler, mais il a aussi un mode interactif qui permet d'interagir interactivement avec les scripts. Interactivement. Vous pouvez appeler des fonctions depuis les scripts que vous chargez, et les résultats seront immédiatement affichés. Pour apprendre, c'est beaucoup plus simple et rapide que de compiler chaque fois que vous faites une modification, et ensuite relancer le programme depuis votre terminal. Le mode interactif est invoqué en tapant `ghci` dans votre terminal. Si vous avez défini des fonctions dans un fichier, par exemple `myfunctions.hs`, vous pouvez les charger en tapant `:l myfunctions`, puis jouer avec, à condition que `myfunctions.hs` soit dans le dossier d'où `ghci` a été invoqué. Si vous modifiez le script .hs, tapez juste `:l myfunctions` à nouveau, ou encore `:r`, qui est équivalent puisqu'il recharge le script courant. Ma procédure habituelle quand je joue avec des choses est de définir des fonctions dans un fichier *a.hs*, de les charger, et de bidouiller avec, puis de retourner changer le .hs, recharger, etc. C'est aussi ce qu'on va faire à présent.

Table des matières

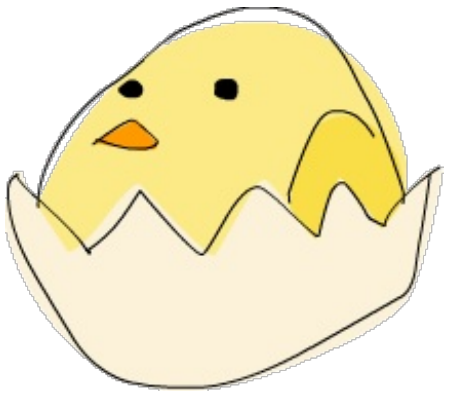
Démarrons ➔

Démarrons

← Introduction | Table des matières | Types et classes de types →

Prêts, feu, partez !

Bien, démarrons ! Si vous êtes le genre de personne horrible qui ne lit pas les introductions et que vous l'avez sautée, vous feriez peut-être bien de tout de même lire la dernière section de l'introduction, car elle explique ce dont vous avez besoin pour suivre ce tutoriel et comment l'on va charger des fonctions. La première chose qu'on va faire, c'est lancer un GHC interactif et appeler quelques fonctions pour se faire une première idée de Haskell. Ouvrez donc votre terminal et tapez `ghci`. Vous serez accueilli par un message semblable à celui-ci.



```
GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude>
```

Bravo, vous êtes dans GHCi ! L'invite `Prelude>` peut devenir de plus en plus long lorsqu'on importera des choses dans la session, alors on va le remplacer par `ghci>`. Si vous voulez le même invite, tapez simplement `:set prompt "ghci> "`.

Voici un peu d'arithmétique élémentaire.

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
ghci>
```

C'est plutôt simple. Nous pouvons également utiliser plusieurs opérateurs sur la même ligne, et les règles de précédence habituelles s'appliquent alors. On peut également utiliser des parenthèses pour rendre la précédence explicite, ou l'altérer.

```
ghci> (50 * 100) - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 * (100 - 4999)
-244950
```

Plutôt cool, non ? Ouais, je sais que c'est pas terrible, mais supportez-moi encore un peu. Attention, un écueil à éviter ici est la négation des nombres. Pour obtenir un nombre négatif, il est toujours mieux de l'entourer de parenthèses. Faire `5 * -3` vous causera les fureurs de GHCi, alors que `5 * (-3)` fonctionnera sans souci.

L'algèbre booléenne est également simple. Comme vous le savez sûrement, `&&` représente un *et* booléen, `||` un *ou* booléen, et `not` retourne la négation de `True` et `False` (NdT: respectivement vrai et faux).

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

Un test d'égalité s'écrit ainsi :

```
ghci> 5 == 5
```

```
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hello" == "hello"
True
```

Et que se passe-t-il si l'on fait `5 + "llama"` ou `5 == True` ? Eh bien, si on essaie le premier, on obtient un gros message d'erreur effrayant !

```
No instance for (Num [Char])
    arising from a use of `+' at <interactive>:1:0-9
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: 5 + "llama"
    In the definition of `it': it = 5 + "llama"
```

Beurk ! Ce que GHCi essaie de nous dire, c'est que `"llama"` n'est pas un nombre, et donc qu'il ne sait pas l'additionner à 5. Même si ce n'était pas `"llama"` mais `"four"` ou même `"4"`, Haskell ne considérerait pas cela comme un nombre. `+` attend comme opérandes à droite et à gauche des nombres. Si on essayait de faire `True == 5`, GHCi nous dirait que les types ne correspondent pas. Ainsi, `+` ne fonctionne qu'avec des paramètres pouvant être considérés comme des nombres, alors que `==` marche sur n'importe quelles deux choses à condition qu'on puisse les comparer. Le piège, c'est qu'elles doivent être toutes les deux du même type de choses. On ne compare pas des pommes et des oranges. Nous nous intéresserons de plus près aux types plus tard. Notez : vous pouvez tout de même faire `5 + 4.0` car `5` est vicieux et peut se faire passer pour un entier ou pour un nombre à virgule flottante. `4.0` ne peut pas se faire passer pour un entier, donc c'est à `5` de s'adapter à lui.

Vous ne le savez peut-être pas, mais nous venons d'utiliser des fonctions tout du long. Par exemple, `*` est une fonction qui prend deux nombres et les multiplie entre eux. Comme vous l'avez constaté, on l'appelle en le mettant en sandwich entre ces paramètres. C'est pour ça qu'on dit que c'est une fonction *infixe*. La plupart des fonctions qu'on n'utilise pas avec des nombres sont des fonctions *préfixes*. Intéressons-nous à celles-ci.

Les fonctions sont généralement préfixes, donc à partir de maintenant, nous ne préciserons pas qu'une fonction est préfixe, on le supposera par défaut. Dans la plupart des langages impératifs, les fonctions sont appelées en écrivant le nom de la fonction, puis ses paramètres entre parenthèses, généralement séparés par des virgules. En Haskell, les fonctions sont appelées en écrivant le nom de la fonction, puis un espace, puis ses paramètres, séparés par des espaces. Par exemple, essayons d'appeler une des fonctions les plus ennuyantes d'Haskell.

```
ghci> succ 8
9
```

La fonction `succ` prend n'importe quoi qui a un successeur, et renvoie ce successeur. Comme vous pouvez le voir, on sépare le nom de la fonction du paramètre par un espace. Appeler une fonction avec plusieurs paramètres est aussi simple. Les fonctions `min` et `max` prennent deux choses qu'on peut ordonner (comme des nombres !). `min` retourne la plus petite, `max` la plus grande. Voyez vous-mêmes :

```
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
```

L'application de fonction (appeler une fonction en mettant un espace après puis ses paramètres) a la plus grande des précédences. Cela signifie pour nous que les deux déclarations suivantes sont équivalentes.

```
ghci> succ 9 + max 5 4 + 1
16
ghci> (succ 9) + (max 5 4) + 1
16
```

Cependant, si nous voulions obtenir le successeur du produit des nombres 9 et 10, on ne pourrait pas écrire `succ 9 * 10`, car cela chercherait le successeur de 9, et le multiplierait par 10. Donc 100. Nous devrions écrire `succ (9 * 10)` pour obtenir 91.

Si une fonction prend deux paramètres, on peut aussi l'appeler de façon infixe en l'entourant d'apostrophes renversées. Par exemple, la fonction `div` prend deux entiers et effectue leur division entière. Faire `div 92 10` retourne 9. Mais quand on l'appelle de cette façon, on peut se demander quel nombre est divisé par quel nombre. On peut donc l'écrire plutôt `92 `div` 10`, ce qui est tout de suite plus clair.

Beaucoup de personnes venant de langages impératifs ont pour habitude de penser que les parenthèses indiquent l'application de fonctions. Par exemple, en C, on utilise des parenthèses pour appeler des fonctions comme `foo()`, `bar(1)` ou `baz(3, "haha")`. Comme nous l'avons vu, les espaces sont utilisés pour l'application de fonctions en Haskell. Donc, en Haskell, on écrirait `foo`, `bar 1` et `baz 3 "haha"`. Si vous voyez quelque chose comme `bar (bar 3)`, cela ne veut donc pas dire que `bar` est appelé avec les paramètres `bar` et `3`. Cela signifie qu'on appelle la fonction `bar` avec un paramètre `3` pour obtenir un nombre, et qu'on appelle `bar` à nouveau sur ce nombre. En C, cela serait `bar(bar(3))`.

Nos premières fonctions

Dans la section précédente, nous avons eu un premier aperçu de l'appel de fonctions. Essayons maintenant de créer les nôtres ! Ouvrez votre éditeur de texte favori et entrez cette fonction qui prend un nombre et le multiplie par deux.

```
doubleMe x = x + x
```

Les fonctions sont définies de la même façon qu'elles sont appelées. Le nom de la fonction est suivi de ses paramètres, séparés par des espaces. Mais lors de la définition d'une fonction, un `=` suivi de la définition de ce que la fonction fait suivent. Sauvez ceci en tant que `baby.hs` ou quoi que ce soit. À présent, naviguez jusqu'à l'endroit où vous l'avez sauvegardé, et lancez `ghci` d'ici. Une fois lancé, tapez `:l baby`. Maintenant que notre script est chargé, on peut jouer avec la fonction que l'on vient de définir.

```
ghci> :l baby
[1 of 1] Compiling Main             ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8.3
16.6
```

Puisque `+` fonctionne sur des entiers aussi bien que sur des nombres à virgule flottante (tout ce que l'on peut considérer comme un nombre en fait), notre fonction fonctionne également sur n'importe quel nombre. Créons une fonction prenant deux nombres et les multipliant chacun par deux, puis les sommant ensemble.

```
doubleUs x y = x*2 + y*2
```

Simple. Nous aurions également pu l'écrire `doubleUs x y = x + x + y + y`. Un test produit les résultats attendus (rappelez-vous bien d'ajouter cette fonction à la fin de `baby.hs`, de sauvegarder le fichier, et de faire `:l baby` dans GHCi).

```
ghci> doubleUs 4 9
26
ghci> doubleUs 2.3 34.2
73.0
ghci> doubleUs 28 88 + doubleMe 123
478
```

Comme attendu, vous pouvez appeler vos propres fonctions depuis les autres fonctions que vous aviez créées. Avec cela en tête, redéfinissons `doubleUs` ainsi :

```
doubleUs x y = doubleMe x + doubleMe y
```

Ceci est un exemple simple d'un motif récurrent en Haskell. Créer des fonctions basiques, qui sont visiblement correctes, puis les combiner pour faire des fonctions plus complexes. De cette manière, on évite la répétition. Que se passerait-il si un mathématicien se rendait compte que 2 est en fait 3 et qu'il fallait changer votre programme ? Vous pourriez simplement redéfinir `doubleMe` comme `x + x + x` et, puisque `doubleUs` appelle `doubleMe`, elle fonctionnerait automatiquement dans cet étrange nouveau monde où 2 est 3.

Les fonctions en Haskell n'ont pas à être dans un ordre particulier, donc il n'importe pas que vous définissiez `doubleMe` puis `doubleUs` ou l'inverse.

Maintenant, nous allons écrire une fonction qui multiplie un nombre par 2, mais seulement si ce nombre est inférieur ou égal à 100, parce que les nombres supérieurs à 100 sont déjà bien assez gros comme ça !

```
doubleSmallNumber x = if x > 100
                      then x
                      else x*2
```

Ici, nous avons introduit la construction if de Haskell. Vous êtes probablement habitué aux constructions if des autres langages. La différence entre le if de Haskell et celui des autres langages, c'est qu'en Haskell, le else est obligatoire. Dans les langages impératifs, vous pouvez sauter quelques étapes si la condition n'est pas satisfaite, mais en Haskell, chaque expression doit renvoyer quelque

chose. Nous aurions aussi pu écrire ce if en une ligne mais je trouve cette version plus lisible. Un autre point à noter est que la construction if en Haskell est une expression. Une expression correspond simplement à tout bout de code retournant une valeur. `5` est une expression car elle retourne 5, `4 + 8` est une expression, `x + y` est une expression car elle retourne la somme de `x` et `y`. Puisque le else est obligatoire, une construction if retournera toujours quelque chose, c'est pourquoi c'est une expression. Si nous voulions ajouter 1 à chaque nombre produit dans la fonction précédente, nous aurions pu l'écrire ainsi.

```
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
```

Si nous avions omis les parenthèses, nous aurions ajouté 1 seulement si `x` était plus petit que 100. Remarquez le `'` à la fin du nom de la fonction. Cette apostrophe n'a pas de signification spéciale en Haskell. C'est un caractère valide à utiliser dans un nom de fonction. On utilise habituellement `'` pour indiquer la version stricte d'une fonction (une version qui n'est pas paresseuse) ou pour la version légèrement modifiée d'une fonction ou d'une variable. Puisque `'` est un caractère valide dans le nom d'une fonction, on peut écrire :

```
conanO'Brien = "It's a-me, Conan O'Brien!"
```

Il y a deux choses à noter ici. La première, c'est que dans le nom de la fonction, nous n'avons pas mis de majuscule au prénom de Conan. C'est parce que les fonctions ne peuvent pas commencer par une majuscule. Nous verrons pourquoi un peu plus tard. La seconde chose, c'est que la fonction ne prend aucun paramètre. Lorsqu'une fonction ne prend pas de paramètre, on dit généralement que c'est une *définition* (ou un *nom*). Puisqu'on ne peut pas changer ce que les noms (et les fonctions) signifient une fois qu'on les a définis, `conanO'Brien` et la chaîne `"It's a-me, Conan O'Brien!"` peuvent être utilisés de manière interchangeable.

Introduction aux listes

Tout comme les listes de courses dans le monde réel, les listes Haskell sont très utiles. C'est la structure de donnée la plus utilisée, et elle peut l'être d'une multitude de façons pour modéliser et résoudre tout un tas de problèmes. Les listes sont TROP géniales. Dans cette section, nous allons découvrir les bases des listes, des chaînes de caractères (qui sont en fait des listes) et des listes en compréhension.

En Haskell, les listes sont des structures de données **homogènes**. Elles contiennent plusieurs éléments du même type. Cela signifie qu'on peut avoir une liste d'entiers, une liste de caractères, mais jamais une liste qui a à la fois des entiers et des caractères. Place à une liste !

Note : Nous utilisons le mot-clé `let` pour définir un nom directement dans GHCi. Écrire `let a = 1` dans GHCi est équivalent à écrire `a = 1` dans un script puis charger ce script.

```
ghci> let lostNumbers = [4,8,15,16,23,42]
ghci> lostNumbers
[4,8,15,16,23,42]
```

Comme vous pouvez le constater, les listes sont dénotées par des crochets, et les valeurs d'une liste sont séparées par des virgules. Si on essayait une liste comme `[1, 2, 'a', 3, 'b', 'c', 4]`, Haskell se plaindrait que des caractères (qui, d'ailleurs, sont dénotés comme un caractère entouré d'apostrophes) ne sont pas des nombres. En parlant de caractères, les chaînes de caractères sont simplement des listes de caractères. `"hello"` est simplement du sucre syntaxique pour `['h', 'e', 'l', 'l', 'o']`. Puisque les chaînes de caractères sont des listes, on peut utiliser les fonctions de listes sur celles-ci, ce qui s'avère très pratique.

Une tâche courante consiste à coller deux listes l'une à l'autre. Ceci est réalisé à l'aide de l'opérateur `++`.

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

Attention lors de l'utilisation répétée de l'opérateur `++` sur de longues chaînes. Lorsque vous accolez deux listes (et même si vous accolez une liste singleton à une autre liste, par exemple : `[1, 2, 3] ++ [4]`), en interne, Haskell doit parcourir la liste de gauche en entier. Ceci ne pose pas de problème tant que les listes restent de taille raisonnable. Mais accoler une liste à la fin d'une liste qui contient cinquante millions d'éléments risque de prendre du temps. Cependant, placer quelque chose au début d'une liste en utilisant l'opérateur `:` (aussi appelé l'opérateur cons) est instantané.

```
ghci> 'A':" SMALL CAT"
"A SMALL CAT"
```

```
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

Remarquez comme `:` prend un nombre et une liste de nombres, ou bien un caractère et une liste de caractères, alors que `++` prend deux listes. Même si vous voulez ajouter un seul élément à la fin d'une liste avec `++`, vous devez l'entourer de crochets pour en faire d'abord une liste.

`[1, 2, 3]` est en fait du sucre syntaxique pour `1:2:3:[]`. `[]` est la liste vide. Si nous ajoutons `3` devant elle, elle devient `[3]`. Si nous ajoutons encore `2` devant, elle devient `[2, 3]`, etc.

Note : `[]`, `[[]]` et `[[], [], []]` sont trois choses différentes. La première est une liste vide, la deuxième est une liste qui contient un élément, cet élément étant une liste vide, la troisième est une liste qui contient trois éléments, qui sont tous des listes vides.

Si vous voulez obtenir un élément d'une liste par son index, utilisez `!!`. Les indices démarrent à 0.

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

Mais si vous essayez d'obtenir le sixième élément d'une liste qui n'en a que quatre, vous obtiendrez une erreur, soyez donc vigilant !

Les listes peuvent aussi contenir des listes. Elles peuvent même contenir des listes qui contiennent des listes…

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

Plusieurs listes à l'intérieur d'une liste peuvent avoir des longueurs différentes, mais elles ne peuvent pas être de types différents. Tout comme l'on ne peut pas avoir une liste contenant quelques caractères et quelques nombres, on ne peut pas avoir de liste qui contient quelques listes de caractères et quelques listes de nombres.

Les listes peuvent être comparées si ce qu'elles contiennent peut être comparé. En utilisant `<`, `<=`, `>` et `>=` pour comparer des listes, celles-ci sont comparées par ordre lexicographique. D'abord, les têtes sont comparées. Si elles sont égales, alors les éléments en deuxième position sont comparés, etc.

```
ghci> [3,2,1] > [2,1,0]
True
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] > [3,4]
True
ghci> [3,4,2] > [2,4]
True
ghci> [3,4,2] == [3,4,2]
True
```

Que peut-on faire d'autre avec des listes ? Voici quelques fonctions de base qui opérent sur des listes.

`head` prend une liste et retourne sa tête. La tête est simplement le premier élément.

```
ghci> head [5,4,3,2,1]
5
```

`tail` prend une liste et retourne sa queue. En d'autres termes, elle coupe la tête de la liste.

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

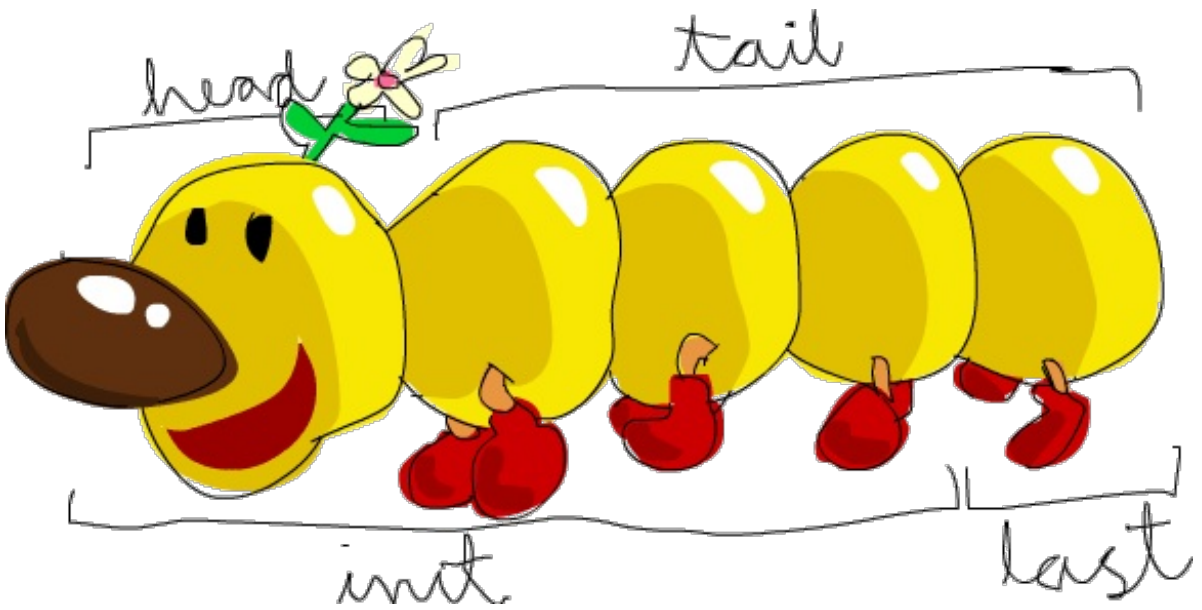
`last` prend une liste et retourne son dernier élément.

```
ghci> last [5,4,3,2,1]
1
```

`init` prend une liste et retourne tout sauf son dernier élément.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

En imaginant une liste comme un monstre, voilà ce que ça donne.



Mais que se passe-t-il si l'on essaie de prendre la tête d'une liste vide ?

```
ghci> head []
*** Exception: Prelude.head: empty list
```

Mon dieu ! Tout nous pète à la figure ! S'il n'y a pas de monstre, il ne peut pas avoir de tête. Lors de l'utilisation de **`head`**, **`tail`**, **`last`** et **`init`**, faites attention à ne pas les utiliser sur des listes vides. Cette erreur ne peut pas être détectée à la compilation, donc il est toujours de bonne pratique de prendre ses précautions pour ne pas demander à Haskell des éléments d'une liste vide.

`length` prend une liste et retourne sa longueur.

```
ghci> length [5,4,3,2,1]
5
```

`null` teste si une liste est vide. Si c'est le cas, elle retourne **`True`**, sinon **`False`**. Utilisez cette fonction plutôt que d'écrire **`xs == []`** (si votre liste s'appelle **`xs`**).

```
ghci> null [1,2,3]
False
ghci> null []
True
```

`reverse` renverse une liste.

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

`take` prend un nombre et une liste. Elle extrait ce nombre d'éléments du début de la liste. Regardez.

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

Voyez comme, si l'on essaie de prendre plus d'éléments que la liste n'en contient, elle retourne la liste entière. Si on essaie d'en prendre 0, on obtient une liste vide.

`drop` marche d'une manière similaire, mais elle jette le nombre d'éléments demandé du début de la liste.

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

`maximum` prend une liste de choses qui peuvent être ordonnées et retourne la plus grande d'entre elles.

`minimum` retourne la plus petite.

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

`sum` prend une liste de nombres et retourne leur somme.

`product` prend une liste de nombres et retourne leur produit.

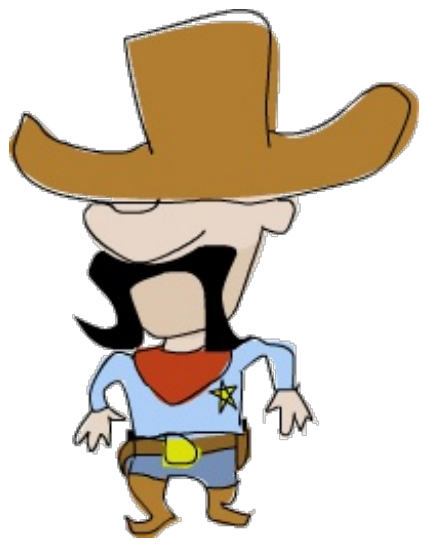
```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

`elem` prend une chose et une liste de choses, et nous indique si cette première apparaît dans la liste. On l'utilise généralement de manière infixe car c'est plus simple à lire.

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

C'était un premier tour des fonctions qui opérent sur des listes. Nous en verrons d'autres plus tard.

Texas rangées



Que faire si l'on veut la liste des nombres de 1 à 20 ? Bien sûr, on pourrait les taper un par un, mais ce n'est pas une solution pour des gentlemen qui demandent l'excellence de leurs langages de programmation. Nous utiliserons plutôt des progressions (NDT : qui sont appelées "ranges" en anglais, d'où le titre de cette section). Les progressions sont un moyen de créer des listes qui sont des suites arithmétiques d'éléments qui peuvent être énumérés. Les nombres peuvent être énumérés. Un, deux, trois, quatre, etc. Les caractères peuvent aussi être énumérés. L'alphabet est une énumération des caractères de A à Z. Les noms ne peuvent pas être énumérés. Qu'est-ce qui suit "John" ? Je ne sais pas.

Pour créer une liste contenant tous les entiers naturels de 1 à 20, on écrit **`[1..20]`**. C'est équivalent à **`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]`**, et il n'y a aucune différence si ce n'est qu'écrire la séquence manuellement est stupide.

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

Les progressions sont cool car on peut préciser un pas. Que faire si l'on cherche les nombres pairs entre 1 et 20 ? Ou un nombre sur trois ?

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

Il suffit juste de séparer les deux premiers éléments par une virgule, puis de spécifier la borne supérieure. Bien que plutôt intelligentes, les progressions ne sont pas aussi intelligentes que ce que certaines personnes attendent. Vous ne pouvez pas écrire **`[1, 2, 4, 8, 16..100]`** en espérant obtenir les puissances de 2.

Premièrement, parce qu'on ne peut spécifier qu'un pas. Secondement, parce que certaines progressions non arithmétiques sont ambigües lorsqu'on ne les énonce que par leurs premiers éléments.

Pour créer une liste des nombres de 20 à 1, vous ne pouvez pas écrire `[20..1]`, vous devez écrire `[20, 19..1]`.

Attention lors de l'utilisation de nombres à virgule flottante dans les progressions ! N'étant pas entièrement précis (par définition), leur utilisation peut donner des résultats originaux.

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

Mon avis est de ne pas les utiliser dans les progressions.

Vous pouvez aussi utiliser les progressions pour définir des listes infinies, simplement en ne précisant pas de borne supérieure. Nous verrons les détails des listes infinies un peu plus tard. Pour l'instant, examinons comment l'on pourrait obtenir les 24 premiers multiples de 13. Bien sûr, vous pourriez écrire `[13, 26..24*13]`. Mais il y a une meilleure façon de faire : `take 24 [13, 26..]`. Puisqu'Haskell est paresseux, il ne va pas essayer d'évaluer la liste infinie immédiatement et ne jamais terminer. Il va plutôt attendre de voir ce que vous voulez obtenir de cette liste infinie. Ici, il voit que vous ne voulez que les 24 premiers éléments, et il s'exécute poliment.

Une poignée de fonctions produit des listes infinies :

`cycle` prend une liste et la cycle en une liste infinie. Si vous essayiez d'afficher le résultat, cela continuerait à jamais, donc il faut la couper quelque part.

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

`repeat` prend un élément et produit une liste infinie contenant uniquement cet élément. Cela correspond à cycler une liste à un élément.

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

Cependant, il est plus simple d'utiliser la fonction `replicate` pour obtenir un certain nombre de fois le même élément dans une liste. `replicate 3 10` retourne `[10, 10, 10]`.

Je suis une liste en compréhension

Si vous avez déjà suivi une classe de mathématiques, vous avez probablement déjà rencontré des ensembles définis en compréhension. On les utilise généralement pour construire des ensembles plus spécifiques à partir d'autres ensembles plus généraux. Une compréhension simple d'un ensemble qui contient les dix premiers entiers naturels pairs est $S = \{ 2 \cdot x \mid x \in \mathbb{N},\ x \leq 10 \}$. La partie située devant la barre verticale est la fonction qui produit la sortie, `x` est la variable, `N` est l'ensemble en entrée et `x <= 10` est un prédicat. Cela signifie que l'ensemble contient le double de tous les entiers naturels qui satisfont le prédicat.

Si nous voulions écrire cela en Haskell, nous pourrions le faire ainsi `take 10 [2,4..]`. Mais que faire si l'on ne voulait pas les doubles des 10 premiers entiers naturels, mais quelque chose de plus compliqué ? On pourrait utiliser une liste en compréhension pour ça. Les listes en compréhension sont très semblables aux ensembles en compréhension. Restons-en au cas des 10 entiers pairs pour l'instant. La liste de compréhension adéquate serait `[x*2 | x <- [1..10]]`. `x` est extrait de `[1..10]` et pour chaque élément de `[1..10]` (désormais attaché à `x`), nous prenons son double. En action.

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

Comme vous pouvez le voir, nous obtenons le résultat attendu. Ajoutons à présent une condition (ou un prédicat) à cette compréhension. Les prédicats se placent après les liaisons, et sont séparés de ceux-ci par une virgule. Disons que l'on cherche les éléments qui, une fois doublés, sont plus grands ou égaux à 12.

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
```

Cool, ça marche. Et si nous voulions tous les nombres de 50 à 100 dont le reste de la division par 7 est 3 ? Facile.

```
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

Succès ! Notez que nettoyer une liste à l'aide de prédicats s'appelle aussi le **filtrage**. Nous avons pris une liste de nombres et l'avons filtrée en accord avec le prédicat. Un autre exemple. Disons qu'on veut une compréhension qui remplace chaque nombre impair plus grand que 10 par `"BANG!"` et chaque nombre impair plus petit que 10 par `"BOOM!"`. Si un nombre est pair, on le rejette de la liste. Pour plus de facilité, on va placer cette compréhension dans une fonction, pour pouvoir la réutiliser facilement.

```
boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
```

La dernière partie de la compréhension est le prédicat. La fonction `odd` renvoie `True` pour un nombre impair, et `False` pour un nombre pair. L'élément est inclus dans la liste seulement si tous les prédicats sont évalués à `True`.

```
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

On peut inclure plusieurs prédicats. Si nous voulions tous les nombres de 10 à 20 qui sont différents de 13, 15 et 19, on pourrait faire :

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

Non seulement on peut avoir plusieurs prédicats dans une liste en compréhension (un élément doit satisfaire tous les prédicats pour être inclus dans la liste résultante), mais on peut également piocher dans plusieurs listes. Lorsqu'on pioche dans plusieurs listes, les compréhensions produisent toutes les combinaisons des listes en entrée et joignent tout ça à l'aide de la fonction que l'on fournit. Une liste produite par une compréhension qui pioche dans deux listes de longueur 4 aura donc pour longueur 16, si tant est qu'on ne filtre pas d'élément. Si l'on a deux listes, `[2, 5, 10]` et `[8, 10, 11]`, et qu'on veut les produits possibles des combinaisons de deux nombres de chacune de ces listes, voilà ce qu'on écrit :

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

Comme prévu, la longueur de la nouvelle liste est 9. Et si l'on voulait seulement les produits supérieurs à 50 ?

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

Pourquoi pas une compréhension qui combine une liste d'adjectifs et une liste de noms… pour provoquer une hilarité épique.

```
ghci> let nouns = ["hobo","frog","pope"]
ghci> let adjectives = ["lazy","grouchy","scheming"]
ghci> [adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog",
"grouchy pope","scheming hobo","scheming frog","scheming pope"]
```

Je sais ! Écrivons notre propre version de `length` ! Appelons-la `length'`.

```
length' xs = sum [1 | _ <- xs]
```

`_` signifie que l'on se fiche de ce qu'on a pioché dans la liste, donc plutôt que d'y donner un nom qu'on n'utilisera pas, on écrit `_`. Cette fonction remplace chaque élément de la liste par un `1`, et somme cette liste. La somme résultante sera donc la longueur de la liste.

Un rappel amical : puisque les chaînes de caractères sont des listes, on peut utiliser les listes en compréhension pour traiter et produire des chaînes de caractères. Voici une fonction qui prend une chaîne de caractères et supprime tout sauf les caractères en majuscule.

```
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

Testons :

```
ghci> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
ghci> removeNonUppercase "IdontLIKEFROGS"
"ILIKEFROGS"
```

Le prédicat ici fait tout le travail. Il indique que le caractère est inclus dans la nouvelle liste uniquement s'il appartient à la liste `['A'..'Z']`. Il est possible

d'imbriquer les compréhensions si vous opérez sur des listes qui contiennent des listes. Soit une liste qui contient plusieurs listes de nombres. Supprimons tous les nombres impairs sans aplatir la liste.

```
ghci> let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]
ghci> [ [ x | x <- xs, even x ] | xs <- xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

Vous pouvez écrire les compréhensions de listes sur plusieurs lignes. Donc, en dehors de GHCi, il vaut mieux les découper sur plusieurs lignes, surtout si elles sont imbriquées.

Tuples

D'une façon, les tuples sont comme des listes - ils permettent de stocker plusieurs valeurs dans une seule. Cependant, il y a des différences fondamentales. Une liste de nombres est une liste de nombres. C'est son type, et cela n'importe pas qu'elle ait un seul nombre ou une infinité. Les tuples, par contre, sont utilisés lorsque vous savez exactement combien de valeurs vous désirez combiner, et son type dépend du nombre de composants qu'il a et du type de ces composantes. Ils sont dénotés avec des parenthèses, et les composantes sont séparées par des virgules.

Une autre différence clé est qu'ils n'ont pas à être homogènes. Contrairement à une liste, un tuple peut contenir une combinaison de différents types.

Demandez-vous comment on représenterait un vecteur bidimensionnel en Haskell. Une possibilité serait d'utiliser une liste. Ça marcherait en partie. Que faire si l'on voulait mettre quelques vecteurs dans une liste pour représenter les points d'une forme du plan ? On pourrait faire `[[1, 2], [8, 11], [4, 5]]`. Le problème de cette méthode est que l'on pourrait également faire `[[1, 2], [8, 11, 5], [4, 5]]`, qu'Haskell accepterait puisque cela reste une liste de listes de nombres, mais ça n'a pas vraiment de sens. Alors qu'un tuple de taille deux (aussi appelé une paire) est un type propre, ce qui signifie qu'une liste ne peut pas avoir quelques paires puis un triplet (tuple de taille trois), utilisons donc cela en lieu et place. Plutôt que d'entourer nos vecteurs de crochets, on utilise des parenthèses : `[(1, 2), (8, 11), (4, 5)]`. Et si l'on essayait d'entrer la forme `[(1, 2), (8, 11, 5), (4, 5)]` ? Eh bien, on aurait cette erreur :

```
Couldn't match expected type `(t, t1)'
against inferred type `(t2, t3, t4)'
In the expression: (8, 11, 5)
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
In the definition of `it': it = [(1, 2), (8, 11, 5), (4, 5)]
```

Cela nous dit que l'on a essayé d'utiliser une paire et un triplet dans la même liste, ce qui ne doit pas arriver. Vous ne pourriez pas non plus créer une liste `[(1, 2), ("One", 2)]` car le premier élément de cette liste est une paire de nombres alors que le second est une paire d'une chaîne de caractères et d'un nombre. Les tuples peuvent aussi être utilisés pour représenter un grand éventail de données. Par exemple, si nous voulions représenter le nom et l'âge d'une personne en Haskell, on pourrait utiliser le triplet : `("Christopher", "Walken", 55)`. Comme dans l'exemple, les tuples peuvent aussi contenir des listes.

Utilisez des tuples lorsque vous savez à l'avance combien de composantes une donnée doit avoir. Les tuples sont beaucoup plus rigides car chaque taille de tuple a son propre type, donc on ne peut pas écrire une fonction générique pour ajouter un élément à un tuple - il faudrait écrire une fonction pour ajouter à une paire, une fonction pour ajouter à un triplet, un fonction pour ajouter à un quadruplet, etc.

Bien qu'il y ait des listes singleton, il n'existe pas de tuple singleton. Ça n'a pas beaucoup de sens si vous y réfléchissez. Un tuple singleton serait seulement la valeur qu'il contient, et n'aurait donc pas d'intérêt.

Comme les listes, les tuples peuvent être comparés entre eux si leurs composantes peuvent être comparées. Seulement, vous ne pouvez pas comparer des tuples de tailles différentes, alors que vous pouvez comparer des listes de tailles différentes. Deux fonctions utiles qui opérent sur des paires :

`fst` prend une paire et renvoie sa première composante.

```
ghci> fst (8,11)
8
ghci> fst ("Wow", False)
"Wow"
```

`snd` prend une paire et renvoie sa seconde composante. Quelle surprise !

```
ghci> snd (8,11)
11
ghci> snd ("Wow", False)
False
```

Note : ces deux fonctions opèrent seulement sur des paires. Elles ne marcheront pas sur des triplets, des quadruplets, etc. Nous verrons comment extraire

des données de tuples de différentes façons un peu plus tard.

Une fonction cool qui produit une liste de paires : `zip`. Elle prend deux listes et les zippe ensemble en joignant les éléments correspondants en des paires. C'est une fonction très simple, mais elle sert beaucoup. C'est particulièrement utile pour combiner deux listes d'une façon ou traverser deux listes simultanément. Démonstration.

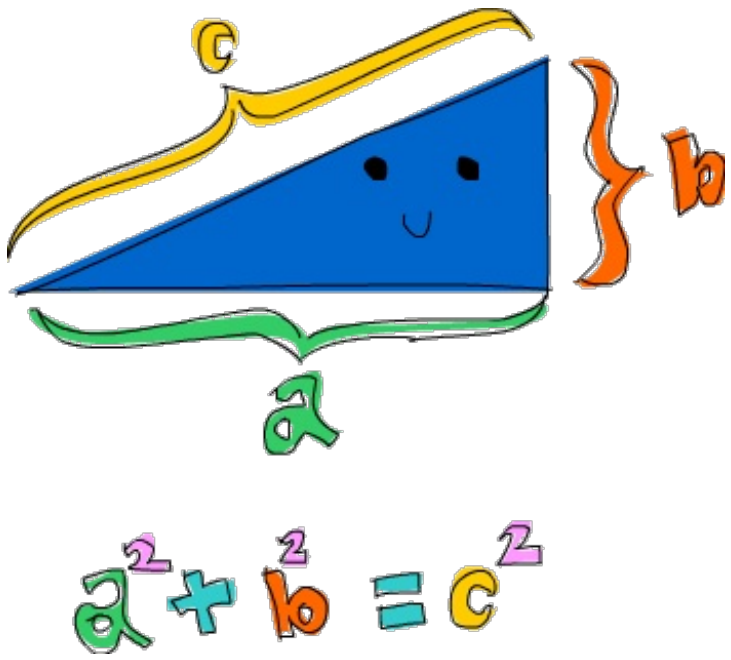
```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5] ["one", "two", "three", "four", "five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```

Elle met en paires les éléments et produit une nouvelle liste. Le premier élément va avec le premier, le deuxième avec le deuxième, etc. Remarquez que, puisque les paires peuvent avoir différents types en elles, `zip` peut prendre deux listes qui contiennent des éléments de différents types et les zipper. Que se passe-t-il si les longueurs des listes ne correspondent pas ?

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
```

La liste la plus longue est simplement coupée pour correspondre à la longueur de la plus courte. Puisqu'Haskell est paresseux, on peut zipper des listes finies avec des listes infinies :

```
ghci> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```



Voici un problème qui combine tuples et listes en compréhensions : quel triangle rectangle a des côtés tous entiers, tous inférieurs ou égaux à 10, et a un périmètre de 24 ? Premièrement, essayons de générer tous les triangles dont les côtés sont inférieurs ou égaux à 10 :

```
ghci> let triangles = [ (a,b,c) | c <- [1..10], b <- [1..10], a <- [1..10] ]
```

On pioche simplement dans trois listes et notre fonction de sortie combine les trois valeurs en triplets. Si vous évaluez en le tapant `triangles`, dans GHCi, vous obtiendrez tous les triangles possibles dont les côtés sont inférieurs ou égaux à 10. Ensuite, ajoutons une condition, que ceux-ci soient rectangles. On va également exploiter le fait que b est plus petit que l'hypoténuse et que le côté a est plus petit que b.

```
ghci> let rightTriangles = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b], a^2 + b^2 == c^2]
```

Nous y sommes presque. Maintenant, il ne reste plus qu'à modifier la fonction en disant que l'on veut ceux dont le périmètre est 24.

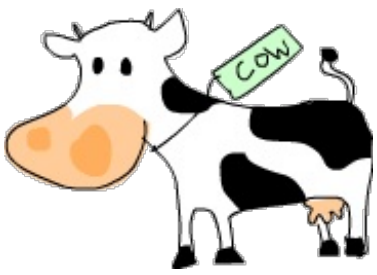
```
ghci> let rightTriangles' = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b], a^2 + b^2 == c^2, a+b+c == 24]
ghci> rightTriangles'
[(6,8,10)]
```

Et voilà notre réponse ! C'est un schéma courant en programmation fonctionnelle. Vous prenez un ensemble de solutions, et vous appliquez des transformations sur ces solutions et les filtrez jusqu'à obtenir les bonnes.

Types et classes de types

← Démarrons | Table des matières | Syntaxe des fonctions →

Faites confiance aux types



Nous avons mentionné auparavant qu'Haskell avait un système de types statique. Le type de chaque expression est connu à la compilation, ce qui mène à un code plus sûr. Si vous écrivez un programme dans lequel vous tentez de diviser un booléen par un nombre, cela ne compilera même pas. C'est pour le mieux, car il vaut mieux trouver ces erreurs à la compilation qu'obtenir un programme qui plante. En Haskell, tout a un type, donc le compilateur peut raisonner sur votre programme avant même de le compiler.

Au contraire de Java ou Pascal, Haskell a de l'inférence des types. Si nous écrivons un nombre, nous n'avons pas à dire à Haskell que c'est un nombre. Il peut l'inférer de lui-même, donc pas besoin de lui écrire explicitement les types des fonctions et des expressions pour arriver à faire quoi que ce soit. Nous avons vu une partie des bases de Haskell en ne regardant les types que très superficiellement. Cependant, comprendre le système de types est une part très importante de l'apprentissage d'Haskell.

Un type est une sorte d'étiquette que chaque expression porte. Il nous indique à quelle catégorie de choses cette expression appartient. L'expression `True` est un booléen, "hello" est une chaîne de caractères, etc.

Utilisons à présent GHCi pour examiner le type de quelques expressions. On le fera à l'aide de la commande `:t` qui, suivie d'une expression valide, nous indique son type. Essayons.

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```

Ici on voit que faire `:t` sur une expression affiche l'expression, suivie de `::` et de son type. `::` se lit "a pour type". Les types explicites sont toujours notés avec une initiale en majuscule. `'a'`, semblerait-il, a pour type `Char`. On en conclue rapidement que cela signifie *caractère*. `True` est de type `Bool`. C'est logique. Mais, et ça ? Examiner `"HELLO!"` renvoie un `[Char]`. Les crochets indiquent une liste. Nous lisons donc cela *liste de caractères*. Contrairement aux listes, chaque longueur de tuple a son propre type. Ainsi, l'expression `(True, 'a')` a pour type `(Bool, Char)`, alors que l'expression `('a', 'b', 'c')` aurait pour type `(Char, Char, Char)`. `4 == 5` renverra toujours `False`, son type est `Bool`.

Les fonctions aussi ont des types. Quand on écrit nos propres fonctions, on peut vouloir leur donner des déclarations de type explicites. C'est généralement considéré comme une bonne pratique, sauf quand on écrit des fonctions courtes. À présent, nous donnerons aux fonctions que nous déclarerons des déclarations de type. Vous souvenez-vous de la compréhension de liste que nous avions écrite, qui filtre une chaîne de caractères pour ne garder que ceux en majuscule ? Voici sa déclaration de type :

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

`removeNonUppercase` est de type `[Char] -> [Char]`, ce qui signifie qu'elle transforme une chaîne de caractères en une chaîne de caractères. Parce qu'elle prend une chaîne de caractères en paramètre, et en retourne une en résultat. Le type `[Char]` est synonyme de `String`, donc il est plus clair d'écrire `removeNonUppercase :: String -> String`. Nous n'avons pas eu à donner à cette fonction de déclaration de type car le compilateur pouvait inférer lui-même que la fonction allait de chaîne de caractères à chaîne de caractères, mais on l'a quand même fait. Mais comment écrire le type d'une fonction qui prend plusieurs paramètres ? Voici une simple fonction qui prend trois entiers et les somme :

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Les paramètres sont séparés par des `->` et il n'y a pas de distinction entre les paramètres et le type retourné. Le type retourné est le dernier élément de la déclaration, et les paramètres sont les trois premiers. Plus tard, nous verrons pourquoi ils sont séparés par des `->` plutôt que d'être distingués plus explicitement, par exemple sous une forme ressemblant à `Int, Int, Int -> Int` ou de ce genre.

Si vous voulez donner une déclaration de type à une fonction, mais n'êtes pas certain du type, vous pouvez toujours ne pas l'écrire, puis le demander à l'aide de `:t`. Les fonctions sont également des expressions, donc `:t` fonctionne dessus sans souci.

Voici un survol des types usuels.

`Int` désigne les entiers. `7` peut être un entier, mais `7.2` ne peut pas. `Int` est borné, ce qui signifie qu'il existe une valeur minimale et une valeur maximale. Habituellement, sur des machines 32-bits, le plus grand `Int` est 2147483647 et le plus petit –2147483648.

`Integer` désigne… euh, aussi les entiers. La différence, c'est qu'il n'est pas borné, donc il peut être utilisé pour représenter des nombres énormes. Je veux dire, vraiment énormes. `Int` est cependant plus efficace.

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

```
ghci> factorial 50
30414093201713378043612608166064768844377641568960512000000000000
```

`Float` est un nombre réel à virgule flottante en simple précision.

```
circumference :: Float -> Float
circumference r = 2 * pi * r
```

```
ghci> circumference 4.0
25.132742
```

`Double` est un nombre réel à virgule flottante en double précision !

```
circumference' :: Double -> Double
circumference' r = 2 * pi * r
```

```
ghci> circumference' 4.0
25.132741228718345
```

`Bool` est un type booléen. Il ne peut prendre que deux valeurs : `True` et `False`.

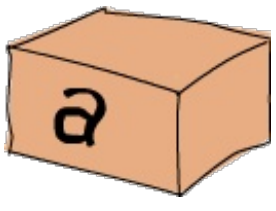
`Char` représente un caractère. Il est dénoté par des apostrophes. Une liste de caractères est une chaîne de caractères.

Les tuples sont des types mais dépendent de la taille et du type de leurs composantes, donc il existe théoriquement une infinité de types de tuples, ce qui fait trop pour être contenu dans ce tutoriel. Remarquez que le tuple vide `()` est aussi un type, qui ne contient qu'une seule valeur : `()`.

Variables de type

Quel est le type de `head` d'après vous ? Puisque `head` prend une liste de n'importe quel type et retourne son premier élément, qu'est-ce que ça peut bien être ? Regardons !

```
ghci> :t head
head :: [a] -> a
```



Hmmm ! C'est quoi ce `a` ? Un type ? Rappelez-vous, nous avions dit que les types doivent commencer par une majuscule, donc ça ne peut pas être exactement un type. Puisqu'il commence par une minuscule, c'est une **variable de type**. Cela signifie que `a` peut être de n'importe quel type. C'est très proche des types génériques dans d'autres langages, seulement en Haskell c'est encore plus puissant puisque cela nous permet d'écrire des fonctions très générales tant qu'elles n'utilisent pas les spécificités d'un type particulier. Des fonctions ayant des variables de type sont dites **fonctions polymorphiques**. La déclaration de type de `head` indique qu'elle prend une liste de n'importe quel type, et retourne un élément de ce type.

Bien que les variables de type puissent avoir des noms plus longs qu'un caractère, généralement on les note a, b, c, d…

Vous souvenez-vous de `fst` ? Elle retourne la première composante d'une paire. Regardons son type.

```
ghci> :t fst
fst :: (a, b) -> a
```

On voit que `fst` prend un tuple qui contient deux types et retourne un élément du même type que celui de la première composante de la paire. C'est pourquoi on peut utiliser `fst` sur une paire qui contient n'importe quels deux types. Notez que, le fait que `a` et `b` soient deux variables de type différentes n'implique pas que les types doivent être différents. Cela indique juste que la première composante et la valeur retournée ont le même type.

Classes de types 101

Une classe de types est une sorte d'interface qui définit un comportement. Si un type appartient à une classe de types, cela signifie qu'il supporte et implémente le comportement décrit par la classe. Beaucoup de gens venant de la programmation orientée objet se méprennent à penser que les classes de types sont comme des classes de langages orientés objet. Ce n'est pas le cas. Vous pouvez plutôt les penser comme des sortes d'interfaces Java, mais en mieux.

Quelle est la signature de type de `==` ?

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

Note : l'opérateur d'égalité, `==` est une fonction. De même pour `+`, `*`, `-`, `/` et quasiment tous les opérateurs. Si une fonction ne contient que des caractères spéciaux, elle est considérée infixe par défaut. Si nous voulons examiner son type, la passer à une autre fonction, ou l'appeler de façon préfixe, nous devons l'entourer de parenthèses.

Intéressant. Nous voyons quelque chose de nouveau ici, le symbole `=>`. Tout ce qui se situe avant le `=>` est appelé une **contrainte de classe**. On peut lire la déclaration de type précédente ainsi : la fonction d'égalité prend deux valeurs du même type et retourne un `Bool`. Le type de ces valeurs doit être membre de la classe `Eq` (ceci est la contrainte de classe).

La classe de types `Eq` offre une interface pour tester l'égalité. Tout type pour lequel tester l'égalité de deux valeurs a du sens devrait être membre de la classe de types `Eq`. Tous les types standards de Haskell à l'exception d'IO (le type qui permet de faire des entrées-sorties) et des fonctions font partie de la classe de types `Eq`.

La fonction `elem` a pour type `(Eq a) => a -> [a] -> Bool` car elle utilise `==` sur les éléments de la liste pour vérifier si la valeur qu'on cherche est dedans.

Quelques classes de types de base :

`Eq` est utilisé pour les types qui supportent un test d'égalité. Ses membres doivent implémenter les fonctions `==` ou `/=` dans leur définition. Donc, si une fonction a une contrainte de classe `Eq` pour une variable de type, c'est que quelque part dans la fonction, elle utilise `==` ou `/=`. Tous les types mentionnés précédemment à l'exception des fonctions sont membres de `Eq`, donc ils peuvent être testés égaux.

```
ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

`Ord` est pour les types qui peuvent être ordonnés.

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

Tous les types que nous avons couverts jusqu'ici sont membres d'`Ord`. `Ord` comprend toutes les fonctions usuelles de comparaison, comme `>`, `<`, `>=` et `<=`. La fonction `compare` prend deux membres d'`Ord` ayant le même type et retourne un ordre. `Ordering` est son type, et les valeurs peuvent être `GT`, `LT` ou `EQ`, respectivement pour *plus grand*, *plus petit* et *égal*.

Pour être membre d'`Ord`, un type doit d'abord être membre du club prestigieux et exclusif `Eq`.

```
ghci> "Abrakadabra" < "Zebra"
True
```

```
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

Les membres de `Show` peuvent être représentés par une chaîne de caractères. Tous les types couverts jusqu'ici sont membres de `Show`. La fonction la plus utilisée en rapport avec la classe `Show` est `show`. Elle prend une valeur d'un type membre de la classe `Show` et nous la représente sous la forme d'une chaîne de caractères.

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

`Read` est en quelque sorte l'opposé de `Show`. La fonction `read` prend une chaîne de caractères, et retourne une valeur d'un type membre de `Read`.

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

Jusqu'ici tout va bien. Encore une fois, tous les types couverts jusqu'ici sont dans cette classe de types. Mais que se passe-t-il si l'on écrit juste `read "4"` ?

```
ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

GHCi nous dit qu'il ne sait pas ce qu'on attend en retour. Remarquez comme, dans les exemples précédents de `read`, on faisait toujours quelque chose du résultat. Ainsi, GHCi pouvait inférer le type de résultat attendu de `read`. Si nous l'utilisions comme un booléen, il savait qu'il fallait retourner un `Bool`. Mais maintenant, il sait seulement qu'on attend un type de la classe `Read`, mais il ne sait pas lequel. Regardons la signature de type de `read`.

```
ghci> :t read
read :: (Read a) => String -> a
```

Vous voyez ? Il retourne un type membre de `Read` mais si on n'essaie pas de l'utiliser ensuite, il n'a aucun moyen de savoir de quel type il s'agit. Pour remédier à cela, on peut utiliser des **annotations de type** explicites. Les annotations de type sont un moyen d'exprimer explicitement le type qu'une expression doit avoir. On le fait en ajoutant `::` à la fin de l'expression et en spécifiant un type. Observez :

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

La plupart des expressions sont telles que le compilateur peut inférer leur type tout seul. Mais parfois, le compilateur ne sait pas s'il doit plutôt retourner une valeur de type `Int` ou `Float` pour une expression comme `read "5"`. Pour voir le type, Haskell devrait évaluer `read "5"`. Mais puisqu'il est statiquement typé, il doit connaître les types avant de compiler le code (ou, dans le cas de GHCi, de l'évaluer). Donc nous devons dire à Haskell : "Hé, cette expression devrait avoir ce type, au cas où tu ne saurais pas !".

Les membres d'`Enum` sont des types ordonnés séquentiellement - on peut les énumérer. L'avantage de la classe de types `Enum` est qu'on peut l'utiliser ses types dans les progressions. Elle définit aussi un successeur et un prédécesseur, qu'on peut obtenir à l'aide des fonctions `succ` et `pred`. Les types membres de cette classe sont : `()`, `Bool`, `Char`, `Ordering`, `Int`, `Integer`, `Float` et `Double`.

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ 'B'
'C'
```

Les membres de `Bounded` ont une borne supérieure et inférieure.

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False
```

`minBound` et `maxBound` sont intéressantes car elles ont pour type `(Bounded a) => a`. D'une certaine façon, ce sont des constantes polymorphiques.

Tous les tuples sont membres de `Bounded` si leurs composantes le sont également.

```
ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

`Num` est la classe des types numériques. Ses membres ont pour propriété de pouvoir se comporter comme des nombres. Examinons le type d'un nombre.

```
ghci> :t 20
20 :: (Num t) => t
```

Il semblerait que tous les nombres soient aussi des constantes polymorphiques. Ils peuvent se faire passer pour tout membre de la classe `Num`.

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

Ce sont tous des types membres de `Num`. Si on examine le type de `*`, on verra qu'elle accepte des nombres.

```
ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

Elle prend deux nombres du même type, et retourne un nombre de ce type. C'est pourquoi `(5 :: Int) * (6 :: Integer)` donnera une erreur de typage, alors que `5 * (6 :: Integer)` fonctionnera et produira un `Integer` car `5` peut se faire passer pour un `Integer` ou un `Int`.

Pour être dans `Num`, un type doit déjà être ami avec `Show` et `Eq`.

`Integral` est aussi une classe de types numériques. `Num` inclue tous les nombres, autant réels qu'entiers, `Integral` ne contient que des entiers. `Int` et `Integer` sont membres de cette classe.

`Floating` inclue seulement des nombres à virgule flottante, donc `Float` et `Double`.

Une fonction très utile pour manipuler des nombres est `fromIntegral`. Sa déclaration de type est `fromIntegral :: (Num b, Integral a) => a -> b`. De cette signature, on voit qu'elle prend un entier et le transforme en un nombre plus général. C'est utile lorsque vous voulez utiliser des entiers et des nombres à virgule flottante ensemble proprement. Par exemple, `length` a pour déclaration de type `length :: [a] -> Int` au lieu d'avoir le type plus général `(Num b) => length :: [a] -> b`. Ainsi, si l'on essaie d'obtenir la taille d'une liste, et de l'ajouter à `3.2`, on aura une erreur puisqu'on essaie d'ajouter un `Int` à un nombre à virgule flottante. Pour éviter ceci, on fait `fromIntegral (length [1,2,3,4]) + 3.2` et tout fonctionne.

Remarquez que `fromIntegral` a plusieurs contraintes de classe dans sa signature. C'est tout à fait valide, et comme vous voyez, les contraintes sont séparées par des virgules dans les parenthèses.

Syntaxe des fonctions

⬅ Types et classes de types | Table des matières | Récursivité ➡

Filtrage par motif

Ce chapitre couvrira certaines des constructions syntaxiques cool d'Haskell, et l'on va commencer par le filtrage par motif. Le filtrage par motif consiste à spécifier des motifs auxquels une donnée doit se conformer, puis vérifier si c'est le cas en déconstruisant la donnée selon ces motifs.

Lorsque vous définissez des fonctions, vous pouvez définir des corps de fonction séparés pour différents motifs. Cela amène à un code très clair et lisible. Vous pouvez filtrer par motif tout type de donnée - des nombres, des caractères, des listes, des tuples, etc. Créons une fonction triviale qui vérifie si le nombre fourni est sept ou non.

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

Lorsque vous appelez `lucky`, les motifs seront vérifiés de haut en bas, et lorsque le paramètre se conforme à un motif, le corps de fonction correspondant sera utilisé. Ici, le seul moyen pour un nombre de correspondre au premier motif est pour lui d'être le nombre 7. Si ce n'est pas le cas, on passe au second motif, qui accepte tout paramètre et l'attache au nom `x`. Cette fonction aurait pu être implémentée à l'aide d'une construction *if*. Mais, si l'on voulait que la fonction énonce les nombres entre 1 et 5, et déclare `"Not between 1 and 5"` pour tout autre nombre ? Sans filtrage par motif, nous devrions utiliser un arbre de constructions *if then else* plutôt alambiqué. Cependant, avec le filtrage :

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

Remarquez que si nous avions mis le dernier motif (celui qui attrape tout) tout en haut, la fonction répondrait toujours `"Not between 1 and 5"`, car le motif attraperait tous les nombres, et ils n'auraient pas l'opportunité d'être testés contre les autres motifs.

Rappelez-vous de la fonction factorielle qu'on avait implémentée auparavant. Nous l'avions défini pour un nombre `n` comme `product [1..n]`. On peut également la définir de manière *récursive*, comme en mathématiques. On commence par dire que la factorielle de 0 est 1. Puis, on dit que la factorielle d'un entier positif est égale au produit de cet entier et de la factorielle de son prédécesseur. En Haskell :

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

C'est la première fois qu'on définit une fonction récursivement. La récursivité est importante en Haskell, et nous y reviendrons plus en détail. Rapidement, ce qui se passe si l'on demande la factorielle de 3. Il essaie de calculer `3 * factorial 2`. La factorielle de 2 est `2 * factorial 1`, donc on a `3 * (2 * factorial 1)`. `factorial 1` vaut `1 * factorial 0`, donc on a `3 * (2 * (1 * factorial 0))`. Ici est l'astuce, on a défini factorielle de 0 comme 1, et puisque ce motif vient avant celui qui accepte tout, cela retourne 1. Le résultat final est donc équivalent à `3 * (2 * (1 * 1))`. Si nous avions placé le second motif au dessus du premier, il attraperait tous les nombres, y compris 0, et le calcul ne terminerait jamais. C'est pourquoi l'ordre est important dans la spécification des motifs, et il est toujours préférable de préciser les motifs les plus spécifiques en premier, et les plus généraux ensuite.

Le filtrage par motif peut aussi échouer. Si l'on définit une fonction :

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

et qu'on essaie de l'appeler avec une entrée inattendue, voilà ce qui arrive :

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'h'
"*** Exception: tut.hs:(53,0)-(55,21): Non-exhaustive patterns in function charName
```

Il se plaint que les motifs ne soient pas exhaustifs, et il a raison. Lorsqu'on utilise des motifs, il faut toujours penser à inclure un motif qui attrape le reste de façon à ce que le programme ne plante pas sur une entrée inattendue.

Le filtrage de motifs s'utilise aussi sur les tuples. Comment écrire une fonction qui prend deux vecteurs en 2D (sous forme de paire) et les somme ? Pour sommer les deux vecteurs, on somme séparément les composantes en x et les composantes en y. Voilà ce que nous aurions fait sans filtrage par motif :

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

Bon, ça marche, mais il y a une meilleure façon de faire. Utilisons le filtrage par motif.

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

Et voilà ! Beaucoup mieux. Remarquez que ceci est déjà un motif attrape-tout. Le type de `addVectors` (dans les deux cas) est `addVectors :: (Num a) => (a, a) -> (a, a) - > (a, a)`, donc nous sommes certains d'avoir deux paires en paramètres.

`fst` et `snd` extraient les composantes d'une paire. Mais pour des triplets ? Eh bien, il n'y a pas de fonction fournie pour faire ça. On peut tout de même l'écrire nous même.

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

Le `_` signifie la même chose que dans les listes en compréhension. Il signifie qu'on se fiche complètement de ce que cette partie est, donc on met juste un `_`.

Ce qui me fait penser, vous pouvez aussi utiliser des motifs dans les listes en compréhension. Voyez par vous-même :

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

Si un filtrage échoue, ça passera simplement au prochain élément.

Les listes peuvent elles-même être utilisées pour le filtrage. Vous pouvez filtrer la liste vide `[]`, ou un motif incluant `:` et la liste vide. Puisque `[1, 2, 3]` est juste du sucre syntaxique pour `1:2:3:[]`, vous pouvez utiliser ce premier. Un motif de la forme `x:xs` attachera la tête à `x` et le reste à `xs`, même s'il n'y a qu'un seul élément, auquel cas `xs` sera la liste vide.

Note : Le motif `x:xs` est très utilisé, particulièrement dans les fonctions récursives. Mais les motifs qui ont un `:` ne peuvent valider que des listes de longueur 1 ou plus.

Si vous désirez lier, mettons, les trois premiers éléments à trois variables, et le reste de la liste à une autre variable, vous pouvez utiliser un motif comme `x:y:z:xs`. Il ne validera que des listes qui ont trois éléments ou plus.

Maintenant que l'on sait utiliser le filtrage par motif sur des listes, implémentons notre fonction `head`.

```
head' :: [a] -> a
head' [] = error "Can't call head on an empty list, dummy!"
head' (x:_) = x
```

Vérifions :

```
ghci> head' [4,5,6]
4
ghci> head' "Hello"
'H'
```

Cool ! Remarquez, si l'on souhaite lier plusieurs variables (même si l'une d'entre elles est `_` et ne lie pas vraiment), il faut les mettre entre parenthèses. Remarquez aussi la fonction `error` qu'on a utilisée. Elle prend une chaîne de caractères et génère une erreur à l'exécution, en utilisant cette chaîne pour indiquer quel genre d'erreur s'est produit. Elle fait planter le programme, donc il ne faut pas trop l'utiliser. Mais appeler `head` sur une liste vide n'a pas vraiment de sens.

Créons une fonction triviale qui nous dit quelques éléments d'une liste en anglais.

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y
tell (x:y:_) = "This list is long. The first two elements are: " ++ show x ++ " and " ++ show y
```

Cette fonction est sûre car elle prend soin de la liste vide, d'une liste singleton, d'une liste à deux éléments, et d'une liste à plus de deux éléments. Remarquez que `(x:[])` et `(x:y:[])` pourraient être réécrits respectivement `[x]` et `[x, y]` (ceci étant du sucre syntaxique, on n'a plus besoin des parenthèses). On ne peut pas réécrire `(x:y:_)` de manière analogue car le motif attrape toutes les listes de taille supérieure à 2.

Nous avons déjà implémenté `length` à l'aide d'une liste en compréhension. Utilisons à présent du filtrage par motif et de la récursivité :

```
length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:xs) = 1 + length' xs
```

C'est très similaire à la fonction factorielle écrite plus tôt. D'abord, on définit le résultat d'une entrée particulière - la liste vide. On parle de cas de base. Puis dans le second motif, on démonte une liste en séparant sa tête et sa queue. On dit alors que la longueur est égale à 1 plus la longueur de la queue. On utilise `_` pour filtrer la tête car on se fiche de sa valeur. Remarquez aussi qu'on a pris en compte tous les cas possibles. Le premier motif accepte une liste vide, le second accepte toute liste non vide.

Voyons ce qui se passe lorsqu'on appelle `length'` sur `"ham"`. D'abord, on vérifie si c'est une liste vide. Ce n'est pas le cas, on descend au second motif. Le second motif est validé, et la longueur à calculer est `1 + length' "am"`, car on a cassé la liste entre tête et queue, et jeté la tête. Ok. La `length'` de `"am"` est, similairement, `1 + length' "m"`. Donc, pour l'instant, nous avons `1 + (1 + length' "m")`. `length' "m"` vaut `1 + length' ""` (qu'on peut aussi écrire `1 + length' []`). Et nous avons défini `length' []` comme `0`. Donc, au final on a `1 + (1 + (1 + 0))`.

Implémentons `sum`. On sait que la somme d'une liste vide est 0. On l'écrit comme un motif. Ensuite, on sait que la somme d'une liste est égale à la tête plus la somme du reste. Cela donne :

```
sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

Il y a aussi quelque chose qu'on appelle des motifs nommés. C'est une manière pratique de détruire quelque chose selon un motif tout en gardant une référence à la chose entière. Vous pouvez utiliser un `@` devant le motif à cet effet. Par exemple, le motif `xs@(x:y:ys)`. Ce motif acceptera exactement les mêmes choses que `x:y:ys`, mais vous pourrez facilement désigner la liste complète via `xs` au lieu de réécrire `x:y:ys` dans le corps de la fonction. Exemple simple :

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

```
ghci> capital "Dracula"
"The first letter of Dracula is D"
```

Normalement, on utilise des motifs nommés pour éviter de se répéter quand on filtre un gros motif mais qu'on a besoin de la chose entière à nouveau dans le corps de la fonction.

Autre chose - vous ne pouvez pas utiliser `++` dans les motifs. Si vous essayez de filtrer `(xs ++ ys)`, qu'est-ce qui irait dans la première ou dans la seconde liste ? Ça n'a pas trop de sens. Cela aurait du sens de filtrer contre `(xs ++ [x, y, z])` ou juste `(xs ++ [x])`, mais par nature des listes, cela est impossible.

Gardes, gardes !

Alors que les motifs sont un moyen de vérifier qu'une valeur se conforme à une forme et de la déconstruire, les gardes permettent de vérifier si des propriétés d'une valeur (ou de plusieurs d'entre elles) sont vraies ou fausses. Ça ressemble étrangement à ce que fait une structure *if*, et c'est en fait très similaire. Le truc, c'est que les gardes sont plus lisibles lors de multiples conditions, et elles fonctionnent très bien en combinaison avec les motifs.

Plutôt que d'expliquer leur syntaxe, plongeons plutôt dans le bain et créons une fonction avec des gardes. Nous allons faire une fonction qui vous classe en fonction de votre IMC (indice de masse corporelle, BMI en anglais pour body mass index). Si votre IMC est inférieur à 18.5, vous êtes considéré en sous-poids. S'il est entre 18.5 et 25, c'est normal. 25 à 30 correspond à un surpoids, et plus de 30 à de l'obésité. Voici la fonction (pour l'instant, elle ne va pas calculer l'indice, juste prendre un indice et vous donner votre classification).

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
```

Les gardes sont indiquées par des barres verticales à la suite d'une fonction et de ses paramètres. Elles sont généralement indentées un peu à droite et alignées. Une garde est simplement une expression booléenne. Si elle est évaluée à `True`, le corps de la fonction correspondant est utilisé. Si elle s'évalue à `False`, la vérification prend la prochaine garde, etc. Si on appelle cette fonction avec `24.3`, elle va d'abord vérifier si c'est plus petit ou égal à `18.5`. Puisque ce n'est pas le cas, elle passe à la prochaine garde. La vérification est effectuée avec la deuxième garde, et puisque 24.3 est inférieur à 25.0, la second chaîne est retournée.

Cela rappelle fortement les grands arbres de *if else* dans les langages impératifs, seulement c'est beaucoup mieux et plus lisible. Alors que les grands arbres de *if else* sont généralement boudés, parfois un problème est défini de telle sorte que l'on ne peut pas vraiment les éviter. Les gardes sont une alternative à cela.

Très souvent, la dernière garde est `otherwise`. `otherwise` est simplement définie comme `otherwise = True` et attrape donc tout. C'est très similaire aux motifs, sauf qu'eux vérifient que l'entrée satisfait un motif alors que les gardes vérifient des conditions booléennes. Si toutes les gardes d'une fonction sont évaluées à `False` (donc, qu'on n'a pas fourni de garde `otherwise` qui vaut toujours `True`), l'évaluation passe au **motif** suivant. C'est ainsi que les motifs et les gardes s'entendent bien. Si aucune combinaison de gardes et de motifs n'est satisfaite, une erreur est levée.

Bien sûr, on peut utiliser les gardes sur des fonctions qui prennent autant de paramètres que l'on souhaite. Plutôt que de calculer son IMC avant d'appeler la fonction, modifions-là pour qu'elle prenne notre taille et notre masse et le calcule pour nous.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

Voyons si je suis gros…

```
ghci> bmiTell 85 1.90
"You're supposedly normal. Pffft, I bet you're ugly!"
```

Youpi ! Je ne suis pas gros ! Mais Haskell vient de me traiter de moche. Peu importe !

Notez qu'il n'y a pas de `=` après les paramètres et avant les gardes. Beaucoup de débutants font des erreurs de syntaxe en le mettant ici.

Un autre exemple simple : implémentons notre propre fonction `max`. Si vous vous souvenez, elle prend deux choses comparables et retourne la plus grande d'entre elles.

```
max' :: (Ord a) => a -> a -> a
max' a b
    | a > b     = a
    | otherwise = b
```

Vous pouvez aussi écrire les gardes sur la même ligne, bien que je le déconseille car c'est moins lisible, même pour des fonctions courtes. À titre d'exemple, on aurait pu écrire :

```
max' :: (Ord a) => a -> a -> a
max' a b | a > b = a | otherwise = b
```

Erf ! Pas très lisible tout ça ! Passons : implémentons notre propre `compare` à l'aide de gardes.

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
    | a > b     = GT
    | a == b    = EQ
    | otherwise = LT
```

```
ghci> 3 `myCompare` 2
GT
```

Note : Non seulement pouvons-nous appeler des fonctions de manière infixe avec des apostrophes renversées, on peut également les définir de cette manière. Parfois, c'est plus simple à lire comme ça.

Où !?

Dans la section précédente, nous définissions le calculateur d'IMC ainsi :

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

Remarquez-vous comme on se répète trois fois ? On se répète, trois fois. Se répéter (trois fois) lorsqu'on programme est aussi souhaitable qu'un coup de pied dans la tronche. Plutôt que de répéter la même expression trois fois, il serait idéal de la calculer une fois, de lier le résultat à un nom, et d'utiliser ce nom partout en lieu et place de l'expression. Eh bien, on peut modifier la fonction ainsi :

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
```

Le mot-clé `where` est placé après les gardes (qu'on indente généralement autant que les gardes) et est suivi de plusieurs définitions de noms ou de fonctions. Ces noms sont visibles à travers toutes les gardes, ce qui nous donne l'avantage de ne pas avoir à nous répéter. Si on décide de calculer l'IMC différemment, il suffit de changer la formule à un endroit. Cela facilite aussi la lisibilité en donnant des noms aux choses, et peut rendre votre programme plus rapide puisque la variable `bmi` est ici calculée une unique fois. On pourrait faire un peu plus de zèle et écrire :

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

Les noms qu'on définit dans la section *where* d'une fonction ne sont visibles que pour cette fonction, donc on n'a pas à se préoccuper de polluer l'espace de nommage d'autres fonctions. Remarquez que tous les noms sont alignés sur une même colonne. Si ce n'était pas le cas, Haskell serait confus car il ne saurait pas vraiment s'ils font partie du même bloc de définitions.

Les liaisons créés dans des clauses *where* ne sont pas partagées par les corps de différentes fonctions. Si vous voulez que plusieurs fonctions aient accès au même nom, il faut le définir globalement.

Vous pouvez aussi utiliser les liaisons *where* pour du **filtrage par motif** ! On aurait pu réécrire la section *where* précédente comme :

```
…
where bmi = weight / height ^ 2
      (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

Créons une autre fonction triviale qui prend un prénom, un nom, et renvoie les initiales.

```haskell
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

Bon, on aurait pu faire le filtrage par motif directement dans les paramètres de la fonction (c'était plus court et plus clair à vrai dire), mais c'est juste histoire de montrer qu'on peut aussi le faire dans les liaisons du *where*.

Comme nous avions défini des constantes dans les blocs *where*, on peut également y définir des fonctions. Pour continuer sur le thème de la programmation diététique, créons une fonction qui prend une liste de paires masse-taille et retourne une liste d'IMC.

```haskell
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
    where bmi weight height = weight / height ^ 2
```

Et c'est tout ! La raison pour laquelle nous avons introduit `bmi` en tant que fonction ici est qu'il y a plus d'un IMC à calculer. Il faut examiner toute la liste, et calculer un IMC différent pour chaque paire.

Les liaisons *where* peuvent aussi être imbriquées. C'est un idiome habituel de créer une fonction et de définir des fonctions auxiliaires dans sa clause *where* et de définir des fonctions auxiliaires à ces fonctions auxiliaires dans leur clause *where*.

Let it be

Les liaisons *let* sont très similaires aux liaisons *where*. Les liaisons *where* sont une construction syntaxique permettant de lier des variables en fin de fonction, visibles depuis toute la fonction, y compris les gardes. Les liaisons *let* vous permettent de lier à des variables n'importe où, et sont elles-mêmes des expressions, mais très locales, donc elles ne sont pas visibles à travers les gardes. Comme toutes les constructions Haskell qui lient des valeurs à des noms, elles supportent le filtrage par motif. Voyons plutôt en action ! Voici comment l'on définit une fonction qui nous donne la surface d'un cylindre à partir de sa hauteur et de son rayon :

```haskell
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

La liaison est de la forme `let <bindings> in <expression>`. Les noms que vous définissez dans *let* sont accessibles dans l'expression qui suit le *in*.

La différence, c'est que les liaisons *let* sont elles-mêmes des expressions. Là où les liaisons *where* sont seulement des constructions syntaxiques. Vous vous souvenez que nous avions vu qu'une construction *if* était une expression, et que par conséquent vous pouviez l'utiliser n'importe où ?

```haskell
ghci> [if 5 > 3 then "Woo" else "Boo", if 'a' > 'b' then "Foo" else "Bar"]
["Woo", "Bar"]
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
```

Il en va de même des liaisons *let*.

```haskell
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

On peut aussi introduire des fonctions à visibilité locale :

```haskell
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
```

Pour définir plusieurs liaisons sur la même ligne, on ne peut évidemment pas les aligner sur la même colonne. On peut donc utiliser des points-virgules comme séparateurs.

```haskell
ghci> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar = "there!" in foo ++ bar)
(6000000,"Hey there!")
```

Le point-virgule situé après la dernière liaison n'est pas nécessaire, mais peut être écrit à votre convenance. Comme on l'a vu, vous pouvez utiliser du filtrage par

motif. C'est très utile pour démanteler rapidement un tuple en ses composantes et donner un nom à chacune d'elles :

```
ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

Vous pouvez aussi mettre des liaisons *let* dans une liste en compréhension. Réécrivons l'exemple précédent du calcul de l'IMC sur des listes de paires masse-taille, en plaçant un *let* dans une liste en compréhension plutôt que d'utiliser un *where*.

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

Nous incluons un *let* dans une liste en compréhension comme un prédicat, seulement qu'il ne filtre pas la liste, mais lie des noms. Les noms définis dans le *let* dans la compréhension sont visibles de la fonction de retour (celle qui est avant le `|`) et de tous les prédicats et sections qui viennent après la liaison. On pourrait faire une fonction qui ne retourne que les IMC des personnes obèses :

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi >= 25.0]
```

On ne peut pas utiliser **`bmi`** dans la partie **`(w, h) <- xs`** car elle est définie avant la liaison **`let`**.

Nous avons omis la partie *in* de la liaison *let* lorsqu'on l'utilisait dans des listes en compréhension, parce que la visibilité des noms est déjà prédéfinie ici. Cependant, on pourrait placer une liaison *let* dans un prédicat afin que les noms définis ne soient visibles que du prédicat. La partie *in* peut aussi être omise lorsqu'on définit des fonctions et des constantes directement dans GHCi. Dans ce cas, les noms sont visibles de toute la session interactive.

```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
<interactive>:1:0: Not in scope: `boot'
```

Si les liaisons *let* sont cool, pourquoi ne pas les utiliser tout le temps, au lieu d'avoir recours à des liaisons *where*, vous vous demandez ? Eh bien, puisque les liaisons *let* sont des expressions très locales dans leur visibilité, on ne peut pas les utiliser à travers les gardes. Certaines personnes préfèrent également que les liaisons *where* se situent après les fonctions qui s'en servent. De cette manière, le corps de la fonction est plus proche de son nom, de sa déclaration de type, et le tout est plus lisible.

Expressions case

Beaucoup de langages impératifs (C, C++, Java, etc.) ont une syntaxe *case*, et si vous avez déjà programmé dans ceux-ci, vous savez probablement de quoi il s'agit. Il s'agit donc de prendre une variable, et d'exécuter un code spécifique en fonction de sa valeur, et éventuellement d'avoir un bloc attrape-tout si la variable a une valeur pour laquelle aucun cas n'est écrit.

Haskell prend ce concept, et l'améliore. Comme le nom l'indique, les expressions *case* sont, eh bien, des expressions, comme les expressions *if else* et les liaisons *let*. Non seulement on peut évaluer ces expressions selon plusieurs cas possibles de la valeur d'une variable, mais on peut aussi faire du filtrage par motif. Hmmm, prendre une variable, la filtrer par motif, évaluer des morceaux de code en fonction de sa valeur, est-ce qu'on n'aurait pas déjà fait ça auparavant ? Bien sûr, le filtrage par motif des paramètres d'une définition de fonction ! Eh bien, il s'agit en fait seulement d'un sucre syntaxique pour les expressions *case*. Ainsi, ces deux bouts de code sont parfaitement interchangeables :

```
head' :: [a] -> a
head' [] = error "No head for empty lists!"
head' (x:_) = x
```

```
head' :: [a] -> a
head' xs = case xs of [] -> error "No head for empty lists!"
                      (x:_) -> x
```

Comme vous pouvez le voir, la syntaxe des expressions *case* est plutôt simple :

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   …
```

`expression` est testée contre les motifs. L'action de filtrage est comme attendue : le premier motif qui correspond est utilisé. Si l'on parcourt toute l'expression *case* sans trouver de motif validé, une erreur d'exécution a lieu.

Alors que le filtrage par motif sur les paramètres d'une fonction ne peut se faire que dans la définition, les expressions *case* peuvent être utilisées à peu près partout. Par exemple :

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
```

Elles sont utiles pour faire du filtrage par motif sur des choses en plein milieu d'une expression. Du fait que le filtrage par motif dans les définitions de fonction est seulement du sucre syntaxique pour des expressions *case*, on aurait aussi pu définir :

```
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what [] = "empty."
          what [x] = "a singleton list."
          what xs = "a longer list."
```

Récursivité

⬅ Syntaxe des fonctions Table des matières Fonctions d'ordre supérieur ➡

Bonjour récursivité !

Nous avons rapidement mentionné la récursivité dans le chapitre précédent. Dans celui-ci, regardons-là de plus près, pourquoi elle est importante en Haskell et comment on peut écrire des solutions concises et élégantes à certains problèmes en réfléchissant récursivement.

Si vous ne savez toujours pas ce qu'est la récursivité, relisez cette phrase. Haha ! Je blague ! La récursivité est en fait une manière de définir des fonctions dans laquelle on utilise la fonction dans sa propre définition. Les définitions mathématiques sont souvent énoncées récursivement. Par exemple, la suite de Fibonacci est définie récursivement. D'abord, on définit les deux premiers nombres de Fibonacci de manière non-récursive. On dit que *F(0) = 0* et *F(1) = 1*, pour dire que le 0ème et le 1er chiffre sont 0 et 1, respectivement. Puis, on dit que pour tout entier naturel, ce nombre de Fibonacci est la somme des deux précédents nombres de Fibonacci. Donc *F(n) = F(n–1) + F(n–2)* . Ainsi, *F(3)* est *F(2) + F(1)*, qui est *(F(1) + F(0)) + F(1)*. Puisque l'on vient d'atteindre des nombres tous définis non-récursivement, on peut déclarer que *F(3)* vaut 2. Un ou plusieurs éléments définis non-récursivement dans une définition récursive sont appelés les **cas de base** de récursivité, et sont importants afin d'assurer que la fonction récursive termine. Sans eux, si *F(0)* et *F(1)* n'avaient pas été définis ainsi, on n'obtiendrait jamais de solution car on atteindrait 0, et on continuerait à décroître. Au bout d'un moment, on dirait que *F(–2000)* est *F(–2001) + F(–2002)* , et on en verrait toujours pas le bout !

La récursivité est importante en Haskell car, contrairement aux langages impératifs, vous faites des calculs en Haskell en définissant ce que les choses *sont* et non pas *comment* on les obtient. C'est pour cela qu'il n'y a pas de boucles *while* et *for* en Haskell, et à la place, on utilise souvent la récursivité pour déclarer ce qu'est quelque chose.

Maximum de fun

La fonction `maximum` prend une liste de choses qui peuvent être ordonnées (i.e. des instances de la classe `Ord`) et retourne la plus grande d'entre elles. Comment implémenteriez-vous ceci de manière impérative ? Vous auriez sûrement une variable qui contiendrait la plus grande valeur rencontrée jusqu'ici, et vous boucleriez à travers les éléments de la liste, remplaçant le maximum courant par un élément s'il s'avère être plus grand. La valeur maximale serait alors le maximum courant une fois arrivé à la fin. Wouah ! Que de mots pour décrire un algorithme aussi simple !

Voyons comment nous le définirions récursivement. Nous aurions d'abord un cas de base, nous disant que le maximum d'une liste singleton est son seul élément. Puis, le maximum d'une liste plus longue est sa tête si elle est plus grande que le maximum de la queue. Sinon, c'est ce dernier qui est le maximum de la liste ! Et voilà ! Implémentons ça en Haskell.

```haskell
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

Comme vous le voyez, le filtrage par motif va bien de pair avec la récursivité ! La plupart des langages impératifs ne proposent pas de filtrage par motif, donc vous êtes obligé d'utiliser plein de *if else* pour tester les cas de base. Ici, on les met simplement dans des motifs. La première condition dit : si la liste est vide, plante ! Raisonnable, après tout quel est le maximum d'une liste vide ? Je ne sais pas. Le deuxième motif est aussi un cas de base. Il dit que pour une liste singleton, il suffit de retourner son unique élément.

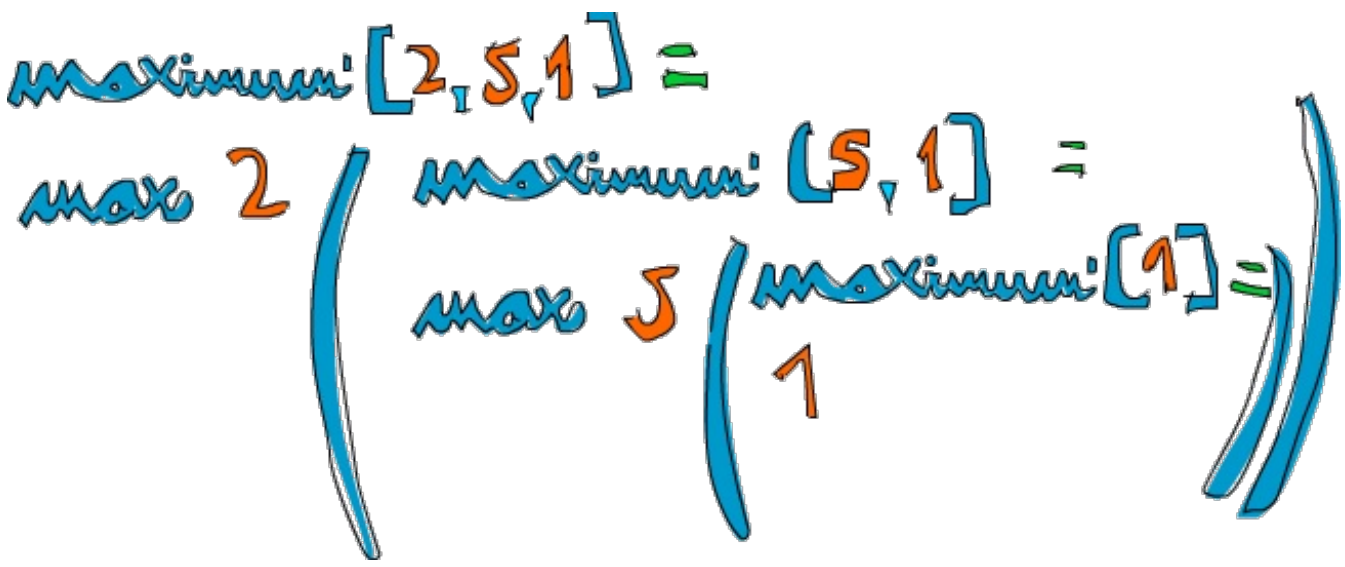
Maintenant, le troisième motif est là où se passe l'action. On utilise du filtrage par motif pour séparer la tête et la queue de la liste. C'est un idiome usuel pour faire de la récursivité sur des listes, donc habituez-vous. On utilise une liaison *where* pour définir `maxTail` comme le maximum du reste de la liste. Ensuite, on vérifie si la tête est plus grande que le maximum du reste de la liste. Si c'est le cas, la tête est retournée. Sinon, le maximum du reste de la liste est retourné.

Prenons par exemple une liste de nombres et voyons comment cela marche : `[2, 5, 1]`. Si on appelle `maximum'` dessus, les deux premiers motifs ne vont pas correspondre. Le troisième marchera, et coupera le `2` du `[5, 1]`. La clause *where* veut connaître le maximum de `[5, 1]`, donc on suit cette route. Elle la compare avec succès au troisième motif à nouveau, et `[5, 1]` est coupé en `5` et `[1]`. À nouveau, la clause *where* veut connaître le maximum de `[1]`. C'est un cas de base, on retourne donc `1`. Enfin ! En remontant d'une étape, `5` est comparé au maximum de `[1]` (qui est `1`), et retourne `5`. On sait à présent que le maximum de `[5, 1]` est `5`. Une étape plus haut, nous comparions `2` au maximum de `[5, 1]`, qui est `5`, et on choisit donc `5`.

Une manière encore plus claire d'écrire cette fonction est d'utiliser `max`. Si vous vous souvenez, `max` est une fonction qui prend deux nombres et retourne le plus grand d'entre eux. Ici, nous pourrions réécrire `maximum'` en utilisant `max` :

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

Comme c'est élégant ! Dans le principe, le maximum d'une liste est le max de son premier élément et du maximum de la queue.



Un peu plus de fonctions récursives

Maintenant que l'on sait penser en termes de récursivité, essayons d'implémenter quelques fonctions récursives. D'abord, nous allons implémenter `replicate`. `replicate` prend un `Int` et un élément, et retourne une liste qui a plusieurs fois cet élément. Par exemple, `replicate 3 5` retourne `[5, 5, 5]`. Réfléchissons au cas de base. À mon avis, le cas de base correspond à des valeurs inférieures ou égales à 0. Si l'on essaie de répliquer quelque chose zéro fois, on devrai renvoyer une liste vide. Et pareil pour des nombres négatifs, sinon ça ne veut pas dire grand chose.

```
replicate' :: (Num i, Ord i) => i -> a -> [a]
replicate' n x
    | n <= 0    = []
    | otherwise = x:replicate' (n-1) x
```

Nous avons utilisé des gardes ici plutôt que des motifs car on teste une condition booléenne. Si `n` est plus petit que 0, retourner la liste vide. Sinon, retourner une liste qui a `x` comme premier élément, et ensuite `x` répliqué n–1 fois pour la queue. À force, la partie en `(n-1)` va amener notre fonction à atteindre son cas de base.

Note : `Num` n'est pas une sous-classe d'`Ord`. Cela signifie qu'un nombre n'a pas forcément besoin d'être ordonnable. C'est pourquoi on doit spécifier à la fois `Num` et `Ord` en contraintes de classe pour pouvoir faire à la fois des additions, des soustractions et des comparaisons.

Maintenant, implémentons `take`. Elle prend un certain nombre d'éléments d'une liste. Par exemple, `take 3 [5, 4, 3, 2, 1]` retourne `[5, 4, 3]`. Si on essaie de prendre 0 ou moins éléments d'une liste, on récupère une liste vide. Également, si l'ont essaie de prendre quoi que ce soit d'une liste vide, on récupère une liste vide. Remarquez que ce sont nos deux cas de base. Écrivons tout cela :

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

Le premier motif spécifie qu'essayer de prendre 0 ou moins éléments retourne une liste vide. Remarquez qu'on utilise `_` pour filtrer la liste parce que nous ne nous intéressons pas vraiment à sa valeur dans ce cas. Remarquez aussi qu'on utilise des gardes, mais pas `otherwise`. Cela veut dire que si `n` s'avère être supérieur à 0, le filtrage passe au motif suivant. Le deuxième motif indique que si l'on essaie de prendre quoi que ce soit dans une liste vide, on récupère une liste vide. Le troisième motif coupe la liste en tête et queue. Puis, on dit que prendre `n` éléments d'une liste équivaut à une liste avec `x` en tête et une liste qui prend `n-1` éléments dans le reste pour queue. Essayez de prendre une feuille de papier pour écrire les étapes de l'évaluation de `take 3 [4, 3, 2, 1]`.

`reverse` renverse une liste. Pensez au cas de base. Quel est-il ? Allons… c'est la liste vide ! Une liste

vide renversée est la liste vide elle-même. Ok. Et le reste ? Eh bien, on peut dire que si l'on coupe une liste en tête et queue, la liste renversée est égale à la queue renversée, avec la tête à la fin.

```haskell
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Et voilà !

Puisqu'Haskell supporte des listes infinies, notre récursivité n'a pas vraiment besoin de cas de base. Si elle n'en a pas, elle va soit continuer à s'agiter sur un calcul à l'infini, ou produire une structure de donnée infinie, comme une liste infinie. La chose bien à propos des listes infinies, c'est qu'on peut les couper où on veut. `repeat` prend un élément et retourne une liste infinie qui a cet élément uniquement. Une implémentation récursive est très simple, regardez.

```haskell
repeat' :: a -> [a]
repeat' x = x:repeat' x
```

Appeler `repeat 3` nous donnera une liste qui commence avec un `3`, et a une infinité de `3` en queue. Appeler `repeat 3` s'évaluera à `3:repeat 3`, puis à `3:(3:repeat 3)`, puis à `3:(3:(3:repeat 3))`, etc. `repeat 3` ne terminera jamais son évaluation, alors que `take 5 (repeat 3)` nous donnera une liste de cinq 3. C'est comme si on avait fait `replicate 5 3`.

`zip` prend deux listes, et les zippe ensemble. `zip [1, 2, 3] [2, 3]` retourne `[(1, 2), (2, 3)]`, parce qu'elle tronque la plus grande liste pour avoir la même taille que l'autre. Et si l'on zippe quelque chose à la liste vide ? Eh bien, on obtient la liste vide. Voilà notre cas de base. Cependant, `zip` prend deux listes en paramètres, donc il y a en fait deux cas de base.

```haskell
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

D'abord, les deux premiers motifs disent que si une des deux listes est vide, on obtient une liste vide. Le troisième dit que deux listes zippées sont égales à la paire de leur tête, suivie de leurs queues zippées. Zipper `[1, 2, 3]` et `['a', 'b']` va finalement essayer de zipper `[3]` et `[]`. Le cas de base arrive et le résultat est donc `(1, 'a'):(2, 'b'):[]`, qui est exactement la même chose que `[(1, 'a'), (2, 'b')]`.

Implémentons encore une fonction de la bibliothèque standard - `elem`. Elle prend un élément et une liste et vérifie si cet élément appartient à la liste. La condition de base, comme souvent, est la liste vide. On sait qu'une liste vide ne contient aucun élément, donc elle ne contient certainement pas les droïdes que vous recherchez.

```haskell
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = a `elem'` xs
```

Plutôt simple et attendu. Si la tête n'est pas l'élément recherché, on cherche dans la queue. Si on arrive à une liste vide, le résultat est `False`.

Vite, triez !

Soit une liste d'éléments qui peuvent être triés. Leur type est une instance de la classe `Ord`. À présent, on souhaite les trier ! Il existe un algorithme très cool pour faire ça, appelé quicksort. C'est une façon très astucieuse de trier des éléments. Alors qu'elle prend facilement 10 lignes à implémenter dans des langages impératifs, l'implémentation Haskell est bien plus courte et élégante. Quicksort est devenu une sorte d'icône d'Haskell. Ainsi, implémentons le, bien qu'implémenter quicksort en Haskell est assez ringard vu que tout le monde le fait déjà pour montrer comme Haskell est élégant.

Donc, la signature de type sera `quicksort :: (Ord a) => [a] -> [a]`. Pas de surprise ici. Le cas de base ? La liste vide, bien sûr. Une liste vide triée est une liste vide. Maintenant, voilà l'algorithme : **une liste triée est une liste pour laquelle on a pris tous les éléments plus petits que la tête, qu'on les a triés, et placés à l'avant, puis on a mis la tête, et à la suite on a placé tous les éléments plus grands que la tête, après les avoir aussi triés"**. Remarquez comme on a dit deux fois *trier* dans la définition, donc il y aura probablement deux appels récursifs ! Remarquez aussi qu'on a utilisé le verbe être pour définir l'algorithme, et pas une suite de *faire ceci, puis faire cela, ensuite faire ceci…* C'est la beauté de la programmation fonctionnelle ! Comment allons nous filtrer d'une liste les éléments plus petits que sa tête, et ceux plus grands ? Avec des listes en compréhension. Bon, plongeons.

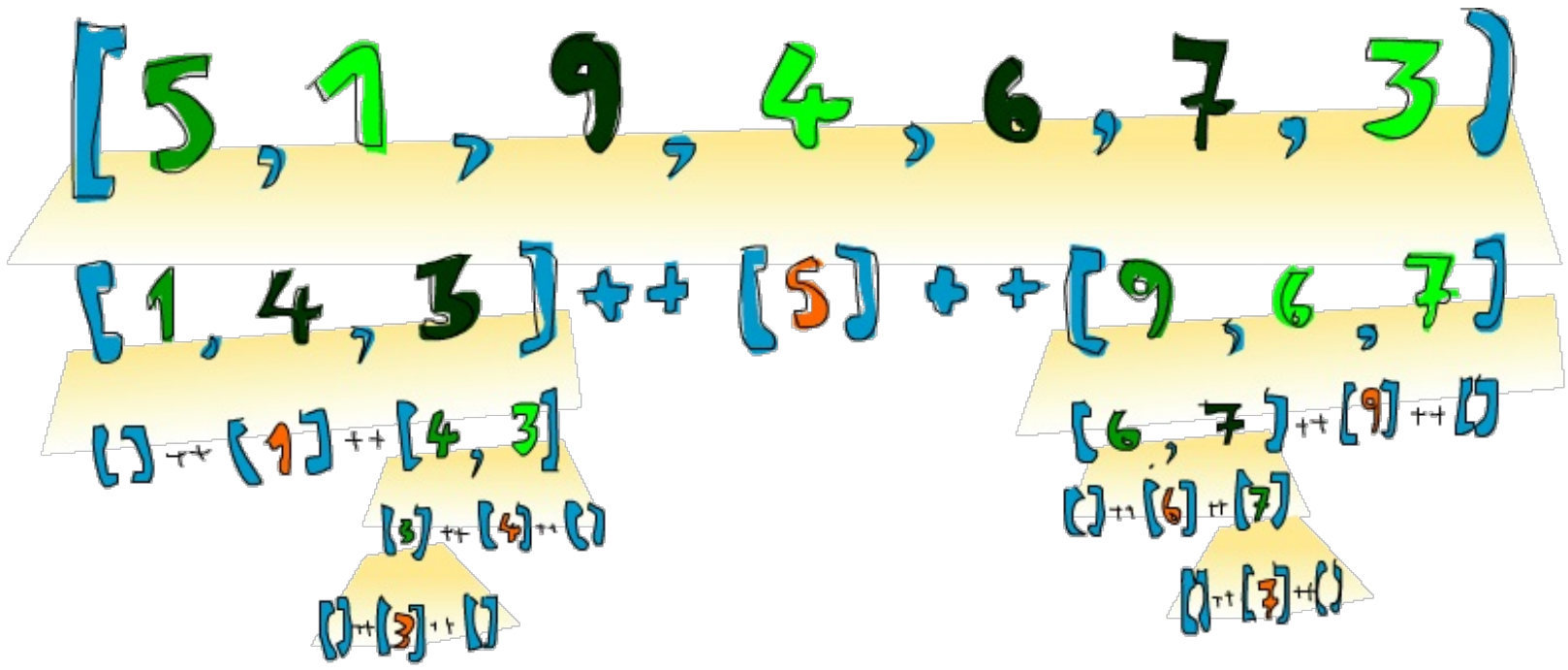
```haskell
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
```

```
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Lançons-là pour voir si elle a l'air de marcher.

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "the quick brown fox jumps over the lazy dog"
"        abcdeeefghhijklmnoooopqrrsttuuvwxyz"
```

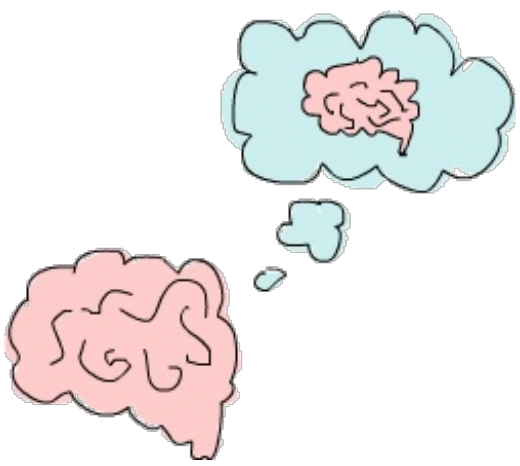
Booyah ! Voilà de quoi je parle ! Si on a, disons `[5, 1, 9, 4, 6, 7, 3]` et qu'on veut la trier, cet algorithme va d'abord prendre sa tête, ici `5`, et la mettre au milieu de deux listes respectivement plus petite et plus grande que la tête. À un moment, on a donc `[1, 4, 3] ++ [5] ++ [9, 6, 7]`. On sait qu'une fois la liste complètement triée, le `5` n'aura pas changé de place, car les 3 nombres à sa gauche sont plus petits que lui, et les 3 à sa droite sont plus grands. Maintenant, si l'on trie récursivement `[1, 4, 3]` et `[9, 6, 7]`, la liste entière est triée ! On les trie à l'aide de la même fonction. Finalement, on va tellement réduire les listes qu'il ne restera que des listes vides, qui sont triées par définition, puisqu'elles sont vides. Voilà une illustration :



Un élément en place et qui ne bougera plus est représenté en **orange**. Si vous les lisez de gauche à droite, vous verrez la liste triée. Bien qu'on ait choisi de comparer tous les éléments aux têtes des listes, on aurait tout à fait pu prendre n'importe quel élément et le comparer à tous les autres. Dans quicksort, l'élément choisi pour être comparé aux autres est appelé le pivot. Ici, les pivots sont représentés en **vert**. Nous avons choisi la tête car il est facile de l'obtenir par filtrage par motif. Les éléments plus petits que le pivot sont en **vert clair** et ceux plus grands que le pivot en **vert foncé**. Le gradient jaune-orangé représente l'application de quicksort.

Penser récursif

Nous avons fait pas mal de récursivité et, comme vous l'avez probablement remarqué, il y a un motif sous-jacent. Généralement, vous définissez des cas de base, et ensuite une fonction qui fait quelque chose avec certains éléments et rappelle cette fonction sur le reste. Peu importe que ce soit une liste, un arbre ou une autre structure de données. Une somme est le premier élément, plus la somme du reste de la liste. Un produit est le premier élément, multiplié par le produit du reste de la liste. La longueur est 1, plus la longueur du reste de la liste. Et cætera, et cætera…



Bien sûr, il y a aussi les cas de base. Généralement, le cas de base est un scénario pour lequel un appel récursif n'a plus de sens. Pour des listes, le plus souvent c'est la liste vide. Si vous traitez des arbres, ce sera généralement une feuille, i.e. un nœud qui n'a pas d'enfants.

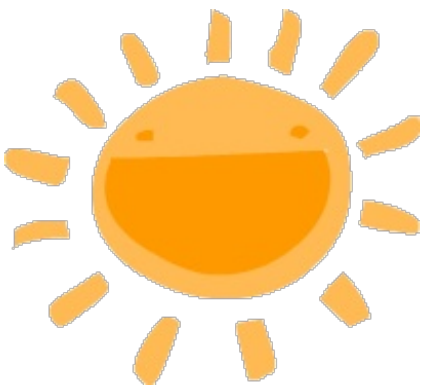
C'est la même chose pour traiter des nombres récursivement. Généralement, on a un nombre et une fonction appliquée à une modification de ce nombre. La factorielle vue plus tôt était le produit d'un nombre et de la factorielle de ce nombre moins un. Une telle application récursive n'a pas de sens pour zéro, parce que les factorielles sont seulement définies pour les entiers positifs. Souvent, le cas de base s'avère être un élément neutre. L'élément neutre de la multiplication est 1 car si l'on multiplie un nombre par 1, on récupère le même nombre. Lorsqu'on somme des listes, on définit la somme de la liste vide comme 0 et 0 est l'élément neutre pour l'addition. Dans quicksort, le cas de base est la liste vide et l'élément neutre est la liste vide également, car si vous rajoutez une liste vide à une autre liste, vous récupérez cette dernière.

Donc, lorsque vous essayez de penser à une solution récursive pour un problème à résoudre, essayez de vous demander quand est-ce qu'une solution récursive ne conviendra pas et d'utiliser cela comme cas de base. Pensez aux éléments neutres et demandez-vous si vous aurez à déconstruire les paramètres de la fonction (par exemple, les listes sont généralement séparées en tête et queue par un filtrage par motif) et sur quelle partie vous effectuerez un appel récursif.

← Syntaxe des fonctions | Table des matières | Fonctions d'ordre supérieur →

Fonctions d’ordre supérieur

← Récursivité | Table des matières | Modules →



Les fonctions Haskell peuvent prendre d’autres fonctions en paramètres, et retourner des fonctions en valeur de retour. Une fonction capable d’une de ces deux choses est dite d’odre supérieur. Les fonctions d’ordre supérieur ne sont pas qu’une partie de l’expérience Haskell, elles sont l’expérience Haskell. Elles s’avèrent indispensable pour définir ce que les choses *sont* plutôt que des étapes qui changent un état et bouclent. Elles sont un moyen très puissant de résoudre des problèmes et de réflexion sur les programmes.

Fonctions curryfiées

En Haskell, toutes les fonctions ne prennent en fait qu’un unique paramètre. Comment avons-nous pu définir des fonctions qui en prenaient plus d’un alors ? Eh bien, grâce à un détail astucieux ! Toutes nos fonctions qui acceptaient *plusieurs paramètres* jusqu’à présent étaient des **fonctions curryfiées**. Qu’est-ce que cela signifie ? Vous comprendrez mieux avec un exemple. Prenons cette bonne vieille fonction `max`. Elle a l’air de prendre deux paramètres, et de retourner le plus grand des deux. Faire `max 4 5` crée une fonction qui prend un paramètre, et retourne soit `4`, soit ce paramètre. On applique alors cette fonction à la valeur `5` pour produire le résultat attendu. On ne dirait pas comme ça, mais c’est en fait assez cool. Les deux appels suivants sont ainsi équivalents :

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

Mettre un espace entre deux choses consiste à **appliquer une fonction**. L’espace est en quelque sorte un opérateur, et il a la plus grande précédence. Examinons le type de `max`. C’est `max :: (Ord a) => a -> a -> a`. On peut aussi écrire ceci : `max :: (Ord a) => a -> (a -> a)`. Et le lire ainsi : `max` prend un `a` et retourne (c’est le premier `->`)) une fonction qui prend un `a` et retourne un `a`. C’est pourquoi le type de retour et les paramètres d’une fonction sont séparés par des flèches.

En quoi cela nous aide-t-il ? Pour faire simple, si on appelle une fonction avec trop peu de paramètres, on obtient une fonction **appliquée partiellement**, c’est à dire une fonction qui prend autant de paramètres qu’on en a oubliés. Utiliser l’application partielle (appeler une fonction avec trop peu de paramètres) est un moyen grâcieux de créer des fonctions à la volée et de les passer à d’autres fonctions pour qu’elle les complète avec d’autres données.

Regardons cette fonction violemment simple :

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

Que se passe-t-il vraiment quand on fait `multThree 3 5 9` ou `((multThree 3) 5) 9` ? D’abord, `3` est appliqué à `multThree`, car ils sont séparés par un espace. Cela crée une fonction qui prend un paramètre et retourne une fonction. Ensuite, `5` est appliqué à cette nouvelle fonction, ce qui crée une fonction qui prend un paramètre et le multiplie par 15. `9` est appliqué à cette dernière et le résultat est un truc comme 135. Rappelez-vous que le type de cette fonction peut être écrit comme `multThree :: (Num a) => a -> (a -> (a -> a))`. La chose avant `->` est l’unique paramètre de la fonction, et la chose après est son unique type retourné. Donc, notre fonction prend un `a` et retourne une fonction qui a pour type `(Num a) => a -> (a -> a)`. De façon similaire, cette fonction prend un `a` et retourne une fonction qui a pour type `(Num a) => a -> a`. Et cette fonction, prend un `a` et retourne un `a`. Regardez ça :

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
ghci> let multWithEighteen = multTwoWithNine 2
ghci> multWithEighteen 10
180
```

En appelant des fonctions avec trop peu de paramètres, en quelque sorte, on crée de nouvelles fonctions à la volée. Comment créer une fonction qui prend un nombre, et le compare à `100` ? Comme ça :

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred x = compare 100 x
```

Si on l'appelle avec `99`, elle retourne `GT`. Simple. Remarquez-vous le `x` tout à droite des deux côtés de l'équation ? Réfléchissons à présent à ce que `compare 100` retourne. Cela retourne une fonction qui prend un nombre, et le compare à 100. Ouah ! Est-ce que ce n'était pas la fonction qu'on voulait ? On devrait réécrire cela :

```
reWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred = compare 100
```

La déclaration de type est inchangée, car `compare 100` retourne bien une fonction. `compare` a pour type `(Ord a) => a -> (a -> Ordering)` et l'appeler avec la valeur `100` retourne un `(Num a, Ord a) => a -> Ordering`. Une classe de contrainte additionnelle s'est insinuée ici parce que `100` est un élément de la classe `Num`.

Yo ! Soyez certain de bien comprendre les fonctions curryfiées et l'application partielle, c'est très important !

Les fonctions infixes peuvent aussi être appliquées partiellement à l'aide de sections. Sectionner une fonction infixe revient à l'entourer de parenthèses et à lui fournir un paramètre sur l'un de ses côtés. Cela crée une fonction qui attend l'autre paramètre et l'applique du côté où il lui manquait une opérande. Une fonction insultante de trivialité :

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
```

Appeler, disons, `divideByTen 200` est équivalent à faire `200 / 10`, ce qui est aussi équivalent à `(/10) 200`. Maintenant, une fonction qui vérifie si un caractère est en majuscule :

```
isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])
```

La seule chose spéciale à propos des sections, c'est avec `-`. Par définition des sections, (–4) devrait être la fonction qui attend un nombre, et lui soustrait 4. Cependant, par habitude, `(-4)` signifie la valeur moins quatre. Pour créer une fonction qui soustrait 4 du nombre en paramètre, appliquez plutôt partiellement la fonction `subtract` ainsi : `(subtract 4)`.

Que se passe-t-il si on tape `multThree 3 4` directement dans GHCi sans la lier avec un *let* ou la passer à une autre fonction ?

```
ghci> multThree 3 4
<interactive>:1:0:
    No instance for (Show (t -> t))
      arising from a use of `print' at <interactive>:1:0-12
    Possible fix: add an instance declaration for (Show (t -> t))
    In the expression: print it
    In a 'do' expression: print it
```

GHCi nous dit que l'expression produite est une fonction de type `a -> a` mais qu'il ne sait pas afficher cela à l'écran. Les fonctions ne sont pas des instances de la classe `Show`, donc elles n'ont pas de représentation littérale. Quand on entre `1 + 1` dans GHCi, il calcule d'abord la valeur `2` puis appelle `show` sur `2` pour obtenir une représentation textuelle de ce nombre. La représentation de `2` est `"2"`, et c'est ce qu'il nous affiche.

À l'ordre du jour : de l'ordre supérieur

Les fonctions peuvent prendre des fonctions en paramètres et retourner des fonctions. Pour illustrer cela, nous allons créer une fonction qui prend une fonction en paramètre, et l'applique deux fois à quelque chose !

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

D'abord, remarquez la déclaration de type. Avant, on n'avait pas besoin de parenthèses, parce que `->` est naturellement associatif à droite. Ici, au contraire, elles sont obligatoires. Elles indiquent que le premier paramètre est une fonction qui prend quelque chose et retourne une chose du même type. Le second paramètre est quelque chose également de ce type, et la valeur retournée est elle aussi de ce type. On pourrait lire cette déclaration de type de manière curryfiée, mais pour ne pas se prendre trop la tête, disons juste qu'elle prend deux paramètres, et retourne une chose. Le premier paramètre est une fonction (de type `a -> a`) et le second est du même type `a`. La fonction peut très bien être de type `Int -> Int` ou `String -> String` ou quoi que ce soit. Mais alors, le second paramètre doit être du type correspondant.

Note : à partir de maintenant, nous dirons qu'une fonction prend plusieurs paramètres malgré le fait qu'elle ne prend en réalité qu'un paramètre et retourne une fonction appliquée partiellement jusqu'à finalement arriver à une valeur solide. Donc, pour simplifier, on dira que `a -> a -> a` prend deux paramètres, bien que l'on sache ce qui se trame en réalité sous la couverture.

Le corps de la fonction est plutôt simple. On utilise le paramètre `f` comme une fonction, on applique cette fonction à `x` en les séparant par un espace, puis on applique `f` au résultat une nouvelle fois. Jouons un peu avec la fonction :

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
ghci> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
ghci> applyTwice (multThree 2 2) 9
144
ghci> applyTwice (3:) [1]
[3,3,1]
```

L'application partielle de fonction est évidemment géniale et très utile. Si notre fonction a besoin d'une fonction qui prend un paramètre, on peut lui passer une autre fonction, qu'on aura partiellement appliquée jusqu'à ce qu'il ne lui manque plus qu'un paramètre.

Maintenant, nous allons profiter de la programmation d'ordre supérieur pour implémenter une fonction très utile de la bibliothèque standard. Elle s'appelle `zipWith`. Elle prend une fonction, et deux listes, et zippe les deux listes à l'aide de la fonction, en l'appliquant aux éléments correspondants. Voici une implémentation :

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

Regardez la déclaration de type. Le premier paramètre est une fonction qui attend deux choses pour en produire une troisième. Elles n'ont pas à avoir le même type, mais elles le peuvent tout de même. Les deuxième et troisième paramètres sont des listes. Le résultat est aussi une liste. La première liste est une liste de `a`, parce que la fonction de jointure attend un `a` en premier argument. La deuxième doit être une liste de `b` parce que la fonction de jointure attend un `b` ensuite. Le résultat est une liste de `c`. Si la déclaration de type d'une fonction dit qu'elle accepte un `a -> b -> c`, alors elle peut accepter une fonction de type `a -> a -> a`, mais pas l'inverse ! Rappelez-vous que quand vous écrivez des fonctions, notamment d'ordre supérieur, et que vous n'êtes pas certain du type, vous pouvez juste omettre la déclaration de type et vérifier ce qu'Haskell a inféré avec `:t`.

L'action dans la fonction est assez proche du `zip` normal. Les conditions de base sont les mêmes, seulement il y a un argument de plus, la fonction de jointure, mais puisqu'on ne va pas s'en servir ici, on met juste un `_`. Le corps de la fonction pour le dernier motif est également similaire à `zip`, seulement à la place de faire `(x, y)`, elle fait `f x y`. Une même fonction d'ordre supérieur peut être utilisée pour une multitude de différentes tâches si elle est assez générale. Voici une petite démonstration de toutes les différentes choses que `zipWith'` peut faire :

```
ghci> zipWith' (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith' (++) ["foo ", "bar ", "baz "] ["fighters", "hoppers", "aldrin"]
["foo fighters","bar hoppers","baz aldrin"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
ghci> zipWith' (zipWith' (*)) [[1,2,3],[3,5,6],[2,3,4]] [[3,2,2],[3,4,5],[5,4,3]]
[[3,4,6],[9,20,30],[10,12,12]]
```

Comme vous le voyez, une même fonction d'ordre supérieur peut être utilisée très flexiblement à plusieurs usages. La programmation impérative utilise généralement des trucs comme des boucles *for*, des boucles *while*, des affectations de variables, des vérifications de l'état courant, etc. pour arriver à un comportement, puis l'encapsule dans une interface, comme une fonction. La programmation fonctionnelle utilise des fonctions d'ordre supérieur pour abstraire des motifs récurrents, comme examiner deux listes paire à paire et faire quelque chose avec ces paires, ou récupérer un ensemble de solutions et éliminer celles qui ne vous intéressent pas.

Nous allons encore implémenter une fonction de la bibliothèque standard, `flip`. Elle prend une fonction et retourne une fonction qui est comme la fonction initiale, mais dont les deux premiers arguments ont échangé leur place. On peut l'implémenter ainsi :

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
```

```
    where g x y = f y x
```

En lisant la déclaration de type, on voit qu'elle prend une fonction qui prend un `a` puis un `b` et retourne une fonction qui prend un `b` puis un `a`. Mais puisque les fonctions sont curryfiées par défaut, la deuxième paire de parenthèses ne sert en fait à rien, vu que `->` est associatif à droite par défaut.
`(a -> b -> c) -> (b -> a -> c)` est la même chose que `(a -> b -> c) -> (b -> (a -> c))`, qui est aussi la même chose que `(a -> b -> c) -> b -> a -> c`. On écrit que `g x y = f y x`. Si c'est le cas, alors `f y x = g x y` n'est-ce pas ? Avec cela en tête, on peut définir la fonction encore plus simplement.

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f y x = f x y
```

Ici, on profite du fait que les fonctions soient curryfiées. Quand on appelle `flip' f` sans les paramètres `y` et `x`, elle retourne une `f` qui prend ces deux paramètres et appelle l'originale avec les arguments dans l'ordre inverse. Bien que les fonctions retournées ainsi sont généralement passées à d'autres fonctions, on peut se servir de la curryfication lorsqu'on écrit des fonctions d'ordre supérieur en réfléchissant à ce que leur résultat serait si elles étaient appliquées entièrement.

```
ghci> flip' zip [1,2,3,4,5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
ghci> zipWith (flip' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
```

Maps et filtres

`map` prend une fonction et une liste, et applique la fonction à tous les éléments de la liste, résultant en une nouvelle liste. Regardons sa signature et sa définition.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

La signature dit qu'elle prend une fonction de `a` vers `b`, une liste de `a`, et retourne une liste de `b`. Il est intéressant de voir que rien qu'en regardant la signature d'une fonction, vous pouvez parfois dire exactement ce qu'elle fait. `map` est une de ces fonctions d'ordre supérieur extrèmement utile qui peut être utilisée de milliers de façons différentes. La voici en action :

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

Vous avez probablement remarqué que chacun de ces exemples aurait pu être réalisé à l'aide de listes en compréhension. `map (+3) [1, 5, 3, 1, 6]` est équivalent à `[x+3 | x <- [1, 5, 3, 1, 6]]`. Cependant, utiliser `map` est bien plus lisible dans certains cas où vous ne faites qu'appliquer une fonction à chaque élément de la liste, surtout quand vous mappez un map auquel cas les crochets s'entassent de manière déplaisante.

`filter` est une fonction qui prend un prédicat (un prédicat est une fonction qui dit si quelque chose est vrai ou faux, une fonction qui retourne une valeur booléenne) et une liste, et retourne la liste des éléments qui satisfont le prédicat. La signature et l'implémentation :

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

Plutôt simple. Si `p x` vaut `True`, l'élément est inclus dans la nouvelle liste. Sinon, il reste dehors. Quelques exemples d'utilisation :

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
```

```
[[1,2,3],[3,4,5],[2,2]]
ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeRent"
"uagameasadifeent"
ghci> filter (`elem` ['A'..'Z']) "i lauGh At You BecAuse u r aLL the Same"
"GAYBALLS"
```

Tout ceci pourrait aussi être fait à base de listes en compréhension avec des prédicats. Il n'y a pas de règle privilégiant l'utilisation de `map` et `filter` à celle des listes en compréhension, vous devez juste décider de ce qui est plus lisible en fonction du code et du contexte. L'équivalent pour `filter` de l'application de plusieurs prédicats dans une liste en compréhension est soit en filtrant plusieurs fois d'affilée, soit en fusionnant des prédicats à l'aide de `&&`.

Vous souvenez-vous de la fonction quicksort du chapitre précédent ? Nous avions utilisé des listes en compréhension pour filtrer les éléments plus petits que le pivot. On peut faire de même avec `filter` :

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort (filter (<=x) xs)
        biggerSorted = quicksort (filter (>x) xs)
    in  smallerSorted ++ [x] ++ biggerSorted
```

Mapper et filtrer sont le marteau et le clou de la boîte à outils de tout programmeur fonctionnel. Bon, il importe peut que vous utilisiez plutôt `map` et `filter` que des listes en compréhension. Souvenez-vous comment nous avions résolu le problème de trouver les triangles rectangles avec un certain périmètre. En programmation impérative, nous aurions imbriqué trois boucles, dans lequelles nous aurions testé si la combinaison courante correspondait à un triangle rectangle et avait le bon périmètre, auquel cas nous en aurions fait quelque chose comme l'afficher à l'écran. En programmation fonctionnelle, ceci est effectué par mappage et filtrage. Vous créez une fonction qui prend une valeur et produit un résultat. Vous mappez cette fonction sur une liste de valeurs, puis vous filtrez la liste résultante pour obtenir les résultats qui satisfont votre recherche. Et grâce à la paresse d'Haskell, même si vous mappez et filtrez sur une liste plusieurs fois, la liste ne sera traversée qu'une seule fois.

Trouvons le **plus grand entier inférieur à 100 000 qui soit divisible par 3 829**. Pour cela, filtrons un ensemble de solutions pour lesquelles on sait que le résultat s'y trouve.

```
largestDivisible :: (Integral a) => a
largestDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3829 == 0
```

On crée d'abord une liste de tous les entiers inférieurs à 100 000 en décroissant. Puis, on la filtre par un prédicat, et puisque les nombres sont en ordre décroissant, le plus grand d'entre eux sera simplement le premier élément de la liste filtrée. On n'a même pas eu besoin d'une liste finie pour démarrer. Encore un signe de la paresse en action. Puisque l'on n'utilise que la tête de la liste filtrée, peu importe qu'elle soit finie ou infinie. L'évaluation s'arrête dès que la première solution est trouvée.

Maintenant, trouvons la **somme de tous les carrés impairs inférieurs à 10 000**. Mais d'abord, puisqu'on va l'utiliser dans la solution, introduisons la fonction `takeWhile`. Elle prend un prédicat et une liste, et parcourt cette liste depuis le début en retournant tous les éléments tant que le prédicat reste vrai. Dès qu'un élément ne satisfait plus le prédicat, elle s'arrête. Si l'on voulait le premier mot d'une chaîne comme `"elephants know how to party"`, on pourrait faire `takeWhile (/=' ') "elephants know how to party"` et cela retournerait `"elephants"`. Ok. La somme des carrés impairs inférieurs à dix mille. D'abord, on va commencer par mapper `(^2)` sur la liste infinie `[1..]`. Ensuite, on va filtrer pour garder les nombres impairs. Puis, on prendra des éléments de cette liste tant qu'ils restent inférieurs à 10 000. Enfin, on va sommer cette liste. On n'a même pas besoin d'une fonction pour ça, une ligne dans GHCi suffit :

```
ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

Génial ! On commence avec une donnée initiale (la liste infinie de tous les entiers naturels) et on mappe par dessus, puis on filtre et on coupe jusqu'à avoir ce que l'on désire, et on somme le tout. On aurait pu écrire ceci à l'aide de listes en compréhension :

```
ghci> sum (takeWhile (<10000) [n^2 | n <- [1..], odd (n^2)])
166650
```

C'est une question de goût, à vous de choisir la forme que vous préférez. Encore une fois, la paresse d'Haskell rend ceci possible. On peut mapper et filtrer une liste infinie, puisqu'il ne va pas mapper et filtrer directement, il retardera ces actions. C'est seulement quand on demande à Haskell de nous montrer la somme que la fonction `sum` dit à la fonction `takeWhile` qu'elle a besoin de cette liste de nombre. `takeWhile` force le filtrage et le mappage, en demandant des nombres un par un tant qu'il n'en croise pas un plus grand que 10000.

Pour notre prochain problème, on va se confronter aux suites de Collatz. On prend un entier naturel. Si cet entier est pair, on le divise par deux. S'il est impair, on

le multiplie par 3 puis on ajoute 1. On prend ce nombre, et on recommence, ce qui produit un nouveau nombre, et ainsi de suite. On obtient donc une chaîne de nombres. On pense que, quel que soit le nombre de départ, la chaîne termine, avec pour valeur 1. Si on prend 13, on obtient la suite : *13, 40, 20, 10, 5, 16, 8, 4, 2, 1*. 13 fois 3 plus 1 vaut 40. 40 divisé par 2 vaut 20, etc. On voit que la chaîne contient 10 termes.

Maintenant on cherche à savoir : **pour tout nombre de départ compris entre 1 et 100, combien de chaînes ont une longueur supérieure à 15 ?**
D'abord, on va écrire une fonction qui produit ces chaînes :

```
chain :: (Integral a) => a -> [a]
chain 1 = [1]
chain n
    | even n =  n:chain (n `div` 2)
    | odd n  =  n:chain (n*3 + 1)
```

Puisque les chaînes terminent à 1, c'est le cas de base. C'est une fonction récursive assez simple.

```
ghci> chain 10
[10,5,16,8,4,2,1]
ghci> chain 1
[1]
ghci> chain 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

Yay ! Ça a l'air de marcher correctement. À présent, la fonction qui nous donne notre réponse :

```
numLongChains :: Int
numLongChains = length (filter isLong (map chain [1..100]))
    where isLong xs = length xs > 15
```

On mappe la fonction `chain` sur la liste `[1..100]` pour obtenir une liste de toutes les chaînes, qui sont elles-même représentées sous forme de listes. Puis, on les filtre avec un prédicat qui vérifie simplement si la longueur est plus grande que 15. Une fois ceci fait, il ne reste plus qu'à compter le nombre de chaînes dans la liste finale.

Note : Cette fonction a pour type `numLongChains :: Int` parce que `length` a pour type `Int` au lieu de `Num a`, pour des raisons historiques. Si nous voulions retourner `Num a`, on aurait pu utiliser `fromIntegral` sur le nombre retourné.

En utilisant `map`, on peut faire des trucs comme `map (*) [0..]`, par exemple pour illustrer la curryfication et le fait que les fonctions appliquées partiellement sont de vraies valeurs que l'on peut manipuler et placer dans des listes (par contre, on ne peut pas les transformer en chaînes de caractères). Pour l'instant, on a seulement mappé des fonctions qui attendent un paramètre sur des listes, comme `map (*2) [0..]` pour obtenir des listes de type `(Num a) => [a]`, mais on peut aussi faire `map (*) [0..]` sans problème. Ce qui se passe ici, c'est que le nombre dans la liste s'applique à la fonction `*`, qui a pour type `(Num a) => a -> a -> a`. Appliquer une fonction sur un paramètre, alors qu'elle en attend deux, retourne une fonction qui attend encore un paramètre. Si l'on mappe `*` sur `[0..]`, on obtient en retour une liste de fonctions qui attendent chacune un paramètre, donc `(Num a) [a -> a]`. `map (*) [0..]` produit une liste comme celle qu'on obtiendrait en écrivant `[(0*), (1*), (2*), (3*), (4*), (5*), …]`.

```
ghci> let listOfFuns = map (*) [0..]
ghci> (listOfFuns !! 4) 5
20
```

Demander l'élément d'index `4` de notre liste retourne une fonction équivalente à `(4*)`. Puis on applique cette fonction à `5`. Donc c'est comme écrire `(4*) 5` ou juste `4 * 5`.

Lambdas

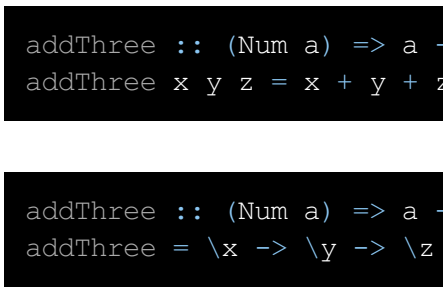
Les lambda expressions sont simplement des fonctions anonymes utilisées lorsqu'on a besoin d'une fonction à usage unique. Normalement, on crée une lambda uniquement pour la passer à une fonction d'ordre supérieur. Pour créer une lambda, on écrit un `\` (parce que ça ressemble à la lettre grecque lambda si vous le fixez assez longtemps des yeux) puis les paramètres, séparés par des espaces. Après cela vient `->` puis le corps de la fonction. On l'entoure généralement de parenthèses, autrement elle s'étend autant que possible vers la droite.

Si vous regardez 10cm plus haut, vous verrez qu'on a utilisé une liaison *where* dans `numLongChains` pour créer la fonction `isLong`, avec pour seul but de passer cette fonction à `filter`. Au lieu de faire ça, on pourrait utiliser une lambda :

```
numLongChains :: Int
```

```
numLongChains = length (filter (\xs -> length xs > 15) (map chain [1..100]))
```

Les lambdas sont des expressions, c'est pourquoi on peut les manipuler comme ça. L'expression `(\xs -> length xs > 15)` retourne une fonction qui nous dit si la longueur de la liste passée en paramètre est plus grande que 15.



Les personnes n'étant pas habituées à la curryfication et à l'application partielle de fonctions utilisent souvent des lambdas là où ils n'en ont en fait pas besoin. Par exemple, les expressions `map (+3) [1, 6, 3, 2]` et `map (\x -> x + 3) [1, 6, 3, 2]` sont équivalentes puisque `(+3)` tout comme `(\x -> x + 3)` renvoient toutes les deux le nombre passé en paramètre plus 3. Il est inutile de vous préciser que créer une lambda dans ce cas est stupide puisque l'application partielle est bien plus lisible.

Comme les autres fonctions, les lambdas peuvent prendre plusieurs paramètres.

```
ghci> zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
```

Et comme les fonctions, vous pouvez filtrer par motif dans les lambdas. La seule différence, c'est que vous ne pouvez pas définir plusieurs motifs pour un paramètre, comme par exemple créer un motif `[]` et un `(x:xs)` pour le même paramètre, et avoir les valeurs parcourir les différents motifs. Si un filtrage par motif échoue dans une lambda, une erreur d'exécution a lieu, donc faites attention en filtrant dans les lambdas.

```
ghci> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

Les lambdas sont habituellement entourées de parenthèses à moins qu'on veuille qu'elles s'étendent le plus possible à droite. Voilà quelque chose d'intéressant : vu la façon dont sont curryfiées les fonctions par défaut, ces deux codes sont équivalents :

```
addThree :: (Num a) => a -> a -> a -> a
addThree x y z = x + y + z
```

```
addThree :: (Num a) => a -> a -> a -> a
addThree = \x -> \y -> \z -> x + y + z
```

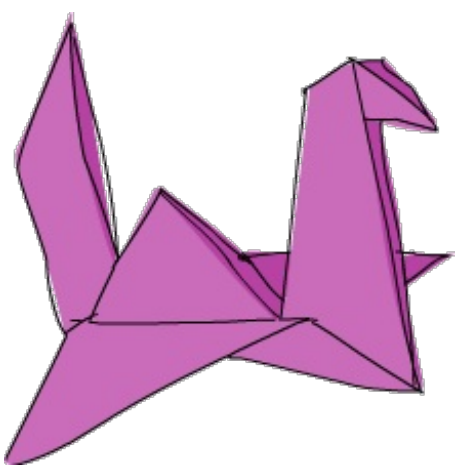
Si on définit une fonction ainsi, il devient évident que la signature doit avoir cette forme. Il y a trois `->` dans le type et dans l'équation. Mais bien sûr, la première façon d'écrire est bien plus lisible, la seconde ne sert qu'à illustrer la curryfication.

Cependant, cette notation s'avère parfois pratique. Je trouve la fonction `flip` plus lisible définie ainsi :

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f = \x y -> f y x
```

Bien que ce soit comme écrire `flip' f x y = f y x`, on fait apparaître clairement que ce sera utilisé pour créer une nouvelle fonction la plupart du temps. L'utilisation la plus classique de `flip` consiste à appeler la fonction avec juste une autre fonction, et de passer le résultat à un `map` ou un `filter`. Utilisez donc les lambdas lorsque vous voulez rendre explicite le fait que cette fonction est généralement appliquée partiellement avant d'être passée à une autre comme paramètre.

Plie mais ne rompt pas



À l'époque où nous faisions de la récursivité, nous avions repéré un thème récurrent dans nos fonctions récursives qui opéraient sur des listes. Nous introduisions un motif `x:xs` et nous faisions quelque chose avec la tête et quelque chose avec la queue. Il s'avère que c'est un motif très commun, donc il existe quelques fonctions très utiles qui l'encapsulent. Ces fonctions sont appelées des *folds* (NDT : des plis). Elles sont un peu comme la fonction `map`, seulement qu'elles réduisent une liste à une simple valeur.

Un *fold* prend une fonction binaire, une valeur de départ (que j'aime appeler l'accumulateur) et une liste à plier. La fonction binaire prend deux paramètres. Elle est appelée avec l'accumulateur en première ou deuxième position, et le premier ou le dernier élément de la liste comme autre paramètre, et produit un nouvel accumulateur. La fonction est appelée à nouveau avec le nouvel accumulateur et la nouvelle extrémité de la queue, et ainsi de suite. Une fois qu'on a traversé toute la liste, seul l'accumulateur reste, c'est la valeur à laquelle on a réduit la liste.

D'abord, regardons la fonction `foldl`, aussi appelée *left fold* (pli à gauche). Elle plie la liste en partant de la gauche. La fonction binaire est appelée avec la valeur de départ de l'accumulateur et la tête de liste. Cela produit un nouvel accumulateur, et la fonction est à nouveau appelée sur cette valeur et le prochain

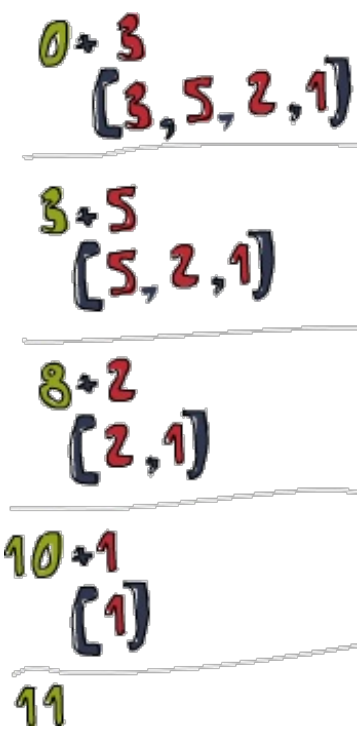
élément de la liste, etc.

Implémentons `sum` à nouveau, mais cette fois, utilisons un *fold* plutôt qu'une récursivité explicite.

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

Un, deux, trois, test :

```
ghci> sum' [3,5,2,1]
11
```



Regardons de près comment ce pli se déroule. `\acc x -> acc + x` est la fonction binaire. `0` est la valeur de départ et `xs` la liste à plier. D'abord, `0` est utilisé pour `acc` et `3` pour `x`. `0 + 3` retourne `3` et devient, pour ainsi dire, le nouvel accumulateur. Ensuite, `3` est utilisé comme accumulateur, et l'élément courant, `5`, résultant en un `8` comme nouvel accumulateur. Encore en avant, `8` est l'accumulateur, `2` la valeur courante, le nouvel accumulateur est donc `10`. Utilisé avec la valeur courante `1`, il produit `11`. Bravo, vous venez d'achever votre premier pli !

Le diagramme professionnel sur la gauche illustre la façon dont le pli se déroule, étape par étape (jour après jour !). Le nombre vert kaki est l'accumulateur. Vous pouvez voir comme la liste est consommée par la gauche par l'accumulateur. Om nom nom nom ! (NDT : "Miam miam miam !") Si on prend en compte le fait que les fonctions sont curryfiées, on peut écrire cette implémentation encore plus rapidement :

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

La lambda `(\acc x -> acc + x)` est équivalente à `(+)`. On peut omettre le `xs` à la fin parce que `foldl (+) 0` retourne une fonction qui attend une liste. En général, lorsque vous avec une fonction `foo a = bar b a`, vous pouvez réécrire `foo = bar b`, grâce à la curryfication.

Bon, implémentons une autre fonction avec un pli à gauche avant de passer aux plis à droite. Je suis sûr que vous savez tous qu'`elem` vérifie si une valeur fait partie d'une liste, donc je ne vais pas vous le rappeler (oups, je viens de le faire !). Implémentons-là avec un pli à gauche.

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
```

Bien, bien, bien, qu'avons-nous là ? La valeur de départ et l'accumulateur sont de type booléen. Le type de l'accumulateur et de l'initialisateur est toujours le même quand on plie. Rappelez-vous en quand vous ne savez plus quoi utiliser comme valeur de départ, ça vous mettra sur la piste. Ici, on commence avec `False`. Il est logique d'utiliser `False` comme valeur initiale. On présume que l'élément n'est pas là, tant qu'on n'a pas de preuve de sa présence. Notez que si l'on appelle `fold` sur une liste vide, on obtient en retour la valeur initiale. Ensuite, on vérifie le premier élément pour savoir si c'est celui que l'on cherche. Si c'est le cas, on passe l'accumulateur à `True`. Sinon, on ne touche pas à l'accumulateur. S'il était `False`, il reste à `False` car on ne vient pas de trouver l'élément. S'il était `True`, on le laisse aussi.

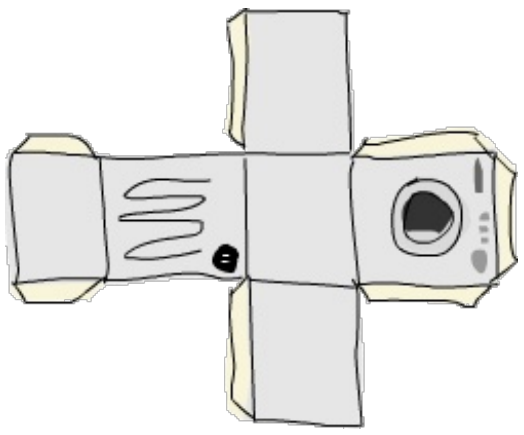
Le pli à droite, `foldr` travaille de manière analogue au pli à gauche, mais l'accumulateur consomme les valeurs en partant de la droite. Aussi, la fonction binaire du pli gauche prend l'accumulateur en premier paramètre, et la valeur courante en second (`\acc x -> …`), celle du pli droit prend la valeur courante en premier et l'accumulateur en second (`\x acc -> …`). C'est assez logique que le pli à droite ait l'accumulateur à droite, vu qu'il plie depuis la droite.

La valeur de l'accumulateur (et donc le résultat) d'un pli peut être de n'importe quel type. Un nombre, un booléen, ou même une liste. Implémentons la fonction `map` à l'aide d'un pli à droite. L'accumulateur sera la liste, on va accumuler les valeurs après mappage, élément par élément. De fait, il est évident que l'élément de départ sera une liste vide.

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

Si l'on mappe `(+3)` à `[1, 2, 3]`, on attaque la liste par la droite. On prend le dernier élément, `3` et on applique la fonction, résultant en un `6`. On le place à l'avant de l'accumulateur, qui était `[]`. `6:[]` est `[6]`, et notre nouvel accumulateur. On applique `(+3)` sur `2`, donnant `5` et on le place devant (`:`) l'accumulateur, l'accumulateur devient `[5, 6]`. On applique `(+3)` à `1` et on le place devant l'accumulateur, ce qui donne pour valeur finale `[4, 5, 6]`.

Bien sûr, nous aurions pu implémenter cette fonction avec un pli gauche aussi. Cela serait `map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs`, mais le problème, c'est que `++` est beaucoup plus coûteux que `:`, donc généralement, on utilise des plis à droite lorsqu'on construit des nouvelles listes à partir d'une liste.



Si vous renversez une liste, vous pouvez faire un pli droit sur une liste comme vous auriez fait un pli gauche sur la liste originale, et vice versa. Parfois, vous n'avez même pas besoin de ça. La fonction `sum` peut être implémentée aussi bien avec un pli à gauche qu'à droite. Une des grosses différences est que les plis à droite fonctionnent sur des listes infinies, alors que les plis à gauche, non ! Pour mettre cela au clair, si vous prenez un endroit d'une liste infinie et que vous vous mettez à plier depuis la droite depuis celui-ci, vous finirez par atteindre le début de la liste. Par contre, si vous vous placez à un endroit d'une liste infinie, et que vous commencez à plier depuis la gauche vers la droite, vous n'atteindrez jamais la fin !

Les plis peuvent être utilisés pour implémenter n'importe quelle fonction qui traverse une liste une fois, élément par élément, et retourne quoi que ce soit basé là-dessus. Si jamais vous voulez parcourir une liste pour retourner quelque chose, vous aurez besoin d'un pli. C'est pourquoi les plis sont, avec les maps et les filtres, un des types les plus utiles en programmation fonctionnelle.

Les fonctions `foldl1` et `foldr1` fonctionnent quasiment comme `foldl` et `foldr`, mais n'ont pas besoin d'une valeur de départ. Elles considèrent que le premier (ou le dernier respectivement) élément de la liste est la valeur de départ, et commencent à plier à partir de l'élément suivant. Avec cela en tête, la fonction `sum` peut être implémentée : `sum = foldl1 (+)`. Puisqu'elles dépendent du fait que les listes qu'elles essaient de plier aient au moins un élément, elles peuvent provoquer des erreurs à l'exécution si on les appelle sur des listes vides. `foldl` et `foldr`, d'un autre côté, fonctionnent bien sur des listes vides. Quand vous faites un pli, pensez donc à la façon dont il se comporte sur une liste vide. Si la fonction n'a aucun sens pour une liste vide, vous pouvez probablement utiliser `foldl1` et `foldr1` pour l'implémenter.

Histoire de vous montrer la puissance des plis, on va implémenter un paquet de fonctions de la bibliothèque standard avec eux :

```
maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

product' :: (Num a) => [a] -> a
product' = foldr1 (*)

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []

head' :: [a] -> a
head' = foldr1 (\x _ -> x)

last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

`head` est tout de même mieux implémenté par filtrage par motif, mais c'était juste pour l'exemple, que l'on peut y arriver avec des plis. Notre fonction `reverse'` est à ce titre plutôt astucieuse. On prend pour valeur initiale une liste vide, et on attaque la liste par la gauche en positionnant les éléments rencontrés à l'avant de notre accumulateur. Au final, on a construit la liste renversée. `\acc x -> x : acc` ressemble assez à la fonction `:`, mais avec ses paramètres dans l'autre sens. C'est pourquoi, on aurait pu écrire reverse comme `foldl (flip (:)) []`.

Une autre façon de se représenter les plis à droite et à gauche consiste à se dire : mettons qu'on a un pli à droite et une fonction binaire `f`, et une valeur initiale `z`. Si l'on plie à droite la liste `[3, 4, 5, 6]`, on fait en gros cela : `f 3 (f 4 (f 5 (f 6 z)))`. `f` est appelé avec le dernier élément de la liste et l'accumulateur, cette valeur est donnée comme accumulateur à la prochaine valeur, etc. Si on prend pour `f` la fonction `+` et pour accumulateur de départ `0`, ça donne `3 + (4 + (5 + (6 + 0)))`. Ou, avec un `+` préfixe, `(+) 3 ((+) 4 ((+) 5 ((+) 6 0)))`. De façon similaire, un pli à gauche avec la fonction binaire `g` et l'accumulateur `z` est équivalent p `g (g (g (g z 3) 4) 5) 6`. Si on utilise `flip (:)` comme fonction binaire, et `[]` comme accumulateur (de manière à renverser la liste), c'est équivalent à `flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6`. Et pour sûr, évaluer cette expression renvoie `[6, 5, 4, 3]`.

`scanl` et `scanr` sont comme `foldl` et `foldr`, mais rapportent tous les résultats intermédiaires de l'accumulateur sous forme d'une liste. Il existe aussi `scanl1` et `scanr1`, analogues à `foldl1` et `foldr1`.

```
ghci> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
ghci> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
ghci> scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1]
[3,4,5,5,7,9,9,9]
ghci> scanl (flip (:)) [] [3,2,1]
[[],[3],[2,3],[1,2,3]]
```

En utilisant un `scanl`, le résultat final sera dans le dernier élément de la liste retournée, alors que pour un `scanr`, il sera en tête.

Les scans sont utilisés pour surveiller la progression d'une fonction implémentable comme un pli. Répondons à cette question : **Combien d'entiers naturels croissants faut-il pour que la somme de leurs racines carrées dépasse 1000 ?** Pour récupérer les racines carrées de tous les entiers naturels, on fait `map sqrt [1..]`. Maintenant, pour obtenir leur somme, on pourrait faire un pli, mais puisqu'on s'intéresse à la progression de la somme, on va plutôt faire un scan. Une fois le scan fait, on compte juste le nombre de sommes qui sont inférieures à 1000. La première somme sera normalement égale à 1. La deuxième, à 1 plus racine de 2. La troisième à cela plus racine de 3. Si X de ces sommes sont inférieures à 1000, alors il faut X+1 éléments pour dépasser 1000.

```
sqrtSums :: Int
sqrtSums = length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..]))) + 1
```

```
ghci> sqrtSums
131
ghci> sum (map sqrt [1..131])
1005.0942035344083
ghci> sum (map sqrt [1..130])
993.6486803921487
```

On utilise `takeWhile` plutôt que `filter` parce que `filter` ne peut pas travailler sur des listes infinies. Bien que nous sachions que cette liste est croissante, `filter` ne le sait pas, donc on utilise `takeWhile` pour arrêter de scanner dès qu'une somme est plus grande que 1000.

Appliquer des fonctions avec $

Bien, maintenant, découvrons la fonction `$`, aussi appelée *application de fonction*. Voyons sa définition :

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

De quoi ? Qu'est-ce que c'est que cet opérateur inutile ? C'est juste une application de fonction ! Eh bien, presque, mais pas complètement ! Alors que l'application de fonction habituelle (avec un espace entre deux choses) a une précédence très élevée, la fonction `$` a la plus faible précédence. Une application de fonction avec un espace est associative à gauche (`f a b c` est équivalent à `((f a) b) c`), alors qu'avec `$` elle est associative à droite.

C'est tout, mais en quoi cela nous aide-t-il ? La plupart du temps, c'est une fonction pratique pour éviter d'écrire des tas de parenthèses. Considérez l'expression `sum (map sqrt [1..130])`. Puisque `$` a une précédence aussi faible, on peut réécrire cette expression `sum $ map sqrt [1..130]`, et éviter de précieuses frappes de clavier ! Quand on rencontre un `$`, la fonction sur sa gauche s'applique à l'expression sur sa droite. Qu'en est-il de `sqrt 3 + 4 + 9` ? Ceci ajoute 9, 4, et la racine de 3. Mais si l'on veut la racine carrée de *3 + 4 + 9*, il faut écrire `sqrt (3 + 4 + 9)`, ou avec `$`, `sqrt $ 3 + 4 + 9`, car `$` a la plus faible précédence de tous les opérateurs. On peut donc voir `$` comme le fait d'écrire une parenthèse ouvrante en lieu et place, et d'aller mettre une parenthèse fermante le plus possible à droite de l'expression.

Et `sum (filter (> 10) (map (*2) [2..10]))` ? Et bien, puisque `$` est associatif à droite, `f (g (z x))` est égal à `f $ g $ z x`. On peut donc réécrire l'expression `sum $ filter (> 10) $ map (*2) [2..10]`.

À part pour se débarasser des parenthèses, `$` implique aussi que l'on peut traiter l'application de fonction comme n'importe quelle fonction. Ainsi, on peut par exemple mapper l'application de fonction sur une liste de fonctions.

```
ghci> map ($ 3) [(4+), (10*), (^2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

Composition de fonctions

En mathématique, la composition de fonctions est définie ainsi : $(f \circ g)(x) = f(g(x))$, ce qui signifie que deux fonctions produisent une nouvelle fonction qui, lorsqu'elle est appelée avec un paramètre, disons x, est équivalent à l'appel de g sur x, puis l'appel de f sur le résultat.

En Haskell, la composition de fonction est plutôt identique. On utilise la fonction `.`, définie ainsi :

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Prêtez attention à la déclaration de type. `f` doit prendre pour paramètre une valeur qui a le même type que la valeur retournée par `g`. Ainsi, la fonction résultante peut prendre un paramètre du même type que celui attendu par `g`, et retourner une valeur du même type que celui retourné par `f`. L'expression `negate . (* 3)` retourne une fonction qui prend un nombre, le multiplie par 3, et retourne la négation.

Une utilisation de la composition de fonctions est de créer des fonctions à la volée pour les passer à d'autre fonctions.

Bien sûr, on peut utiliser des lambdas à cet effet, mais souvent, la composition de fonctions est plus claire et concise. Disons qu'on a une liste de nombres, et qu'on veut tous les rendre négatifs. Un moyen de procéder serait de prendre la valeur absolue de chacun d'entre eux, et de renvoyer son opposé :

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Remarquez la lambda-expression, comme elle ressemble à une composition de fonctions. Avec la composition, on peut écrire :

```
ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Fabuleux ! La composition est associative à droite, donc on peut composer plusieurs fonctions à la fois. L'expression `f (g (z x))` est équivalente à `(f . g . z) x`. Avec ça en tête, on peut transformer :

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

en :

```
ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

Mais qu'en est-il des fonctions à plusieurs paramètres ? Pour les utiliser dans de la composition de fonctions, il faut les avoir partiellement appliquées jusqu'à ce qu'elles ne prennent plus qu'un paramètre. `sum (replicate 5 (max 6.7 8.9))` peut être réécrit `(sum . replicate 5 . max 6.7) 8.9` ou bien `sum . replicate 5 . max 6.7 $ 8.9`. Ce qui se passe ici : une fonction qui prend la même chose que `max 6.7` et applique `replicate 5` à celle-ci est créée. Ensuite, une fonction qui prend ça et applique `sum` est créée. Finalement, cette fonction est appelée avec `8.9`. Vous liriez normalement comme cela : prend `8.9` et applique `max 6.7`, puis applique `replicate 5` à ça, et applique `sum` à ça. Si vous voulez réécrire une expression pleine de parenthèses avec de la composition de fonctions, vous pouvez commencer par mettre le dernier paramètre de la dernière fonction à la suite d'un `$`, et ensuite composer les appels successifs, en les écrivant sans leur dernier paramètre et en les séparant par des points. Si vous avez `replicate 100 (product (map (*3) (zipWith max [1, 2, 3, 4, 5] [4, 5, 6, 7, 8])))`, vous pouvez l'écrire `replicate 100 . product . map (*3) . zipWith max [1, 2, 3, 4, 5] $ [4, 5, 6, 7, 8]`. Si l'expression se terminait par trois parenthèses, la réécriture contiendra probablement trois opérateurs de composition.

Une autre utilisation courante de la composition de fonctions consiste à définir des fonctions avec un style dit sans point (certains disent même sans but !). Prenez par exemple la fonction définie auparavant :

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (+) 0 xs
```

Le `xs` est exposé à gauche comme à droite. À cause de la curryfication, on peut l'omettre des deux côtés, puisqu'appeler `foldl (+) 0` renverra une fonction qui attend une liste. Écrire une fonction comme `sum' = foldl (+) 0` est dit dans le style sans point. Comment écrire ceci en style sans point ?

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

On ne peut pas se débarasser du `x` des deux côtés. Le `x` du côté du corps de la fonction a des parenthèses après lui. `cos (max 50)` ne voudrait rien dire. On ne peut pas prendre le cosinus d'une fonction. On peut cependant exprimer `fn` comme une composition de fonctions :

```
fn = ceiling . negate . tan . cos . max 50
```

Excellent ! Souvent, un style sans point est plus lisible et concis, car il vous fait réfléchir en termes de fonctions composant leur résultat plutôt que de réfléchir à comment est-ce que les données sont transportées d'un endroit à l'autre. Vous pouvez prendre de simple fonctions et utiliser la composition pour les coller et former des fonctions plus complexes. Cependant, souvent, écrire une fonction en style sans point peut être moins lisible parce que la fonction est trop complexe. C'est pourquoi l'on décourage les trop grandes chaînes de composition, bien que je sois coupable d'être fan de composition. Le style préféré consiste à utiliser des liaisons *let* pour donner des noms aux résultats intermédiaires ou découper le problème en sous-problèmes et ensuite mettre tout en place pour que les fonctions aient du sens pour le lecteur, plutôt que de voir une grosse chaîne de compositions.

Dans la section sur les maps et les filtres, nous avions résolu le problème de trouver la somme des carrés impairs inférieurs à 10000. Voici la solution sous la forme d'une fonction.

```
```

```haskell
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

Étant un fan de composition, j'aurais probablement écrit :

```haskell
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]
```

Cependant, s'il était possible que quelqu'un doive lire ce code, j'aurais plutôt écrit :

```haskell
oddSquareSum :: Integer
oddSquareSum =
    let oddSquares = filter odd $ map (^2) [1..]
        belowLimit = takeWhile (<10000) oddSquares
    in  sum belowLimit
```

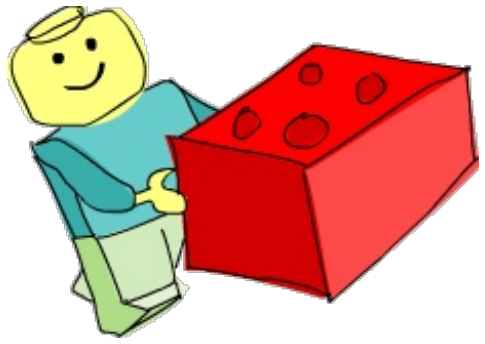
Je ne gagnerai pas de compétition de golf ainsi, mais quelqu'un qui aura à lire ça trouvera certainement cela plus simple à lire qu'une chaîne de compositions.

← Récursivité | Table des matières | Modules →

Modules

⬅ Fonctions d'ordre supérieur | Table des matières | Créer nos propres types et classes de types ➡

Charger des modules



Un module Haskell est une collection de fonctions, types et classes de types en rapport les uns avec les autres. Un programme Haskell est une collection de modules où le module principal charge les autres modules et utilise les fonctions qu'ils définissent. Séparer son code dans plusieurs modules a plusieurs avantages. Si un module est assez générique, les fonctions qu'il exporte peuvent être utilisées dans une multitude de programmes. Si votre propre code est séparé dans des modules assez indépendants (on dit qu'ils ont un couplage faible), vous pourrez les réutiliser plus tard. Cela rend la programmation plus facile à gérer en séparant des entités avec des buts précis.

La bibliothèque standard Haskell est coupée en modules, chacun d'eux contenant des fonctions et des types qui sont reliés et servent un but partagé. Il y a un module de manipulation de listes, un module de programmation concurrente, un module pour les nombres complexes, etc. Toutes les fonctions, types et classes de types que nous avons utilisés jusqu'à présent faisaient partie du module `Prelude`, qui est importé par défaut. Dans ce chapitre, on va explorer quelques modules utiles et les fonctions qu'ils contiennent. Mais d'abord, voyons comment importer un module.

La syntaxe d'import de modules dans un script Haskell est `import <module name>`. Cela doit être fait avant de définir des fonctions, donc généralement, les imports sont faits en début de fichier. Un script peut bien sûr importer plusieurs modules. Écrivez simplement une ligne d'import par module. Importons le module `Data.List`, qui a un paquet de fonctions utiles pour travailler sur des listes, et utilisons une des fonctions que le module exporte afin de savoir combien d'éléments uniques une liste comporte.

```
import Data.List

numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
```

Quand vous écrivez `import Data.List`, toutes les fonctions que `Data.List` exporte deviennent disponibles dans votre espace de nommage global, donc vous pouvez les appeler de n'importe où dans le script. `nub` est une fonction définie dans `Data.List` qui prend une liste et supprime les éléments en double. Composer `length` et `nub` en faisant `length . nub` produit une fonction équivalente à `\xs -> length (nub xs)`.

Vous pouvez aussi importer des fonctions d'un module dans l'espace de nommage global de GHCi. Si vous êtes dans GHCi, et que vous désirez appeler des fonctions exportées par `Data.List`, faites :

```
ghci> :m + Data.List
```

Pour charger plusieurs modules, pas besoin de taper `:m +` plusieurs fois, vous pouvez en charger plusieurs à la fois.

```
ghci> :m + Data.List Data.Map Data.Set
```

Cependant, en chargeant un script qui importe des modules, vous n'avez même pas besoin d'utiliser `:m +` pour accéder à ces modules.

Si vous n'avez besoin que de quelques fonctions d'un module, vous pouvez importer sélectivement juste ces fonctions. Si nous ne voulions que `nub` et `sort` de `Data.List`, on ferait :

```
import Data.List (nub, sort)
```

Vous pouvez aussi choisir d'importer toutes les fonctions d'un module, sauf certaines. C'est souvent utile quand plusieurs modules exportent des fonctions qui ont le même nom et que vous voulez vous débarasser de celles qui ne vous concernent pas. Mettons qu'on ait défini une fonction `nub` et qu'on veuille importer toutes les fonctions de `Data.List` à part `nub` :

```
import Data.List hiding (nub)
```

Un autre moyen de gérer les collisions de noms est d'utiliser les imports qualifiés. Le module `Data.Map`, qui offre une structure de donnée pour trouver des valeurs grâce à une clé, exporte un paquet de fonctions qui ont les mêmes noms que celles du `Prelude`, comme `filter` ou `null`. Donc, quand on importe

`Data.Map` et qu'on appelle `filter`, Haskell ne sait pas de laquelle on parle. Voici comment on résout cela :

```
import qualified Data.Map
```

Cela nous force à écrire `Data.Map.filter` pour désigner la fonction `filter` de `Data.Map`, alors que `filter` tout court correpond à la fonction du `Prelude` qu'on connaît et qu'on aime tous. Mais écrire `Data.Map` devant chaque fonction du module est un peu fastidieux. C'est pourquoi on peut renommer les imports qualifiés :

```
import qualified Data.Map as M
```

Maintenant, pour parler de la fonction `filter` de `Data.Map`, on peut utiliser `M.filter`.

Utilisez cette référence très pratique pour savoir quels modules font partie de la bibliothèque standard. Un bon moyen d'apprendre des choses sur Haskell est de se balader dans cette référence et d'explorer des modules et leurs fonctions. Pour chaque module, le code source Haskell est disponible. Lire le code source de certains modules est un très bon moyen d'apprendre Haskell et de se faire une idée solide de ce dont il s'agit.

Pour trouver des fonctions et savoir dans quel module elles résident, utilisez Hoogle. C'est un moteur de recherche pour Haskell génial, vous pouvez chercher quelque chose par son nom, par le nom de son module, ou même par son type !

Data.List

Le module `Data.List` s'occupe des listes, évidemment. Il contient des fonctions très pratiques. Nous en avons déjà croisées quelques unes (comme `map` et `filter`) parce que le module `Prelude` exporte quelques fonctions de `Data.List` par commodité. Vous n'avez pas besoin d'importer `Data.List` de manière qualifiée, il n'a de collision avec aucun nom du `Prelude`, à part évidemment les fonctions que le `Prelude` lui avait empruntées. Intéressons-nous à d'autres fonctions que nous n'avions pas encore vues.

`intersperse` prend un élément et une liste, et place cet élément entre chaque paire d'éléments de la liste. Démonstration :

```
ghci> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

`intercalate` prend une liste de listes et une liste. Elle insère cette dernière entre toutes les listes de la première, et aplatit le résultat.

```
ghci> intercalate " " ["hey","there","guys"]
"hey there guys"
ghci> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

`transpose` transpose une liste de listes. Si vous pensez à une liste de listes comme à une matrice bidimensionnelle, les colonnes deviennent des lignes et vice versa.

```
ghci> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
ghci> transpose ["hey","there","guys"]
["htg","ehu","yey","rs","e"]
```

Mettons qu'on ait les polynômes $3x^2 + 5x + 9$, $10x^3 + 9$ et $8x^3 + 5x^2 + x - 1$, et que l'on souhaite les additionner. On peut utiliser les listes `[0, 3, 5, 9]`, `[10, 0, 0, 9]` et `[8, 5, 1, -1]` pour les représenter en Haskell. Maintenant, pour les sommer, il suffit de faire :

```
ghci> map sum $ transpose [[0,3,5,9],[10,0,0,9],[8,5,1,-1]]
[18,8,6,17]
```

Lorsqu'on transpose ces trois listes, les puissances de 3 se retrouvent dans la première ligne, les puissances de 2 dans la deuxième, et ainsi de suite. Mapper `sum` sur cette liste produit le résultat désiré.

`foldl'` et `foldl1'` sont des versions strictes de leur équivalents paresseux. Si vous faites des plis paresseux sur des listes très grandes, vous pouvez souvent avoir des erreurs de débordement de pile. Le coupable est la nature paresseuse des plis, la valeur de l'accumulateur n'étant pas réellement mise à jour pendant la phase de pli. Ce qui se passe en réalité, c'est que l'accumulateur fait en quelque sorte la promesse qu'il se rappellera comment calculer sa valeur quand on la lui demandera (NDT : pour cela, ce qu'on appelle un glaçon, traduction de *thunk*, est créé). Cela a lieu pour chaque accumulateur intermédiaire, et tous les glaçons sont empilés sur votre pile, et finissent par la faire déborder. Les plis stricts

ne sont pas de vils paresseux, et calculent les valeurs intermédiaires en route au lieu de remplir votre pile de glaçons. Donc, si jamais vous rencontrez des problèmes de débordement de pile en faisant des plis paresseux, essayez d'utiliser plutôt les versions strictes.

`concat` aplatit une liste de listes en une liste simple.

```
ghci> concat ["foo","bar","car"]
"foobarcar"
ghci> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

Elle enlèvera seulement un niveau d'imbrication. Si vous voulez complètement aplatir la liste `[[[2,3],[3,4,5],[2]],[[2,3],[3,4]]]`, qui est une liste de listes de listes, vous devrez la concaténer deux fois.

`concatMap` équivaut à d'abord mapper une fonction, puis concaténer la liste à l'aide de `concat`.

```
ghci> concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]
```

`and` prend une liste de valeurs booléennes et retourne `True` seulement si toutes les valeurs de la liste sont `True`.

```
ghci> and $ map (>4) [5,6,7,8]
True
ghci> and $ map (==4) [4,4,4,3,4]
False
```

`or` est comme `and`, mais retourne `True` si n'importe laquelle des valeurs booléennes est `True`.

```
ghci> or $ map (==4) [2,3,4,5,6,1]
True
ghci> or $ map (>4) [1,2,3]
False
```

`any` et `all` prennent un prédicat et vérifient respectivement si l'un ou tous les éléments d'une liste satisfont ce prédicat. On les utilise généralement en lieu et place d'un mappage du prédicat sur la liste suivi de `and` ou `or`.

```
ghci> any (==4) [2,3,5,6,1,4]
True
ghci> all (>4) [6,9,10]
True
ghci> all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
ghci> any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True
```

`iterate` prend une fonction et une valeur initiale. Elle applique la fonction à la valeur, puis applique la fonction au résultat, puis applique la fonction au résultat à nouveau, etc. Elle retourne tous ces résultats sous la forme d'une liste infinie.

```
ghci> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
ghci> take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
```

`splitAt` prend un nombre et une liste. Ensuite, elle coupe la liste au nombre d'éléments spécifié, retournant chaque bout dans une paire.

```
ghci> splitAt 3 "heyman"
("hey","man")
ghci> splitAt 100 "heyman"
("heyman","")
ghci> splitAt (-3) "heyman"
("","heyman")
ghci> let (a,b) = splitAt 3 "foobar" in b ++ a
"barfoo"
```

`takeWhile` est très utile. Elle prend des éléments de la liste tant que le prédicat est satisfait, et s'arrête dès qu'un élément l'invalide. C'est très utile.

```
ghci> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
```

```
[6,5,4]
ghci> takeWhile (/=' ') "This is a sentence"
"This"
```

Si l'on cherchait la somme de toutes les puissances de 3 inférieures à 10 000, on ne pourrait pas mapper `(^3)` à `[1..]`, puis appliquer un filtre et sommer le résultat, parce que `filter` ne termine pas sur des listes infinies. Vous savez peut-être que les éléments sont croissants, mais Haskell ne le sait pas. Vous pouvez donc faire ça :

```
ghci> sum $ takeWhile (<10000) $ map (^3) [1..]
53361
```

On applique `(^3)` à une liste infinie, et on coupe dès que ça dépasse 10 000. Il ne reste plus qu'à sommer aisément.

`dropWhile` est similaire, mais elle jette les éléments tant que le prédicat est vrai. Une fois que le prédicat est invalidé, elle retourne ce qui reste de la liste. Fonction très utile et adorable !

```
ghci> dropWhile (/=' ') "This is a sentence"
" is a sentence"
ghci> dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
```

Imaginons qu'on ait une liste des valeurs d'un stock par date. La liste est composée de tuples dont la première composante est la valeur du stock, la deuxième est l'année, la troisième le mois, la quatrième le jour. On désire savoir quand est-ce que la valeur du stock a dépassé 1000 $ pour la première fois !

```
ghci> let stock = [(994.4,2008,9,1),(995.2,2008,9,2),(999.2,2008,9,3),(1001.4,2008,9,4),(998.3,2008,9,5)]
ghci> head (dropWhile (\(val,y,m,d) -> val < 1000) stock)
    (1001.4,2008,9,4)
```

`span` est un peu comme `takeWhile`, mais retourne une paire de listes. La première liste contient tout ce que contiendrait la liste retournée par `takeWhile` appelée avec le même prédicat et la même liste. La seconde liste correspond à ce qui aurait été laissé.

```
ghci> let (fw, rest) = span (/=' ') "This is a sentence" in "First word:" ++ fw ++ ", the rest:" ++ rest
"First word: This, the rest: is a sentence"
```

Alors que `span` s'étend sur la liste tant que la prédicat est vrai, `break` la coupe dès qu'il devient vrai. Ainsi, `break p` est équivalent à `span (not . p)`.

```
ghci> break (==4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
ghci> span (/=4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

Dans le tuple retourné par `break`, la seconde liste débute avec le premier élément ayant satisfait le prédicat.

`sort` trie une liste. Le type des éléments de la liste doit être membre de la classe de types `Ord`, parce que si l'on ne peut pas ordonner les éléments, eh bien on ne peut pas les trier.

```
ghci> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
ghci> sort "This will be sorted soon"
"    Tbdeehiillnooorssstw"
```

`group` prend une liste et groupe les éléments adjacents en sous-listes s'ils sont égaux.

```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

Si l'on trie une liste avant de grouper ses éléments, on peut connaître le compte de chaque élément.

```
ghci> map (\l@(x:xs) -> (x,length l)) . group . sort $ [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[(1,4),(2,7),(3,2),(5,1),(6,1),(7,1)]
```

`inits` et `tails` sont comme `init` et `tail`, mais appliquent ces fonctions récursivement tant que faire se peut. Observez :

```
ghci> inits "w00t"
["","w","w0","w00","w00t"]
ghci> tails "w00t"
["w00t","00t","0t","t",""]
ghci> let w = "w00t" in zip (inits w) (tails w)
[("","w00t"),("w","00t"),("w0","0t"),("w00","t"),("w00t","")]
```

Utilisons un pli pour implémenter la recherche d'une liste dans une sous-liste.

```
search :: (Eq a) => [a] -> [a] -> Bool
search needle haystack =
    let nlen = length needle
    in  foldl (\acc x -> if take nlen x == needle then True else acc) False (tails haystack)
```

D'abord, on appelle **`tails`** avec la liste qu'on cherche. Puis on parcourt chaque queue retournée pour voir si elle commence par ce qu'on cherche.

On vient de faire une fonction qui se comporte comme **`isInfixOf`**. **`isInfixOf`** cherche une sous-liste dans une liste et retourne **`True`** si la sous-liste qu'on cherche apparaît quelque part dans la liste cible.

```
ghci> "cat" `isInfixOf` "im a cat burglar"
True
ghci> "Cat" `isInfixOf` "im a cat burglar"
False
ghci> "cats" `isInfixOf` "im a cat burglar"
False
```

`isPrefixOf` et **`isSuffixOf`** cherchent une sous-liste respectivement au début et à la fin d'une liste.

```
ghci> "hey" `isPrefixOf` "hey there!"
True
ghci> "hey" `isPrefixOf` "oh hey there!"
False
ghci> "there!" `isSuffixOf` "oh hey there!"
True
ghci> "there!" `isSuffixOf` "oh hey there"
False
```

`elem` et **`notElem`** cherchent si un élément appartient ou n'appartient pas à une liste.

`partition` prend une liste et un prédicat, et retourne une paire de listes. La première liste contient tous les éléments qui satisfont le prédicat, la seconde tous les autres.

```
ghci> partition (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOBMORGAN","sidneyeddy")
ghci> partition (>3) [1,3,5,6,3,2,1,0,3,7]
([5,6,7],[1,3,3,2,1,0,3])
```

Il est important de saisir la différence avec **`span`** et **`break`** :

```
ghci> span (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOB","sidneyMORGANeddy")
```

Alors que **`span`** et **`break`** s'arrêtent dès qu'ils rencontrent le premier élément qui ne satisfait pas ou qui satisfait le prédicat, **`partition`** traverse toute la liste et la découpe conformément au prédicat.

`find` prend une liste et un prédicat et retourne le premier élément de la liste satisfaisant le prédicat. Mais il retourne ce prédicat encapsulé dans une valeur de type **`Maybe`**. Nous couvrirons les types de données algébriques plus en profondeur dans le prochain chapitre, mais pour l'instant, voilà ce que vous avez besoin de savoir : une valeur **`Maybe`** peut soit être **`Just something`**, soit **`Nothing`**. Tout comme une liste peut être soit la liste vide, soit une liste avec des éléments, une valeur **`Maybe`** peut contenir soit aucun, soit un élement. Et comme le type d'une liste d'entiers est par exemple **`[Int]`**, le type d'un éventuel entier est **`Maybe Int`**. Bon, faisons un tour de **`find`** :

```
ghci> find (>4) [1,2,3,4,5,6]
Just 5
ghci> find (>9) [1,2,3,4,5,6]
Nothing
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

Prêtez attention au type de `find`. Elle retourne un `Maybe a`. C'est un peu comme s'il y avait un `[a]`, seulement qu'une valeur de type `Maybe` ne peut contenir qu'un ou zéro élément, alors qu'une liste peut en contenir zéro, un ou plus.

Vous vous souvenez quand nous cherchions la première fois que notre stock dépassait 1000 $? On avait fait `head (dropWhile (\(val,y,m,d) -> val < 1000) stock)`. Rappelez-vous que `head` n'est pas sûre. Que se serait-il passé si le stock n'avait jamais dépassé 1000 $? Notre application de `dropWhile` aurait retourné la liste vide, et essayer de prendre sa tête aurait été une erreur. Cependant, si l'on réécrivait cela `find (\(val,y,m,d) -> val > 1000) stock`, on serait beaucoup plus tranquille. Si notre stock ne dépassait jamais 1000 $ (donc, aucun élément ne satisfait le prédicat), on récupérerait `Nothing`. Mais s'il y avait une réponse correcte dans la liste, on récupérerait, mettons, `Just (1001.4, 2008, 9, 4)`.

`elemIndex` est un peu comme `elem`, mais ne retourne pas une valeur booléenne. Elle retourne éventuellement l'indice de l'élément qu'on cherche. Si cet élément n'est pas dans la liste, elle retourne `Nothing`.

```
ghci> :t elemIndex
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
ghci> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
ghci> 10 `elemIndex` [1,2,3,4,5,6]
Nothing
```

`elemIndices` est comme `elemIndex`, seulement qu'elle retourne une liste d'indices dans le cas où l'élément que l'on cherche apparaîtrait plusieurs fois dans la liste. Puisqu'on utilise des listes pour représenter les indices, on n'a pas besoin d'un type `Maybe`, il suffit de représenter un échec comme une liste vide, qui sera donc très synonyme de `Nothing`.

```
ghci> ' ' `elemIndices` "Where are the spaces?"
[5,9,13]
```

`findIndex` est comme `find`, mais retourne éventuellement l'indice du premier élément qui satisfait le prédicat. `findIndices` retourne les indices de tous les éléments qui satisfont le prédicat sous forme d'une liste.

```
ghci> findIndex (==4) [5,3,2,1,6,4]
Just 5
ghci> findIndex (==7) [5,3,2,1,6,4]
Nothing
ghci> findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
[0,6,10,14]
```

Nous avons déjà couvert `zip` et `zipWith`. Nous avions vu qu'elles zippent ensemble deux listes, soit sous la forme d'un tuple, soit en appliquant une fonction binaire (c'est-à-dire qui prend deux paramètres). Mais comment zipper ensemble trois listes ? Ou zipper trois listes à l'aide d'une fonction qui attend trois paramètres ? Pour cela, il existe `zip3`, `zip4`, etc. et `zipWith3`, `zipWith4`, etc. Ces variantes existent jusqu'à 7. Bien que ça ait l'air un peu arbitraire et ad-hoc, ça s'avère plutôt suffisant, car vous ne voudrez sûrement jamais zipper ensemble 8 listes. Il existe un moyen astucieux de zipper ensemble une infinité de listes, mais nous ne sommes pas assez avancé pour couvrir ça maintenant.

```
ghci> zipWith3 (\x y z -> x + y + z) [1,2,3] [4,5,2,2] [2,2,3]
[7,9,8]
ghci> zip4 [2,3,3] [2,2,2] [5,5,3] [2,2,2]
[(2,2,5,2),(3,2,5,2),(3,2,3,2)]
```

Comme un zip normal, des listes plus longues que la plus courte des listes zippées seront coupées.

`lines` est une fonction très utile pour traiter des fichiers ou une entrée. Elle prend une chaîne de caractères et retourne une liste des différentes lignes.

```
ghci> lines "first line\nsecond line\nthird line"
["first line","second line","third line"]
```

`'\n'` est le caractère UNIX pour aller à la ligne. L'antislash a une signification spéciale dans les caractères et les chaînes de caractères.

`unlines` est la fonction inverse de `lines`. Elle prend une liste de chaînes de caractères et les joint ensemble avec des `'\n'`.

```
ghci> unlines ["first line", "second line", "third line"]
"first line\nsecond line\nthird line\n"
```

`words` et `unwords` servent à séparer une ligne de texte en plusieurs mots, ou à joindre une liste de mots en un texte. Très pratique.

```
```

```
ghci> words "hey these are the words in this sentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> words "hey these           are    the words in this\nsentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> unwords ["hey","there","mate"]
"hey there mate"
```

Nous avons déjà mentionné **`nub`**. Elle prend une liste et retire les éléments en double, retournant une liste où chaque élément est aussi unique qu'un flocon de neige ! Le nom de la fonction est un peu bizarre. Il s'avère que "nub" désigne l'essence, le cœur de quelque chose. À mon avis, ils devraient plutôt utiliser des vrais noms pour les fonctions, pas des mots de vieilles personnes.

```
ghci> nub [1,2,3,4,3,2,1,2,3,4,3,2,1]
[1,2,3,4]
ghci> nub "Lots of words and stuff"
"Lots fwrdanu"
```

`delete` prend un élément, une liste, et supprime la première occurence de cet élément dans la liste.

```
ghci> delete 'h' "hey there ghang!"
"ey there ghang!"
ghci> delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere ghang!"
ghci> delete 'h' . delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere gang!"
```

`\\` est la fonction de différence sur les listes. Elle se comporte comme la différence ensembliste. Pour chaque élément de la liste de droite, elle supprime un élément correspondant dans la liste de gauche.

```
ghci> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
ghci> "Im a big baby" \\ "big"
"Im a  baby"
```

Faire **`[1..10] \\ [2, 5, 9]`** est comme faire **`delete 2 . delete 5 . delete 9 $ [1..10]`**.

`union` se comporte aussi comme une fonction sur les ensembles. Elle retourne l'union de deux listes. En gros, elle parcourt toute la liste de droite et ajoute les éléments en queue de celle de gauche s'ils n'y sont pas déjà. Attention, elle supprime les doublons dans la liste de droite !

```
ghci> "hey man" `union` "man what's up"
"hey manwt'sup"
ghci> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
```

`intersect` fonctionne comme l'intersection ensembliste. Elle ne retourne que les éléments apparaissant dans les deux listes.

```
ghci> [1..7] `intersect` [5..10]
[5,6,7]
```

`insert` prend un élément et une liste d'éléments qui peuvent être triés et insère l'élément à la dernière position où il reste inférieur au prochain élément. **`insert`** va parcourir la liste jusqu'à trouver un élément plus grand que celui passé en paramètre, et va alors insérer ce dernier devant l'autre.

```
ghci> insert 4 [3,5,1,2,8,2]
[3,4,5,1,2,8,2]
ghci> insert 4 [1,3,4,4,1]
[1,3,4,4,4,1]
```

Le **`4`** est inséré juste après le **`3`** et juste avant le **`5`** dans le premier exemple, et entre le **`3`** et le **`4`** dans le second.

Propriété intéressante : si l'on utilise **`insert`** pour insérer dans une liste triée, la liste reste triée.

```
ghci> insert 4 [1,2,3,5,6,7]
[1,2,3,4,5,6,7]
ghci> insert 'g' $ ['a'..'f'] ++ ['h'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> insert 3 [1,2,4,3,2,1]
[1,2,3,4,3,2,1]
```

Ce que `length`, `take`, `drop`, `splitAt`, `!!` et `replicate` ont de commun, c'est qu'elles prennent ou retournent un `Int`, alors qu'elles pourraient être plus générales en utilisant plutôt un type appartenant aux classes `Integral` ou `Num` (en fonction de la fonction). Elles ne le font pas pour des raisons historiques. Réparer cela détruirait certainement beaucoup de code existant. C'est pourquoi `Data.List` contient les équivalents plus génériques, nommés `genericLength`, `genericTake`, `genericDrop`, `genericSplitAt`, `genericIndex` et `genericReplicate`. Par exemple, `length` a pour signature `length :: [a] -> Int`. Si l'on essaie de récupérer la moyenne d'une liste en faisant `let xs = [1..6] in sum xs / length xs`, on obtient une erreur de type, parce que `/` ne fonctionne pas sur les `Int`. `genericLength`, au contraire, a pour signature `genericLength :: (Num a) => [b] -> a`. Puisqu'un `Num` peut se faire passer pour un nombre à virgule flottante, trouver la moyenne en faisant `let xs = [1..6] in sum xs / genericLength xs` marchera.

Les fonctions `nub`, `delete`, `union`, `intersect` et `group` ont toute un équivalent plus général, respectivement `nubBy`, `deleteBy`, `unionBy`, `intersectBy` et `groupBy`. La différence entre les deux, c'est que les premières utilisent `==` pour tester l'égalité, alors que les versions en *By* prennent en paramètre une fonction utilisée pour tester l'égalité. `group` est donc équivalente à `groupBy (==)`.

Par exemple, mettons qu'on a une liste qui décrit les valeurs d'une fonction à chaque seconde, et on voudrait la segmenter entre les valeurs positives et les valeurs négatives. Un `group` normal ne regrouperait que les valeurs adjacentes égales entre elles. On voudrait les grouper en fonction de leur signe. C'est là que `groupBy` entre en jeu ! La fonction d'égalité des fonctions en *By* doit prendre deux éléments de même type et retourne `True` si elle les considère égales selon ses propres critères.

```
ghci> let values = [-4.3, -2.4, -1.2, 0.4, 2.3, 5.9, 10.5, 29.1, 5.3, -2.4, -14.5, 2.9, 2.3]
ghci> groupBy (\x y -> (x > 0) == (y > 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

On voit ici clairement quelles sections sont positives et négatives. La fonction d'égalité fournie prend deux éléments et retourne `True` seulement s'ils sont tous les deux positifs ou tous les deux négatifs. Elle pourrait aussi être écrite `\x y -> (x > 0) && (y > 0) || (x <= 0) && (y <= 0)`, bien que je trouve la première version plus lisible. Une manière encore plus claire d'écrire les fonctions d'égalité pour les fonctions en *By* consiste à importer `on` depuis `Data.Function`. `on` est définie ainsi :

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
f `on` g = \x y -> f (g x) (g y)
```

Ainsi, faire `(==) `on` (> 0)` retourne une fonction d'égalité qui ressemble à `\x y -> (x > 0) == (y > 0)`. `on` est très utilisée avec les fonctions *By* parce qu'avec elle, on peut faire :

```
ghci> groupBy ((==) `on` (> 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

Super lisible, non ? Vous pouvez le lire tout haut : groupe ceci par égalité sur le fait que les éléments soient plus grands que zéro.

De façon similaire, les fonctions `sort`, `insert`, `maximum` et `minimum` ont aussi un équivalent plus général. Les fonctions comme `groupBy` prenaient une fonction pour déterminer si deux éléments étaient égaux. `sortBy`, `insertBy`, `maximumBy` et `minimumBy` prennent une fonction pour déterminer si un élément est plus petit, plus grand ou égal à l'autre. La signature de `sortBy` est `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`. Si vous vous souvenez bien, le type `Ordering` peut prendre pour valeurs `LT`, `EQ` ou `GT`. `sort` est équivalent à `sortBy compare`, car `compare` prend juste deux éléments dont le type est membre de la classe `Ord` et retourne leur relation d'ordre.

Les listes peuvent être comparées, mais lorsqu'elles le sont, elles sont comparées lexicographiquement. Et si l'on avait une liste de listes et que l'on souhaitait la trier non pas par rapport au contenu des listes internes, mais plutôt en fonction de leur longueur ? Vous l'avez peut-être déjà deviné, on utilisera la fonction `sortBy`.

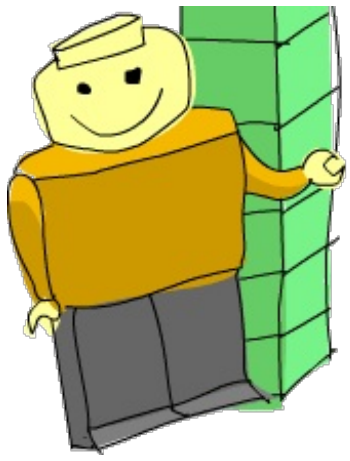
```
ghci> let xs = [[5,4,5,4,4],[1,2,3],[3,5,4,3],[],[2],[2,2]]
ghci> sortBy (compare `on` length) xs
[[],[2],[2,2],[1,2,3],[3,5,4,3],[5,4,5,4,4]]
```

Génial ! `compare `on` length`… man, on dirait presque de l'anglais naturel ! Si vous n'êtes pas certain de ce que fait `on` ici, `compare `on` length` est équivalent à `y -> length x `compare` length y`. Quand vous utilisez des fonctions en *By* qui attendent une fonction d'égalité, vous ferez souvent `(==) `on` something`, et quand vous utilisez des fonctions en *By* qui attendent une fonction de comparaison, vous ferez souvent `compare `on` length`.

Data.Char

Le module `Data.Char` fait ce que son nom indique. Il exporte des fonctions qui agissent sur des caractères. Et qui servent aussi pour filtrer ou mapper sur des chaînes de caractères, puisque ce ne sont que des listes de caractères.

`Data.Char` exporte tout un tas de prédicats sur les caractères. C'est-à-dire, des fonctions qui prennent un caractère et nous indiquent s'il satisfait une condition. Les voici :



`isControl` vérifie si c'est un caractère de contrôle (NDT : caractères non affichables du sous-ensemble Latin–1 d'Unicode).

`isSpace` vérifie si c'est un caractère d'espacement. Cela inclue les espaces, les tabulations, les nouvelles lignes, etc.

`isLower` vérifie si le caractère est minuscule.

`isUpper` vérifie si le caractère est majuscule.

`isAlpha` vérifie si le caractère est une lettre.

`isAlphaNum` vérifie si le caractère est une lettre ou un chiffre.

`isPrint` vérifie si le caractère est affichable. Les caractères de contrôle, par exemple, ne le sont pas.

`isDigit` vérifie si le caractère est un chiffre.

`isOctDigit` vérifie si le caractère est un chiffre octal.

`isHexDigit` vérifie si le caractère est un chiffre hexadécimal.

`isLetter` vérifie si le caractère est une lettre.

`isMark` vérifie les caractères Unicode de marque. Ce sont des caractères qui se combinent au précédent pour créer des caractères accentués. Utile pour nous français.

`isNumber` vérifie si un caractère est numérique.

`isPunctuation` vérifie si le caractère est un caractère de ponctuation.

`isSymbol` vérifie si le caractère est un symbole mathématique ou monétaire.

`isSeparator` vérifie les espaces et séparateurs Unicode.

`isAscii` vérifie si un caractère fait partie des 128 premiers caractères Unicode.

`isLatin1` vérifie si le code fait partie des 256 premiers caractères Unicode.

`isAsciiUpper` vérifie si c'est un caractère ASCII majuscule.

`isAsciiLower` vérifie si c'est un caractère ASCII minuscule.

Tous ces prédicats ont pour signature `Char -> Bool`. La plupart du temps, vous les utiliserez pour filtrer des chaînes de caractères. Par exemple, disons qu'on écrive un programme qui prend un nom d'utilisateur, et que ce nom doit être composé seulement de caractères alphanumériques. On peut utiliser la fonction `all` de `Data.List` en combinaison avec les prédicats de `Data.Char` pour vérifier la validité du nom de l'utilisateur.

```
ghci> all isAlphaNum "bobby283"
True
ghci> all isAlphaNum "eddy the fish!"
False
```

Cool ! Au cas où vous auriez oublié, `all` prend un prédicat et retourne `True` seulement si tous les éléments de la liste valident ce prédicat.

On peut aussi utiliser `isSpace` pour simuler la fonction `words` de `Data.List`.

```
ghci> words "hey guys its me"
["hey","guys","its","me"]
ghci> groupBy ((==) `on` isSpace) "hey guys its me"
["hey"," ","guys"," ","its"," ","me"]
ghci>
```

Hmmm… ça marche à peu près comme `words`, mais il nous reste les éléments qui ne contiennent que des espaces. Que devrions-nous faire ? Je sais, filtrons ces importuns !

```
ghci> filter (not . any isSpace) . groupBy ((==) `on` isSpace) $ "hey guys its me"
["hey","guys","its","me"]
```

Ah !

Le module `Data.Char` exporte également un type de donnée similaire à `Ordering`. `Ordering` peut prendre pour valeur `LT`, `EQ` ou `GT`. C'est une sorte d'énumération. Cela décrit les éventualités du résultat d'une comparaison. Le type `GeneralCategory` est aussi une énumération. Il décrit les différentes catégories auxquelles un caractère peut appartenir. La fonction principale retournant la catégorie d'un caractère est `generalCategory`. Son type est `generalCategory :: Char -> GeneralCategory`. Il y a environ 31 catégories, donc on ne va pas les lister ici, mais jouons un peu avec la fonction.

```
ghci> generalCategory ' '
Space
ghci> generalCategory 'A'
UppercaseLetter
ghci> generalCategory 'a'
LowercaseLetter
ghci> generalCategory '.'
OtherPunctuation
ghci> generalCategory '9'
DecimalNumber
ghci> map generalCategory " \t\nA9?|"
[Space,Control,Control,UppercaseLetter,DecimalNumber,OtherPunctuation,MathSymbol]
```

Puisque le type `GeneralCategory` est membre de la classe `Eq`, on peut aussi écrire des choses comme `generalCategory c == Space`.

`toUpper` convertit un caractère minuscule en majuscule. Les espaces, nombres et autres restent inchangés.

`toLower` convertit un caractère majuscule en minuscule.

`toTitle` convertit un caractère en casse de titre. Pour la plupart des caractères, la casse de titre est majuscule.

`digitToInt` convertit un caractère en un `Int`. Pour réussir, le caractère doit être dans les intervalles `'0'..'9'`, `'a'..'f'` ou `'A'..'F'`.

```
ghci> map digitToInt "34538"
[3,4,5,3,8]
ghci> map digitToInt "FF85AB"
[15,15,8,5,10,11]
```

`intToDigit` est la fonction inverse de `digitToInt`. Elle prend un `Int` compris dans `0..15` et le convertit en caractère minuscule.

```
ghci> intToDigit 15
'f'
ghci> intToDigit 5
'5'
```

Les fonctions `ord` et `chr` convertissent un caractère vers sa valeur numérique et vice versa.

```
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
```

La différence entre les valeurs `ord` de deux caractères correspond à leur espacement dans la table Unicode.

Le chiffre de César est une méthode primitive d'encodage de messages à base de décalage de chaque caractère d'un nombre fixé de positions, par rapport à l'alphabet. On peut facilement créer un chiffre de César nous-mêmes, sans se restreindre à l'alphabet.

```
encode :: Int -> String -> String
encode shift msg =
    let ords = map ord msg
        shifted = map (+ shift) ords
    in  map chr shifted
```

Ici, on convertit d'abord la chaîne de caractères en une liste de nombres. Puis, on ajoute le décalage à chacun des nombres, avant de reconvertir cette liste de nombres en caractères. Si vous êtes un cowboy de la composition, vous pouvez écrire le corps de cette fonction comme `map (chr . (+ shift) . ord) msg`. Essayons d'encoder quelques messages.

```
ghci> encode 3 "Heeeeey"
"Khhhhh|"
ghci> encode 4 "Heeeeey"
"Liiiii}"
ghci> encode 1 "abcd"
```

```
"bcde"
ghci> encode 5 "Marry Christmas! Ho ho ho!"
"Rfww~%Hmwnxyrfx&%Mt%mt%mt&"
```

C'est effectivement encodé. Décoder ces messages revient simplement à décaler les nombres d'autant de places dans le sens opposé au sens initial.

```
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg
```

```
ghci> encode 3 "Im a little teapot"
"Lp#d#olwwoh#whdsrw"
ghci> decode 3 "Lp#d#olwwoh#whdsrw"
"Im a little teapot"
ghci> decode 5 . encode 5 $ "This is a sentence"
"This is a sentence"
```

Data.Map

Les listes associatives (ou dictionnaires) sont des listes utilisées pour stocker des paires clé-valeur, où l'ordre n'est pas important. Par exemple, on peut utiliser une liste associative pour stocker des numéros de téléphone, où les numéros de téléphone seraient les valeurs, et les noms des personnes les clés. On se fiche de l'ordre dans lequel c'est rangé, on souhaite seulement pouvoir récupérer le bon numéro de téléphone pour une personne donnée.

La façon la plus évidente de représenter des listes associatives en Haskell est sous forme de listes de paires. La première composante de la paire serait la clé, la seconde serait la valeur. Voici un exemple de liste associative de numéros de téléphone :

```
phoneBook =
    [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

Malgré cette indentation bizarre, c'est bien une liste de paires de chaînes de caractères. L'opération la plus courante associée aux listes associatives est la recherche d'une valeur par sa clé. Créons une fonction qui trouve une valeur à partir d'une clé.

```
findKey :: (Eq k) => k -> [(k,v)] -> v
findKey key xs = snd . head . filter (\(k,v) -> key == k) $ xs
```

Plutôt simple. La fonction prend une clé, une liste, filtre la liste de façon à ce que seules les clés correspondantes ne restent, prend la première paire clé-valeur, et en retourne la valeur. Mais que se passe-t-il si la clé que l'on cherche n'est pas dans la liste ? Hmmm. Ici, si la clé n'est pas dans la liste, on va essayer de prendre la tête d'une liste vide, ce qui génère une erreur à l'exécution. On ne devrait pas laisser notre programme planter si facilement, utilisons donc un type de données `Maybe`. Si l'on ne trouve pas la clé, on retourne `Nothing`. Si l'on trouve quelque chose, on retourne `Just something`, où *something* sera la valeur correspondant à la clé.

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key [] = Nothing
findKey key ((k,v):xs) = if key == k
                         then Just v
                         else findKey key xs
```

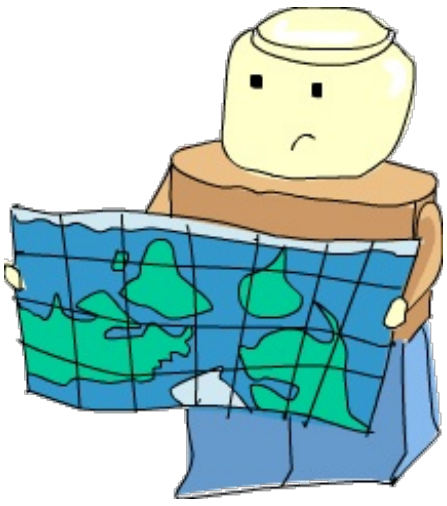
Regardez la déclaration de type. Elle prend une clé qui dispose d'un test d'égalité, une liste associative, et retourne éventuellement une valeur. Ça a l'air correct.

Ceci est un cas d'école de fonction récursive opérant sur une liste. Cas de base, découpage de la liste en queue et tête, appels récursifs, c'est tout ce qu'il se passe. C'est le motif classique du pli, implémentons-le donc comme un pli.

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key = foldr (\(k,v) acc -> if key == k then Just v else acc) Nothing
```

Note : il est généralement mieux d'utiliser des plis pour effectuer de la récursivité standard sur les listes plutôt que d'écrire explicitement la récursivité, car c'est plus simple à lire et à identifier. Tout le monde connaît les plis et sait en identifier un lorsqu'il voit un appel à `foldr`, mais cela prend plus de temps de déchiffrer une récursivité explicite.

```
ghci> findKey "penny" phoneBook
Just "853-2492"
ghci> findKey "betty" phoneBook
Just "555-2938"
ghci> findKey "wilma" phoneBook
Nothing
```



Ça fonctionne comme un charme ! Si on a le numéro de téléphone de cette fille, on récupère `Just` le numéro, sinon, on récupère `Nothing`.

On vient juste d'implémenter la fonction `lookup` de `Data.List`. Si l'on veut trouver la valeur correspondant à une clé, on doit traverser la liste jusqu'à ce qu'on la trouve. Le module `Data.Map` offre des listes associatives qui sont beaucoup plus rapides (car elles sont implémentées à l'aide d'arbres) et fournit également un paquet de fonctions utiles. À partir de maintenant, nous dirons qu'on travaille sur des maps plutôt que des listes associatives.

`Data.Map` exporte des fonctions qui collisionnent avec le `Prelude` et `Data.List`, donc on l'importe qualifié.

```
import qualified Data.Map as Map
```

Placez cette ligne dans un script et chargez-le dans GHCi.

Allons à présent voir ce que `Data.Map` a en magasin pour nous ! Voilà la liste des fonctions.

`fromList` prend une liste associative (sous forme de liste) et retourne une map avec les mêmes associations.

```
ghci> Map.fromList [("betty","555-2938"),("bonnie","452-2928"),("lucille","205-2928")]
fromList [("betty","555-2938"),("bonnie","452-2928"),("lucille","205-2928")]
ghci> Map.fromList [(1,2),(3,4),(3,2),(5,5)]
fromList [(1,2),(3,2),(5,5)]
```

S'il y a des clés en doublon dans la liste originale, les doublons sont ignorés. Voici la signature de type de `fromList` :

```
Map.fromList :: (Ord k) => [(k, v)] -> Map.Map k v
```

Elle indique qu'elle prend une liste de paires `k` et `v` et retourne une map qui mappe des clés de type `k` sur des valeurs de type `v`. Remarquez que lorsqu'on faisait des listes associatives sous forme de liste, on avait juste besoin de l'égalité sur les clés (leur type appartenait à `Eq`), mais à présent elles doivent être ordonnables. C'est une contrainte essentielle pour le module `Data.Map`. Il a besoin de pouvoir ordonner les clés afin de les arranger en arbre.

Vous devriez toujours utiliser `Data.Map` pour associer des clés-valeurs, à moins que vous ayez des clés qui ne sont pas membres d'`Ord`.

`empty` représente une map vide. Elle ne prend pas d'argument, et retourne une map vide.

```
ghci> Map.empty
fromList []
```

`insert` prend une clé, une valeur et une map, et retourne une nouvelle map identique à l'ancienne, avec en plus la nouvelle clé-valeur insérée.

```
ghci> Map.empty
fromList []
ghci> Map.insert 3 100 Map.empty
fromList [(3,100)]
ghci> Map.insert 5 600 (Map.insert 4 200 ( Map.insert 3 100  Map.empty))
fromList [(3,100),(4,200),(5,600)]
ghci> Map.insert 5 600 . Map.insert 4 200 . Map.insert 3 100 $ Map.empty
fromList [(3,100),(4,200),(5,600)]
```

On peut implémenter notre propre `fromList` en utilisant la map vide, `insert` et un pli. Observez :

```
fromList' :: (Ord k) => [(k,v)] -> Map.Map k v
fromList' = foldr (\(k,v) acc -> Map.insert k v acc) Map.empty
```

C'est un pli plutôt simple. On commence avec la map vide, et on la plie depuis la droite, en insérant les paires clé-valeur dans l'accumulateur tout du long.

`null` vérifie si une map est vide.

```
ghci> Map.null Map.empty
True
ghci> Map.null $ Map.fromList [(2,3),(5,5)]
False
```

`size` retourne la taille d'une map.

```
ghci> Map.size Map.empty
0
ghci> Map.size $ Map.fromList [(2,4),(3,3),(4,2),(5,4),(6,4)]
5
```

`singleton` prend une clé et une valeur, et crée une map qui contient uniquement cette association.

```
ghci> Map.singleton 3 9
fromList [(3,9)]
ghci> Map.insert 5 9 $ Map.singleton 3 9
fromList [(3,9),(5,9)]
```

`lookup` fonctionne comme `lookup` de `Data.List`, mais opère sur des maps. Elle retourne `Just something` si elle trouve quelque chose, `Nothing` sinon.

`member` est un prédicat qui prend une clé, une map, et indique si la clé est dans la map.

```
ghci> Map.member 3 $ Map.fromList [(3,6),(4,3),(6,9)]
True
ghci> Map.member 3 $ Map.fromList [(2,5),(4,5)]
False
```

`map` et `filter` fonctionnent comme leur équivalent.

```
ghci> Map.map (*100) $ Map.fromList [(1,1),(2,4),(3,9)]
fromList [(1,100),(2,400),(3,900)]
ghci> Map.filter isUpper $ Map.fromList [(1,'a'),(2,'A'),(3,'b'),(4,'B')]
fromList [(2,'A'),(4,'B')]
```

`toList` est l'inverse de `fromList`.

```
ghci> Map.toList . Map.insert 9 2 $ Map.singleton 4 3
[(4,3),(9,2)]
```

`keys` et `elems` retournent des listes de clés et de valeurs respectivement. `keys` est l'équivalent de `map fst . Map.toList`, et `elems` l'équivalent de `map snd . Map.toList`.

`fromListWith` est une petite fonction assez cool. Elle agit un peu comme `fromList`, mais elle ne jette pas les clés en double, et utilise une fonction passée en paramètre pour décider quoi faire d'elles. Disons qu'une fille peut avoir plusieurs numéros de téléphone, et qu'on a une liste associative comme celle-ci :

```
phoneBook =
    [("betty","555-2938")
    ,("betty","342-2492")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("patsy","943-2929")
    ,("patsy","827-9162")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ,("penny","555-2111")
    ]
```

Maintenant, si on utilise seulement `fromList` pour transformer cela en map, on va perdre quelques numéros ! Voilà ce qu'on va faire :

```
phoneBookToMap :: (Ord k) => [(k, String)] -> Map.Map k String
phoneBookToMap xs = Map.fromListWith (\number1 number2 -> number1 ++ ", " ++ number2) xs
```

```
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
"827-9162, 943-2929, 493-2928"
```

```
ghci> Map.lookup "wendy" $ phoneBookToMap phoneBook
"939-8282"
ghci> Map.lookup "betty" $ phoneBookToMap phoneBook
"342-2492, 555-2938"
```

Si une clé en double est trouvée, la fonction qu'on passe est utilisée pour combiner les valeurs en une nouvelle valeur. On aurait aussi pu transformer toutes les valeurs de la liste en listes singleton, puis utiliser `(++)` comme combinateur.

```
phoneBookToMap :: (Ord k) => [(k, a)] -> Map.Map k [a]
phoneBookToMap xs = Map.fromListWith (++) $ map (\(k,v) -> (k,[v])) xs
```

```
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
["827-9162","943-2929","493-2928"]
```

Plutôt chic ! Un autre cas d'utilisation est celui où l'on veut créer une map à partir d'une liste d'association, et lors d'un doublon, l'on souhaite conserver la plus grande des valeurs par exemple.

```
ghci> Map.fromListWith max [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,100),(3,29),(4,22)]
```

On pourrait tout aussi bien choisir d'additionner des valeurs de même clé.

```
ghci> Map.fromListWith (+) [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,108),(3,62),(4,37)]
```

`insertWith` est à `insert` ce que `fromListWith` est à `fromList`. Elle insère une paire clé-valeur dans la map, et si la map contient déjà cette clé, utilise la fonction passée afin de déterminer quoi faire.

```
ghci> Map.insertWith (+) 3 100 $ Map.fromList [(3,4),(5,103),(6,339)]
fromList [(3,104),(5,103),(6,339)]
```

Ce n'était qu'un aperçu de `Data.Map`. Vous pouvez voir la liste complète des fonctions dans la documentation.

Data.Set

Le module `Data.Set` nous offre des ensembles. Comme les ensembles en mathématiques. Les ensembles sont un peu comme un mélange de listes et de maps. Tous les éléments d'un ensemble sont uniques. Et comme ils sont implémentés en interne à l'aide d'arbres (comme les maps de `Data.Map`), ils sont ordonnés. Vérifier l'appartenance, insérer, supprimer, etc. sont des opérations bien plus rapides sur les ensembles que sur les listes. Les opérations les plus courantes sur les ensembles sont l'insertion, le test d'appartenance et la conversion en liste.

Les noms dans `Data.Set` collisionnent beaucoup avec le `Prelude` et `Data.List`, on fait donc un import qualifié.

Placez cette déclaration dans un script :

```
import qualified Data.Set as Set
```

Puis chargez ce script via GHCi.

Supposons qu'on ait deux bouts de texte. On souhaite trouver les caractères utilisés dans les deux chaînes.

```
text1 = "I just had an anime dream. Anime... Reality... Are they so different?"
text2 = "The old man left his garbage can out and now his trash is all over my lawn!"
```

La fonction `fromList` fonctionne comme on peut s'y attendre. Elle prend une liste et la convertit en un ensemble.

```
ghci> let set1 = Set.fromList text1
ghci> let set2 = Set.fromList text2
ghci> set1
fromList " .?AIRadefhijlmnorstuy"
ghci> set2
fromList " !Tabcdefghilmnorstuvwy"
```

Comme vous pouvez le voir, les éléments sont ordonnés et chaque élément est unique. Maintenant, utilisons la fonction **`intersection`** pour voir les éléments qu'ils partagent.

```
ghci> Set.intersection set1 set2
fromList " adefhilmnorstuy"
```

On peut utiliser la fonction **`difference`** pour voir quelles lettres sont dans le premier ensemble mais pas le second, et vice versa.

```
ghci> Set.difference set1 set2
fromList ".?AIRj"
ghci> Set.difference set2 set1
fromList "!Tbcgvw"
```

Ou bien, on peut voir les lettres uniques à chaque ensemble en utilisant **`union`**.

```
ghci> Set.union set1 set2
fromList " !.?AIRTabcdefghijlmnorstuvwy"
```

Les fonctions **`null`**, **`size`**, **`member`**, **`empty`**, **`singleton`**, **`insert`** et **`delete`** fonctionnent toutes comme on peut s'y attendre.

```
ghci> Set.null Set.empty
True
ghci> Set.null $ Set.fromList [3,4,5,5,4,3]
False
ghci> Set.size $ Set.fromList [3,4,5,3,4,5]
3
ghci> Set.singleton 9
fromList [9]
ghci> Set.insert 4 $ Set.fromList [9,3,8,1]
fromList [1,3,4,8,9]
ghci> Set.insert 8 $ Set.fromList [5..10]
fromList [5,6,7,8,9,10]
ghci> Set.delete 4 $ Set.fromList [3,4,5,4,3,4,5]
fromList [3,5]
```

On peut aussi tester si un ensemble est un sous-ensemble (ou sous-ensemble strict) d'un autre. Un ensemble A est sous-ensemble de B si B contient tous les éléments de A. A est un sous-ensemble strict de B si B contient tous les éléments de A, mais a strictement plus d'éléments.

```
ghci> Set.fromList [2,3,4] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isProperSubsetOf` Set.fromList [1,2,3,4,5]
False
ghci> Set.fromList [2,3,4,8] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
False
```

On peut aussi **`map`** et **`filter`** les ensembles.

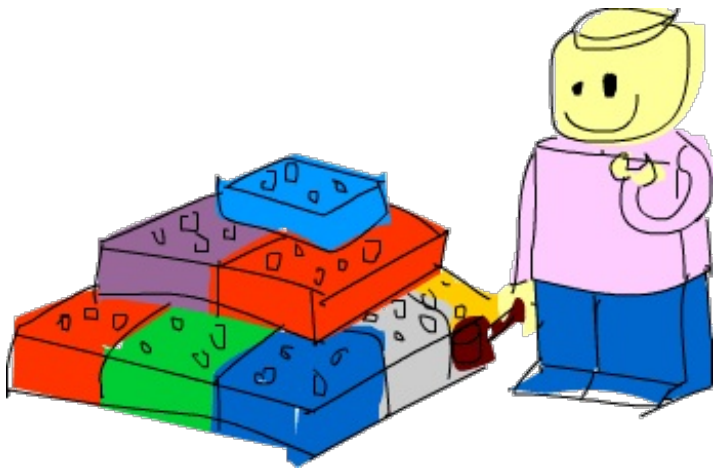
```
ghci> Set.filter odd $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,5,7]
ghci> Set.map (+1) $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,4,5,6,7,8]
```

Les ensembles sont souvent utilisés pour supprimer les doublons d'une liste en la transformant avec **`fromList`** en ensemble, puis en la reconvertissant en liste à l'aide de **`toList`**. La fonction **`nub`** de **`Data.List`** fait aussi ça, mais supprimer les doublons d'une liste est beaucoup plus rapide en convertissant la liste en ensemble puis en liste plutôt qu'en utilisant **`nub`**. Mais **`nub`** ne nécessite qu'une contrainte de classe **`Eq`** sur le type des éléments, alors que pour la transformer en ensemble, le type doit être membre de **`Ord`**.

```
ghci> let setNub xs = Set.toList $ Set.fromList xs
ghci> setNub "HEY WHATS CRACKALACKIN"
" ACEHIKLNRSTWY"
ghci> nub "HEY WHATS CRACKALACKIN"
"HEY WATSCRKLIN"
```

`setNub` est généralement plus rapide que **`nub`** sur des grosses listes, mais comme vous pouvez le voir, **`nub`** préserve l'ordre des éléments alors que **`setNub`** les mélange.

Créer nos propres modules



On vient de regarder des modules plutôt cool, mais comment crée-t-on nos propres modules ? Presque tous les langages de programmation vous permettent de découper votre code en plusieurs fichiers et Haskell également. Lorsqu'on programme, il est de bonne pratique de prendre des fonctions et des types qui partagent un but similaire et de les placer dans un module. Ainsi, vous pouvez facilement réutiliser ces fonctions plus tard dans d'autres programmes juste en important le module.

Voyons comment faire nos propres modules en créant un petit module fournissant des fonctions de calcul de volume et d'aire d'objets géométriques. Commençons par créer un fichier `Geometry.hs`.

On dit qu'un module *exporte* des fonctions. Cela signifie que quand j'importe un module, je peux utiliser les fonctions que celui-ci exporte. Il peut définir des fonctions que ses propres fonctions appellent en interne, mais on peut seulement voir celles qu'il a exportées.

Au début d'un module, on spécifie le nom du module. On a créé un fichier `Geometry.hs`, nous devrions donc nommer notre module `Geometry`. Puis, nous spécifions les fonctions qu'il exporte, et après cela, on peut commencer à écrire nos fonctions. Démarrons.

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where
```

Comme vous pouvez le voir, nous allons faire des aires et des volumes de sphères, de cubes et de pavés droits. Définissons nos fonctions :

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c

cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

De la géométrie élémentaire. Quelques choses à noter tout de même. Puisqu'un cube est un cas spécial de pavé droit, on a défini son aire et son volume comme ceux d'un pavé dont les côtés ont tous la même longueur. On a également défini la fonction auxiliaire `rectangleArea`, qui calcule l'aire d'un rectangle à partir des longueurs de ses côtés. C'est plutôt trivial puisqu'il s'agit d'une simple multiplication. Remarquez comme on l'utilise dans nos fonctions de ce module (dans `cuboidArea` et `cuboidVolume`), mais on ne l'exporte pas ! On souhaite que notre module présente des fonctions de calcul sur des objets en trois dimensions, donc on n'exporte pas `rectangleArea`.

Quand on crée un module, on n'exporte en général que les fonctions qui agissent en rapport avec l'interface de notre module, de manière à cacher l'implémentation. Si quelqu'un utilise notre module `Geometry`, il n'a pas à se soucier des fonctions que l'on n'a pas exportées. On peut décider de changer ces fonctions complètement ou de les effacer dans une nouvelle version (on pourrait supprimer `rectangleArea` et utiliser `*` à la place) et personne ne s'en souciera parce qu'on ne les avait pas exportées.

Pour utiliser notre module, on fait juste :

```
import Geometry
```

`Geometry.hs` doit tout de même être dans le même dossier que le programme qui souhaite l'importer.

Les modules peuvent aussi être organisés hiérarchiquement. Chaque module peut avoir un nombre de sous-modules et eux-mêmes peuvent avoir leurs sous-modules. Découpons ces fonctions de manière à ce que `Geometry` soit un module avec trois sous-modules, un pour chaque type d'objet.

D'abord, créons un dossier `Geometry`. Attention à la majuscule à G. Dans ce dossier, placez trois dossiers : `Sphere.hs`, `Cuboid.hs` et `Cube.hs` (NDT : "Cuboid" signifie pavé droit). Voici ce que les fichiers contiennent :

`Sphere.hs`

```
module Geometry.Sphere
( volume
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

`Cuboid.hs`

```
module Geometry.Cuboid
( volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c

area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

`Cube.hs`

```
module Geometry.Cube
( volume
, area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

Parfait ! Tout d'abord, nous avons `Geometry.Sphere`. Remarquez comme on l'a placé dans le dossier `Geometry` puis nommé `Geometry.Sphere`. Idem pour le pavé. Remarquez aussi comme dans chaque sous-module, nous avons défini des fonctions avec le même nom. On peut le faire car les modules sont séparés. On veut utiliser des fonctions de `Geometry.Cuboid` dans `Geometry.Cube`, mais on ne peut pas simplement `import Geometry.Cuboid` parce que ce module exporte des fonctions ayant le même nom que celles de `Geometry.Cube`. C'est pourquoi l'import est qualifié, et tout va bien.

Donc maintenant, si l'on se trouve dans un fichier qui se trouve au même niveau que le dossier `Geometry`, on peut par exemple :

```
import Geometry.Sphere
```

Et maintenant, on peut utiliser `area` et `volume`, qui nous donneront l'aire et le volume d'une sphère. Et si l'on souhaite jongler avec deux ou plus de ces modules, on doit utiliser des imports qualifiés car ils exportent des fonctions avec des noms identiques. Tout simplement :

```
import qualified Geometry.Sphere as Sphere
```

```
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

Et maintenant, on peut appeler `Sphere.area`, `Sphere.volume`, `Cuboid.area`, etc. et chacune calculera l'aire ou le volume de l'objet correspondant.

La prochaine fois que vous vous retrouvez en train d'écrire un fichier très gros avec plein de fonctions, essayez de voir lesquelles partagent un but commun et si vous pouvez les regrouper dans un module. Vous n'aurez plus qu'à importer ce module si vous souhaitez réutiliser ces fonctionnalités dans un autre programme.

⬅ Fonctions d'ordre supérieur | Table des matières | Créer nos propres types et classes de types ➡

Créer nos propres types et classes de types

⬅ Modules | Table des matières | Entrées et sorties ➡

Dans les chapitres précédents, nous avons vu quelques types et classes de types qui existent en Haskell. Dans ce chapitre, nous verrons comment créer les nôtres et les mettre en pratique !

Introduction aux types de données algébriques

Jusqu'ici, nous avons croisé beaucoup de types de données. `Bool`, `Int`, `Char`, `Maybe`, etc. Mais comment créer les nôtres ? Eh bien, un des moyens consiste à utiliser le mot-clé **data** pour définir un type. Voyons comment le type `Bool` est défini dans la bibliothèque standard.

```
data Bool = False | True
```

`data` signifie qu'on crée un nouveau type de données. La partie avant le `=` dénote le type, ici `Bool`. Les choses après le `=` sont des **constructeurs de valeurs**. Ils spécifient les différentes valeurs que peut prendre ce type. Le `|` se lit comme un *ou*. On peut donc lire cette déclaration : le type `Bool` peut avoir pour valeur `True` ou `False`. Le nom du type tout comme les noms des constructeurs de valeurs doivent commencer par une majuscule.

De manière similaire, on peut imaginer `Int` défini ainsi :

```
data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ... | 2147483647
```

Le premier et le dernier constructeurs sont les valeurs minimales et maximales d'un `Int`. En vrai le type n'est pas défini ainsi, les points de suspension cachent l'omission d'un paquet énorme de nombres, donc ceci est juste à titre illustratif.

Maintenant, imaginons comment l'on représenterait une forme en Haskell. Une possibilité serait d'utiliser des tuples. Un cercle pourrait être `(43.1, 55.0, 10.4)` où les deux premières composantes serait les coordonnées du centre, et la troisième le rayon. Ça a l'air raisonnable, mais ça pourrait tout aussi bien représenter un vecteur 3D ou je ne sais quoi. Une meilleure solution consiste à créer notre type pour représenter les données. Disons qu'une forme puisse être un cercle ou un rectangle. Voilà :

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

Qu'est-ce que c'est que ça ? Pensez-y ainsi. Le constructeur de valeurs `Circle` a trois champs, qui sont tous des `Float`. Ainsi, lorsqu'on écrit un constructeur de valeurs, on peut optionellement ajouter des types à la suite qui définissent les valeurs qu'il peut contenir. Ici, les deux premiers champs sont les coordonnées du centre, le troisième est son rayon. Le constructeur de valeurs `Rectangle` a quatre champs qui acceptent des `Float`. Les deux premiers sont les coordonnées du coin supérieur gauche, et les deux autres celles du coin inférieur droit.

Quand je dis champ, il s'agit en fait de paramètres. Les constructeurs de valeurs sont ainsi des fonctions qui retournent ultimement un type de données. Regardons les signatures de type de ces deux constructeurs de valeurs.

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Cool, donc les constructeurs de valeurs sont des fonctions comme les autres. Qui l'eût cru ? Créons une fonction qui prend une forme et retourne sa surface.

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

La première chose notable, c'est la déclaration de type. Elle dit que la fonction prend un `Shape` et retourne un `Float`. On n'aurait pas pu écrire une déclaration de type `Circle -> Float` parce que `Circle` n'est pas un type, alors que `Shape` en est un. Tout comme on ne peut pas écrire une fonction qui a pour déclaration de type `True -> Int`. La prochaine chose remarquable, c'est le filtrage par motif sur les constructeurs de valeurs. Nous l'avons en fait déjà fait (tout le temps à vrai dire), lorsque l'on filtrait des valeurs contre `[]` ou `False` ou `5`, seulement ces valeurs n'avaient pas de champs. On écrit seulement le constructeur, puis on lie ses champs à des noms. Puisqu'on s'intéresse au rayon, on se fiche des autres champs, qui n'indiquent que la position du cercle.

```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
```

Yay, ça marche ! Mais si on essaie d'afficher `Circle 10 20 5` dans l'invite, on obtient une erreur. C'est parce qu'Haskell ne sait pas (encore) comment afficher ce type de données sous une forme littérale. Rappelez-vous, quand on essaie d'afficher une valeur dans l'invite, Haskell exécute la fonction `show` sur cette valeur pour obtenir une représentation en chaîne de caractères, puis affiche celle-ci dans le terminal. Pour rendre `Shape` membre de la classe de types `Show`, on peut le modifier ainsi :

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float deriving (Show)
```

On ne s'intéressera pas trop à la dérivation pour l'instant. Disons seulement qu'en ajoutant `deriving (Show)` à la fin d'une déclaration *data*, Haskell rend magiquement ce type membre de la classe de types `Show`. Ainsi, on peut à présent faire :

```
ghci> Circle 10 20 5
Circle 10.0 20.0 5.0
ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
```

Les constructeurs de valeurs sont des fonctions, on peut donc les mapper, les appliquer partiellement, etc. Si l'on veut créer une liste de cercles concentriques de différents rayons, on peut faire :

```
ghci> map (Circle 10 20) [4,5,6,6]
[Circle 10.0 20.0 4.0,Circle 10.0 20.0 5.0,Circle 10.0 20.0 6.0,Circle 10.0 20.0 6.0]
```

Notre type de données est satisfaisant, bien qu'il pourrait être amélioré. Créons un type de données intermédiaire qui définit un point dans l'espace bidimensionnel. On pourra l'utiliser pour rendre nos formes plus compréhensibles.

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
```

Remarquez qu'en définissant un point, on a utilisé le même nom pour le type de données et pour le constructeur de valeurs. Ça n'a pas de sens particulier, mais il est assez usuel de donner le même nom lorsqu'un type n'a qu'un seul constructeur de valeurs. À présent, le `Circle` a deux champs, un de type `Point` et un autre de type `Float`. Cela simplifie la compréhension de ce qu'ils représentent. Idem pour le rectangle. On doit maintenant ajuster `surface` pour refléter ces changements.

```
surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

On a seulement changé les motifs. Dans le motif du cercle, on ignore complètement le point. Dans celui du rectangle, on utilise des motifs imbriqués pour récupérer les champs des points. Si l'on voulait référencer le point lui-même pour une raison quelconque, on aurait pu utiliser des motifs nommés.

```
ghci> surface (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> surface (Circle (Point 0 0) 24)
1809.5574
```

Et pourquoi pas une fonction qui bouge une forme ? Elle prend une forme, une quantité sur l'axe x et sur l'axe y, et retourne une forme aux mêmes dimensions, mais déplacée selon ces quantités.

```
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b = Rectangle (Point (x1+a) (y1+b)) (Point (x2+a) (y2+b))
```

Plutôt simple. On ajoute simplement les déplacements aux coordonnées correspondantes.

```
ghci> nudge (Circle (Point 34 34) 10) 5 10
Circle (Point 39.0 44.0) 10.0
```

Si on ne veut pas manipuler directement les points, on peut créer des fonctions auxiliaires qui créent des formes d'une taille donnée à l'origine, puis les déplacer.

```
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r
```

```
baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle (Point 0 0) (Point width height)
ghci> nudge (baseRect 40 100) 60 23
Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
```

Vous pouvez bien sûr exporter vos types de données de vos modules. Pour ce faire, ajoutez simplement les types que vous souhaitez exporter au même endroit que les fonctions à exporter, puis ouvrez des parenthèses et spécifiez les constructeurs de valeurs que vous voulez exporter, séparés par des virgules. Si vous voulez exporter tous les constructeurs, vous pouvez écrire `..`.

Si on voulait exporter les fonctions et types définis ici dans un module, on pourrait commencer ainsi :

```
module Shapes
( Point(..)
, Shape(..)
, surface
, nudge
, baseCircle
, baseRect
) where
```

En faisant **`Shape(..)`**, on exporte tous les constructeurs de valeurs de **`Shape`**, donc quiconque importe le module peut créer des formes à l'aide de **`Rectangle`** et **`Circle`**. C'est équivalent à **`Shape (Rectangle, Circle)`**.

On pourrait aussi choisir de ne pas exporter de constructeurs de valeurs pour **`Shape`** en écrivant simplement **`Shape`** dans la déclaration d'export. Ainsi, quelqu'un qui voudrait créer une forme serait obligé de le faire en passant par les fonctions **`baseCircle`** et **`baseRect`**. **`Data.Map`** utilise cette approche. Vous ne pouvez pas créer de map en faisant **`Map.Map [(1,2),(3,4)]`** parce qu'il n'exporte pas ce constructeur de valeurs. Cependant, vous pouvez en créer à l'aide de fonctions auxiliaires comme **`Map.fromList`**. Souvenez-vous, les constructeurs de valeurs sont juste des fonctions qui prennent des champs en paramètres et retournent une valeur d'un certain type (comme **`Shape`**). Donc quand on choisit de ne pas les exporter, on empêche seulement les personnes qui importent notre module d'utiliser ces fonctions, mais si d'autres fonctions exportées retournent un type, on peut les utiliser pour créer des valeurs de ce type de données.

Ne pas exporter les constructeurs de valeurs d'un type de données le rend plus abstrait en cachant son implémentation. Également, quiconque utilise ce module ne peut pas filtrer par motif sur les constructeurs de valeurs.

Syntaxe des enregistrements

OK, on vient de nous donner la tâche de créer un type de données pour décrire une personne. Les informations à stocker pour décrire une personne sont : prénom, nom de famille, âge, poids, numéro de téléphone, et parfum de glace préféré. Je ne sais pas pour vous, mais c'est tout ce que je souhaite savoir à propos d'une personne. Allons-y !

```
data Person = Person String String Int Float String String deriving (Show)
```

Ok. Le premier champ est le prénom, le deuxième le nom, etc. Créons une personne.

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

Plutôt cool, bien qu'un peu illisible. Comment faire une fonction pour récupérer juste une information sur la personne ? Une pour son prénom, une pour son nom, etc. Nous devrions les définir ainsi :

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height
```

```haskell
phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

Wow ! Personnellement, je n'ai pas trop apprécié écrire ça ! En dépit d'être très encombrante et ENNUYEUSE à écrire, cette méthode fonctionne.

```haskell
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> firstName guy
"Buddy"
ghci> height guy
184.2
ghci> flavor guy
"Chocolate"
```

Il doit y avoir un meilleur moyen, vous vous dites ! Eh bien non, il n'y en a pas, désolé.

Non, je blague, il y en a un. Hahaha ! Les créateurs d'Haskell étaient très intelligents et ont anticipé ce scénario. Ils ont inclus une version alternative de l'écriture des types de données. Voici comment l'on pourrait obtenir les mêmes fonctionnalités avec une syntaxe d'enregistrements.

```haskell
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

Donc, plutôt que de seulement nommer les types des champs les uns après les autres et de les séparer par des espaces, on peut utiliser des accolades. On écrit d'abord le nom du champ, par exemple `firstName`, puis un `::` (aussi appelé Paamayim Nekudotayim, haha) puis on spécifie son type. Le type de données résultant est exactement le même. Le principal avantage est que cela crée les fonctions pour examiner chaque champ du type de données. En utilisant la syntaxe des enregistrements, Haskell a automatiquement créé ces fonctions : `firstName`, `lastName`, `age`, `height`, `phoneNumber` et `flavor`.

```haskell
ghci> :t flavor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

Il y a également un autre avantage à utiliser la syntaxe des enregistrements. Lorsqu'on dérive `Show` pour le type, il est affiché différemment lorsqu'on utilise la syntaxe des enregistrements pour définir et instancier le type. Mettons qu'on a un type qui représente des voitures. On souhaite garder une trace de la compagnie qui l'a créée, le nom du modèle et l'année de production. Regardez.

```haskell
data Car = Car String String Int deriving (Show)
```

```haskell
ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
```

Si on le définit à l'aide de la syntaxe des enregistrements, on peut créer une nouvelle voiture comme suit.

```haskell
data Car = Car {company :: String, model :: String, year :: Int} deriving (Show)
```

```haskell
ghci> Car {company="Ford", model="Mustang", year=1967}
Car {company = "Ford", model = "Mustang", year = 1967}
```

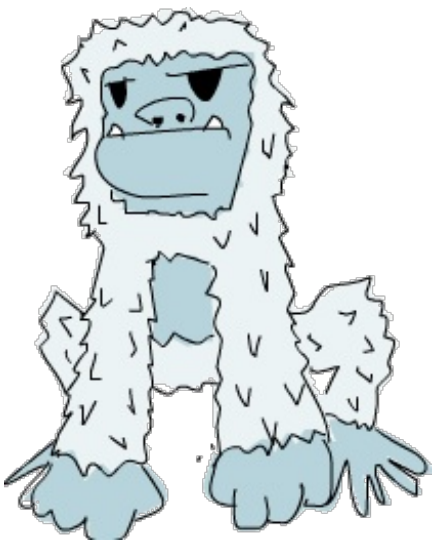
À la création de la voiture, on n'a pas nécessairement à mettre les champs dans le bon ordre, du moment qu'on les liste tous. Alors que sans la syntaxe des enregistrements, l'ordre importe.

Utilisez cette syntaxe lorsqu'un constructeur a plusieurs champs et qu'il n'est pas évident de prédire lequel correspond à quoi. Si l'on crée un type de données vecteur 3D en faisant `data Vector = Vector Int Int Int`, il est plutôt évident que les champs sont les trois composantes du vecteur. Par contre, pour `Person` ou `Car`, c'était plus compliqué et on a gagné à utiliser la syntaxe des enregistrements.

Paramètres de types

Un constructeur de valeurs peut prendre plusieurs valeurs comme paramètres et retourner une nouvelle valeur. Par exemple, le constructeur `Car` prend trois valeurs et produit une valeur de type `Car`. De manière similaire, les **constructeurs de types** peuvent prendre des types en paramètres pour créer de nouveaux types. Ça peut sembler un peu trop méta au premier abord, mais ce n'est pas si compliqué. Si vous êtes familier avec les templates C++, vous verrez des parallèles. Pour se faire une bonne image de l'utilisation des paramètres de types en action, regardons comment un des types que nous avons déjà rencontré est implémenté.

```
data Maybe a = Nothing | Just a
```



Le `a` est un paramètre de type. Et puisqu'il y a un paramètre de type impliqué, on dit que `Maybe` est un constructeur de type. En fonction de ce que l'on veut que ce type de données contienne lorsqu'il ne vaut pas `Nothing`, ce constructeur de types peut éventuellement construire un type `Maybe Int`, `Maybe Car`, `Maybe String`, etc. Aucune valeur ne peut avoir pour type `Maybe`, car ce n'est pas un type per se, mais un constructeur de types. Pour pouvoir être un type réel qui peut avoir une valeur, il doit avoir tous ses paramètres remplis.

Donc, si l'on passe `Char` en paramètre de type à `Maybe`, on obtient un type `Maybe Char`. La valeur `Just 'a'` a pour type `Maybe Char` par exemple.

Vous ne le savez peut-être pas, mais on a utilisé un type qui a un paramètre de type avant même d'utiliser `Maybe`. Ce type est le type des listes. Bien qu'il y ait un peu de sucre syntaxique en jeu, le type liste prend un paramètre et produit un type concret. Des valeurs peuvent avoir pour type `[Int]`, `[Char]`, `[[String]]`, mais aucune valeur ne peut avoir pour type `[]`.

Jouons un peu avec le type `Maybe`.

```
ghci> Just "Haha"
Just "Haha"
ghci> Just 84
Just 84
ghci> :t Just "Haha"
Just "Haha" :: Maybe [Char]
ghci> :t Just 84
Just 84 :: (Num t) => Maybe t
ghci> :t Nothing
Nothing :: Maybe a
ghci> Just 10 :: Maybe Double
Just 10.0
```

Les paramètres de types sont utiles parce qu'on peut créer différents types en fonction de la sorte de types qu'on souhaite que notre type de données contienne. Quand on fait `:t Just "Haha"`, le moteur d'inférence de types se rend compte que le type doit être `Maybe [Char]`, parce que le `a` dans le `Just a` est une chaîne de caractères, donc le `a` de `Maybe a` doit aussi être une chaîne de caractères.

Remarquez que le type de `Nothing` est `Maybe a`. Son type est polymorphique. Si une fonction nécessite un `Maybe Int` en paramètre, on peut lui donner `Nothing`, parce que `Nothing` ne contient pas de valeur de toute façon, donc peu importe. Le type `Maybe a` peut se comporter comme `Maybe Int` s'il le faut, tout comme `5` peut se comporter comme un `Int` ou un `Double`. De façon similaire, le type de la liste vide est `[a]`. Une liste vide peut être une liste de quoi que ce soit. C'est pourquoi on peut faire `[1, 2, 3] ++ []` et `["ha", "ha", "ha"] ++ []`.

Utiliser les paramètres de types est très bénéfique, mais seulement quand les utiliser a un sens. Généralement, on les utilise quand notre type de données fonctionne sans se soucier du type de ce qu'il contient en lui, comme pour notre type `Maybe a`. Si notre type se comporte comme une sorte de boîte, il est bien de les utiliser. On pourrait changer notre type de données `Car` de ceci :

```
data Car = Car { company :: String
               , model :: String
               , year :: Int
               } deriving (Show)
```

en cela :

```
data Car a b c = Car { company :: a
                     , model :: b
                     , year :: c
                     } deriving (Show)
```

Mais y gagnerait-on vraiment ? La réponse est : probablement pas, parce qu'on finirait par définir des fonctions qui ne fonctionnent que sur le type `Car String String Int`. Par exemple, vu notre première définition de `Car`, on pourrait écrire une fonction qui affiche les propriétés de la voiture avec un joli petit texte.

```
```

```
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

```
ghci> let stang = Car {company="Ford", model="Mustang", year=1967}
ghci> tellCar stang
"This Ford Mustang was made in 1967"
```

Quelle jolie petite fonction ! La déclaration de type est mignonne et fonctionne bien. Maintenant, si `Car` était `Car a b c` ?

```
tellCar :: (Show a) => Car String String a -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

Nous devrions forcer cette fonction à prendre un type `Car` tel que `(Show a) => Car String String a`. Vous pouvez constater que la signature de type est plus compliquée, et le seul avantage qu'on en tire serait qu'on pourrait utiliser n'importe quel type instance de la classe de types `Show` pour `c`.

```
ghci> tellCar (Car "Ford" "Mustang" 1967)
"This Ford Mustang was made in 1967"
ghci> tellCar (Car "Ford" "Mustang" "nineteen sixty seven")
"This Ford Mustang was made in \"nineteen sixty seven\""
ghci> :t Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967 :: (Num t) => Car [Char] [Char] t
ghci> :t Car "Ford" "Mustang" "nineteen sixty seven"
Car "Ford" "Mustang" "nineteen sixty seven" :: Car [Char] [Char] [Char]
```

Dans la vie réelle cependant, on finirait par utiliser `Car String String Int` la plupart du temps, et il semblerait que paramétrer le type `Car` ne vaudrait pas le coup. On utilise généralement les paramètres de types lorsque le type contenu dans les divers constructeurs de valeurs du type de données n'est pas vraiment important pour que le type fonctionne. Une liste de choses est une liste de choses, peu importe ce que les choses sont, ça marche. Si on souhaite sommer une liste de nombre, on peut spécifier au dernier moment que la fonction de sommage attend une liste de nombres. De même pour `Maybe`. `Maybe` représente une option qui est soit de n'avoir rien, soit d'avoir quelque chose. Peu importe ce que le type de cette chose est.

Un autre exemple de type paramétré que nous avons déjà rencontré est `Map k v` de `Data.Map`. Le `k` est le type des clés, le `v` le type des valeurs. C'est un bon exemple d'endroit où les types paramétrés sont très utiles. Avoir des maps paramétrées nous permet de créer des maps de n'importe quel type vers n'importe quel autre type, du moment que le type de la clé soit membre de la classe de types `Ord`. Si nous souhaitions définir un type de map, on pourrait ajouter la contrainte de classe dans la déclaration *data* :

```
data (Ord k) => Map k v = ...
```

Cependant, il existe une très forte convention en Haskell qui est de **ne jamais ajouter de contraintes de classe à une déclaration de données**. Pourquoi ? Eh bien, parce que le bénéfice est minimal, mais on se retrouve à écrire plus de contraintes de classes, même lorsqu'elles ne sont pas nécessaires. Que l'on mette ou non, la contrainte `Ord k` dans la déclaration *data* de `Map k v`, on aura à écrire la contrainte dans les fonctions qui supposent un ordre sur les clés de toute façon. Mais, si l'on ne met pas la contrainte dans la déclaration *data*, alors on n'aura pas à mettre `(Ord k) =>` dans les déclarations de types des fonctions qui n'ont pas besoin de cette contrainte pour fonctionner. Un exemple d'une telle fonction est `toList`, qui prend un mapping et le convertit en liste associative. Sa signature de type est `toList :: Map k a -> [(k, a)]`. Si `Map k v` avait une contrainte de classe dans sa déclaration *data*, le type de `ToList` devrait être `toList :: (Ord k) => Map k a -> [(k, a)]`, alors que cette fonction ne fait aucune comparaison de clés selon leur ordre.

Conclusion : ne mettez pas de contraintes de classe dans les déclarations *data* même lorsqu'elles ont l'air sensées, parce que de toute manière, vous devrez les écrire dans les fonctions qui en dépendent.

Implémentons un type de vecteur 3D et ajoutons-y quelques opérations. Nous utiliserons un type paramétré afin qu'il supporte plusieurs types numériques, bien qu'en général on n'en utilise qu'un seul.

```
data Vector a = Vector a a a deriving (Show)

vplus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)

vectMult :: (Num t) => Vector t -> t -> Vector t
(Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)

scalarMult :: (Num t) => Vector t -> Vector t -> t
(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

`vplus` somme deux vecteurs. Deux vecteurs sont sommés en sommant leurs composantes deux à deux. `scalarMult` est le produit scalaire de deux vecteurs et

`vectMult` permet de multiplier un vecteur par un scalaire. Ces fonctions peuvent opérer sur des types comme `Vector Int`, `Vector Integer`, `Vector Float`, à condition que le `a` de `Vector a` soit de la classe `Num`. Également, si vous examinez les déclarations de type des fonctions, vous verrez qu'elles n'opèrent que sur des vecteurs de même type et que les scalaires doivent également être du type contenu dans les vecteurs. Remarquez qu'on n'a pas mis de contrainte `Num` dans la déclaration *data*, puisqu'on a eu à l'écrire dans chaque fonction qui en dépendait de toute façon.

Une fois de plus, il est très important de distinguer le constructeur de types du constructeur de valeurs. Lorsqu'on déclare un type, le nom à gauche du `=` est le constructeur de types, et les constructeurs situés après (séparés par des `|`) sont des constructeurs de valeurs. Donner à une fonction le type `Vector t t t -> Vector t t t -> t` serait faux, parce que l'on doit donner des types dans les déclarations de types, le constructeur de **types** vecteurs ne prend qu'un paramètre, alors que le constructeur de valeurs en prend trois. Jouons avec nos vecteurs.

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
ghci> Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 12 9 19
ghci> Vector 3 9 7 `vectMult` 10
Vector 30 90 70
ghci> Vector 4 9 5 `scalarMult` Vector 9.0 2.0 4.0
74.0
ghci> Vector 2 9 3 `vectMult` (Vector 4 9 5 `scalarMult` Vector 9 2 4)
Vector 148 666 222
```

Instances dérivées

Dans la section Classes de types 101, nous avons vu les bases des classes de types. Nous avons dit qu'une classe de types est une sorte d'interface qui définit un comportement. Un type peut devenir une **instance** d'une classe de types s'il supporte ce comportement. Par exemple : le type `Int` est une instance d'`Eq` parce que cette classe définit le comportement de ce qui peut être testé pour l'égalité. Et puisqu'on peut tester l'égalité de deux entiers, `Int` est membre de la classe `Eq`. La vraie utilité vient des fonctions qui agissent comme l'interface d'`Eq`, à savoir `==` et `/=`. Si un type est membre d'`Eq`, on peut utiliser les fonctions `==` et `/=` avec des valeurs de ce type. C'est pourquoi des expressions comme `4 == 4` et `"foo" /= "bar"` sont correctement typées.

Nous avons aussi mentionné qu'elles sont souvent confondues avec les classes de langages comme Java, Python, C++ et autres, ce qui sème la confusion dans l'esprit de beaucoup de gens. Dans ces langages, les classes sont des patrons à partir desquels sont créés des objets qui contiennent un état et peuvent effectuer des actions. Les classes de types sont plutôt comme des interfaces. On ne crée pas de données à partir de classes de types. Plutôt, on crée d'abord notre type de données, puis on se demande pour quoi il peut se faire passer. S'il peut être comparé pour l'égalité, on le rend instance de la classe `Eq`. S'il peut être ordonné, on le rend instance de la classe `Ord`.

Dans la prochaine section, nous verrons comment créer manuellement nos instances d'une classe de types en implémentant les fonctions définies par cette classe. Pour l'instant, voyons comment Haskell peut magiquement faire de nos types des instances de n'importe laquelle des classes de types suivantes : `Eq`, `Ord`, `Enum`, `Bounded`, `Show` et `Read`. Haskell peut dériver le comportement de nos types dans ces contextes si l'on utilise le mot clé *deriving* lors de la création du type de données.

Considérez ce type de données :

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     }
```

Il décrit une personne. Posons comme hypothèse que deux personnes n'ont jamais les mêmes nom, prénom et âge. Maintenant, si l'on a un enregistrement pour deux personnes, est-ce que cela a un sens de vérifier s'il s'agit de la même personne ? Bien sûr. On peut essayer de voir si les enregistrements sont les mêmes ou non. C'est pourquoi il serait sensé que ce type soit membre de la classe `Eq`. Nous allons dériver cette instance.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq)
```

Lorsqu'on dérive l'instance `Eq` pour un type, puis qu'on essaie de comparer deux valeurs de ce type avec `==` ou `/=`, Haskell va regarder si les deux constructeurs sont égaux (ici il n'y en a qu'un possible cependant), puis va tester si les données contenues dans les valeurs sont égales deux à deux en testant chaque paire de champ avec `==`. Ainsi, il y a un détail important, qui est que les types des champs doivent aussi êtres membres de la classe `Eq`. Puisque `String` et `Int` le sont, tout va bien. Testons notre instance d'`Eq`.

```
ghci> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> let adRock = Person {firstName = "Adam", lastName = "Horovitz", age = 41}
ghci> let mca = Person {firstName = "Adam", lastName = "Yauch", age = 44}
```

```
ghci> mca == adRock
False
ghci> mikeD == adRock
False
ghci> mikeD == mikeD
True
ghci> mikeD == Person {firstName = "Michael", lastName = "Diamond", age = 43}
True
```

Bien sûr, puisque `Person` est maintenant dans `Eq`, on peut utiliser comme `a` pour toute fonction ayant une contrainte de classe `Eq a` dans sa signature, comme `elem`.

```
ghci> let beastieBoys = [mca, adRock, mikeD]
ghci> mikeD `elem` beastieBoys
True
```

Les classes de types `Show` et `Read` sont pour les choses qui peuvent être converties respectivement vers et depuis une chaîne de caractères. Comme avec `Eq`, si un constructeur de types a des champs, leur type doit aussi être membre de `Show` ou `Read` si on veut faire de notre type une instance de l'une de ces classes. Faisons de notre type de données `Person` un membre de `Show` et de `Read`.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq, Show, Read)
```

Maintenant, on peut afficher une personne dans le terminal.

```
ghci> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> mikeD
Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> "mikeD is: " ++ show mikeD
"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\", age = 43}"
```

Si nous avions voulu afficher une personne dans le terminal avant de rendre `Person` membre de `Show`, Haskell se serait plaint du fait qu'il ne sache pas représenter une personne comme une chaîne de caractères. Mais maintenant qu'on a dérivé `Show`, il sait comment faire.

`Read` est en gros l'inverse de `Show`. `Show` convertit des valeurs de notre type en des chaînes de caractères, `Read` convertit des chaîne de caractères en valeurs de notre type. Souvenez-vous cependant, lorsqu'on avait utilisé la fonction `read`, on avait dû annoter explicitement le type qu'on désirait. Sans cela, Haskell ne sait pas vers quel type on veut convertir.

```
ghci> read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" :: Person
Person {firstName = "Michael", lastName = "Diamond", age = 43}
```

Si on utilise le résultat de `read` dans un calcul plus élaboré, Haskell peut inférer le type qu'on attend, et l'on n'a alors pas besoin d'annoter le type.

```
ghci> read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" == mikeD
True
```

On peut aussi lire des types paramétrés, et il faut alors annoter le type avec les paramètres complétés. Ainsi, on ne peut pas faire `read "Just 't'" :: Maybe a`, mais on peut faire `read "Just 't'" :: Maybe Char`.

On peut dériver des instances la classe de types `Ord`, qui est pour les types dont les valeurs peuvent être ordonnées. Si l'on compare deux valeurs du même type ayant été construites par deux constructeurs différents, la valeur construite par le constructeur défini en premier sera considérée plus petite. Par exemple, considérez le type `Bool`, qui peut avoir pour valeur `False` ou `True`. Pour comprendre comment il fonctionne lorsqu'il est comparé, on peut l'imaginer défini comme :

```
data Bool = False | True deriving (Ord)
```

Puisque le constructeur de valeurs `False` est spécifié avant le constructeur de valeurs `True`, on peut considérer que `True` est plus grand que `False`.

```
ghci> True `compare` False
GT
ghci> True > False
True
ghci> True < False
```

```
False
```

Dans le type de données `Maybe a`, le constructeur de valeurs `Nothing` est spécifié avant le constructeur de valeurs `Just`, donc une valeur `Nothing` est toujours plus petite qu'une valeur `Just something`, même si ce *something* est moins un milliard de milliards. Mais si l'on compare deux valeurs `Just`, alors Haskell compare ce qu'elles contiennent.

```
ghci> Nothing < Just 100
True
ghci> Nothing > Just (-49999)
False
ghci> Just 3 `compare` Just 2
GT
ghci> Just 100 > Just 50
True
```

Mais on ne peut pas faire `Just (*3) > Just (*2)`, parce que `(*3)` et `(*2)` sont des fonctions, qui ne sont pas des instances d'`Ord`.

On peut facilement utiliser des types de données algébriques pour créer des énumérations, et les classes de types `Enum` et `Bounded` nous aident dans la tâche. Considérez les types de données suivants :

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Puisque tous les constructeurs de valeurs sont nullaires (ils ne prennent pas de paramètres, ou champs), on peut rendre le type membre de la classe `Enum`. La classe de types `Enum` est pour les choses qui ont des prédécesseurs et des successeurs. On peut aussi le rendre membre de la classe `Bounded`, qui est pour les choses avec une plus petite valeur et une plus grande valeur. Tant qu'on y est, faisons-en aussi une instance des autres classes qu'on a vues.

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
           deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

Puisque le type est membre des classes `Show` et `Read`, on peut le convertir vers et depuis des chaînes de caractères.

```
ghci> Wednesday
Wednesday
ghci> show Wednesday
"Wednesday"
ghci> read "Saturday" :: Day
Saturday
```

Puisqu'il est membre des classes `Eq` et `Ord`, on peut comparer et tester l'égalité de deux jours.

```
ghci> Saturday == Sunday
False
ghci> Saturday == Saturday
True
ghci> Saturday > Friday
True
ghci> Monday `compare` Wednesday
LT
```

Il est aussi membre de `Bounded`, donc on peut demander le plus petit jour et le plus grand jour.

```
ghci> minBound :: Day
Monday
ghci> maxBound :: Day
Sunday
```

C'est aussi une instance d'`Enum`. On peut obtenir le prédécesseur et le successeur d'un jour, et créer une progression de jours !

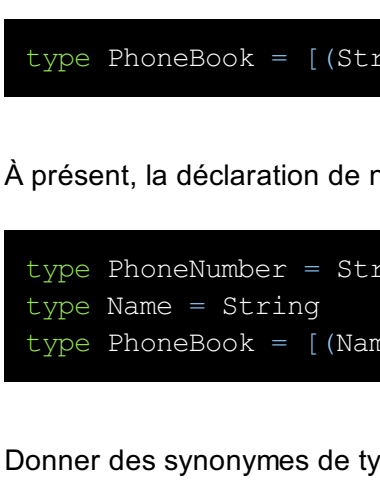
```
ghci> succ Monday
Tuesday
ghci> pred Saturday
Friday
ghci> [Thursday .. Sunday]
[Thursday,Friday,Saturday,Sunday]
ghci> [minBound .. maxBound] :: [Day]
[Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday]
```

C’est plutôt génial.

Synonymes de types

Précédemment, on a mentionné qu’en écrivant des types, les types `[Char]` et `String` étaient équivalents et interchangeables. Ceci est implémenté à l’aide des **synonymes de types**. Les synonymes de types ne font rien per se, il s’agit simplement de donner des noms différents au même type afin que la lecture du code ou de la documentation ait plus de sens pour le lecteur. Voici comment la bibliothèque standard définit `String` comme un synonyme de `[Char]`.

```
type String = [Char]
```



On a introduit le mot-clé *type*. Le mot clé peut être déroutant pour certains, puisqu’on ne crée en fait rien de nouveau (on faisait cela avec le mot-clé *data*), mais on crée seulement un synonyme pour un type déjà existant.

Si l’on crée une fonction qui convertit une chaîne de caractères en majuscules et qu’on l’appelle `toUpperString`, on peut lui donner comme déclaration de type `toUpperString :: [Char] -> [Char]` ou `toUpperString :: String -> String`. Ces deux déclarations sont en effet identiques, mais la dernière est plus agréable à lire.

Quand on utilisait le module `Data.Map`, on a commencé par représenter un carnet téléphonique comme une liste associative avant de le représenter comme une map. Comme nous l’avions alors vu, une liste associative est une liste de paires clé-valeur. Regardons le carnet que nous avions alors.

```
phoneBook :: [(String,String)]
phoneBook =
    [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

On voit que le type de `phoneBook` est `[(String, String)]`. Cela nous indique que c’est une liste associative qui mappe des chaînes de caractères vers des chaînes de caractères, mais pas grand chose de plus. Créons un synonyme de types pour communiquer plus d’informations dans la déclaration de type.

```
type PhoneBook = [(String,String)]
```

À présent, la déclaration de notre carnet peut être `phoneBook :: PhoneBook`. Créons aussi un synonyme pour `String`.

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
```

Donner des synonymes de types au type `String` est pratique courante chez les programmeurs en Haskell lorsqu’ils souhaitent indiquer plus d’information à propos des chaînes de caractères qu’ils utilisent dans leur programme et ce qu’elles représentent.

À présent, lorsqu’on implémente une fonction qui prend un nom et un nombre, et cherche si cette combinaison de nom et de numéro est dans notre carnet téléphonique, on peut lui donner une description de type très jolie et descriptive.

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name,pnumber) `elem` pbook
```

Si nous n’avions pas utilisé de synonymes de types, notre fonction aurait pour type `String -> String -> [(String,String)] -> Bool`. Ici, la déclaration tirant parti des synonymes de types est plus facile à lire. Cependant, n’en faites pas trop. On introduit les synonymes de types pour soit décrire ce que des types existants représentent dans nos fonctions (et ainsi les déclarations de types de nos fonctions deviennent de meilleures documentations), soit lorsque quelque chose a un type assez long et répété souvent (comme `[(String, String)]`) et représente quelque chose de spécifique dans le contexte de nos fonctions.

Les synonymes de types peuvent aussi être paramétrés. Si on veut un type qui représente le type des listes associatives de façon assez générale pour être utilisé quelque que soit le type des clés et des valeurs, on peut faire :

```
type AssocList k v = [(k,v)]
```

Maintenant, une fonction qui récupère la valeur associée à une clé dans une liste associative peut avoir pour type `(Eq k) => k -> AssocList k v -> Maybe v`. `AssocList` est un constructeur de types qui prend deux types et produit un type concret, comme

`AssocList Int String` par exemple.

Fonzie dit : Hey ! Quand je parle de *types concrets*, je veux dire genre des types appliqués complètement comme `Map Int String`, ou si on se frotte à une de ces fonctions polymorphes, `[a]` ou `(Ord a) => Maybe a` et tout ça. Et tu vois, parfois moi et mes potes on dit que `Maybe` c'est un type, mais bon, c'est pas ce qu'on veut dire, parce que même les idiots savent que `Maybe` est un constructeur de types, t'as vu. Quand j'applique `Maybe` à un type, comme dans `Maybe String`, alors j'ai un type concret. Tu sais, les valeurs, elles ne peuvent avoir que des types concrets ! En gros, vis vite, aime fort, et ne prête ton peigne à personne !

Tout comme l'on peut appliquer partiellement des fonctions pour obtenir de nouvelles fonctions, on peut appliquer partiellement des constructeurs de types pour obtenir de nouveaux constructeurs de types. Tout comme on appelle une fonctions avec trop peu de paramètres, on peut spécifier trop peu de paramètres de types à un constructeur de types et obtenir un constructeur de types partiellement appliqué. Si l'on voulait un type qui représente une map (de `Data.Map`) qui va des entiers vers quelque chose, on pourrait faire soit :

```
type IntMap v = Map Int v
```

Ou bien :

```
type IntMap = Map Int
```

D'une manière ou de l'autre, le constructeur de types `IntMap` prend un seul paramètre qui est le type de ce vers quoi les entiers doivent pointer.

Ah oui. Si vous comptez essayer d'implémenter ceci, vous allez probablement importer `Data.Map` qualifié. Quand vous faites un import qualifié, les constructeurs de type doivent aussi être précédés du nom du module. Donc vous écririez `type IntMap = Map.Map Int`.

Soyez certains d'avoir bien saisi la distinction entre constructeurs de types et constructeurs de valeurs. Juste parce qu'on a créé un synonyme de types `IntMap` ou `AssocList` ne signifie pas que l'on peut faire quelque chose comme `AssocList [(1, 2), (4, 5), (7, 9)]`. Tout ce que cela signifie, c'est qu'on peut parler de ce type en utilisant des noms différents. On peut faire `[(1, 2), (3, 5), (8, 9)] :: AssocList Int Int`, ce qui va faire que les nombres à l'intérieur auront pour type `Int`, mais on peut continuer à utiliser cette liste comme une liste normale qui contient des paires d'entiers. Les synonymes de types (et plus généralement les types) ne peuvent être utilisés que dans la partie types d'Haskell. On se situe dans cette partie lorsqu'on définit de nouveaux types (donc, dans une déclaration *data* ou *type*), ou lorsqu'on se situe après un `::`. Le `::` peut être dans des déclarations de types ou pour des annotations de types.

Un autre type de données cool qui prend deux types en paramètres est `Either a b`. Voici grossièrement sa définition :

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

Il a deux constructeurs de valeurs. Si le constructeur `Left` est utilisé, alors le contenu a pour type `a`, si `Right` est utilisé, le contenu a pour type `b`. Ainsi, on peut utiliser ce type pour encapsuler une valeur qui peut avoir un type ou un autre, et lorsqu'on récupère une valeur qui a pour type `Either a b`, on filtre généralement par motif sur `Left` et `Right` pour obtenir différentes choses en fonction.

```
ghci> Right 20
Right 20
ghci> Left "w00t"
Left "w00t"
ghci> :t Right 'a'
Right 'a' :: Either a Char
ghci> :t Left True
Left True :: Either Bool b
```

Jusqu'ici, nous avons vu que `Maybe a` était principalement utilisé pour représenter des résultats de calculs qui pouvaient avoir soit échoué, soit réussi. Mais parfois, `Maybe a` n'est pas assez bien, parce que `Nothing` n'indique pas d'information autre que le fait que quelque chose a échoué. C'est bien pour les fonctions qui peuvent échouer seulement d'une façon, ou lorsqu'on se fiche de savoir comment elles ont échoué. Une recherche de clé dans `Data.Map` ne peut échouer que lorsqu'une clé n'était pas présente, donc on sait ce qui s'est passé. Cependant, lorsqu'on s'intéresse à comment une fonction a échoué ou pourquoi, on utilise généralement le type `Either a b`, où `a` est un type qui peut d'une certaine façon indiquer les raisons d'un éventuel échec, et `b` est le type du résultat d'un calcul réussi. Ainsi, les erreurs utilisent le constructeur de valeurs `Left`, alors que les résultats utilisent le constructeur de valeurs `Right`.

Exemple : un lycée a des casiers dans lesquels les étudiants peuvent ranger leurs posters de Guns'n'Roses. Chaque casier a une combinaison. Quand un étudiant veut un nouveau casier, il indique au superviseur des casiers le numéro de casier qu'il voudrait, et celui-ci lui donne le code. Cependant, si quelqu'un utilise déjà ce casier, il ne peut pas lui donner le casier, et l'étudiant doit en choisir un autre. On va utiliser une map de `Data.Map` pour représenter les casiers.

Elle associera à chaque numéro de casier une paire indiquant si le casier est utilisé ou non, et le code du casier.

```
import qualified Data.Map as Map

data LockerState = Taken | Free deriving (Show, Eq)

type Code = String

type LockerMap = Map.Map Int (LockerState, Code)
```

Simple. On introduit un nouveau type de données pour représenter un casier occupé ou libre, et on crée un synonyme de types pour le code du casier. On crée aussi un synonyme de types pour les maps qui prennent un entier et renvoient une paire d'état et de code de casier. À présent, on va créer une fonction qui cherche le code dans une map de casiers. On va utiliser un type `Either String Code` pour représenter le résultat, parce que la recherche peut échouer de deux façons - le casier peut être déjà pris, auquel cas on ne peut pas dévoiler le code, ou bien le numéro de casier peut tout simplement ne pas exister. Si la recherche échoue, on va utiliser une `String` pour décrire ce qui s'est passé.

```
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map =
    case Map.lookup lockerNumber map of
        Nothing -> Left $ "Locker number " ++ show lockerNumber ++ " doesn't exist!"
        Just (state, code) -> if state /= Taken
                              then Right code
                              else Left $ "Locker " ++ show lockerNumber ++ " is already taken!"
```

On fait une recherche normale dans la map. Si l'on obtient `Nothing`, on retourne une valeur de type `Left String`, qui dit que le casier n'existe pas. Si on trouve le casier, on effectue une vérification supplémentaire pour voir s'il est utilisé. Si c'est le cas, on retourne `Left` en indiquant qu'il est déjà pris. Autrement, on retourne une valeur de type `Right Code`, dans laquelle on donne le code dudit casier. En fait, c'est un `Right String`, mais on a introduit un synonyme de types pour introduire un peu de documentation dans la déclaration de types. Voici une map à titre d'exemple :

```
lockers :: LockerMap
lockers = Map.fromList
    [(100,(Taken,"ZD39I"))
    ,(101,(Free,"JAH3I"))
    ,(103,(Free,"IQSA9"))
    ,(105,(Free,"QOTSA"))
    ,(109,(Taken,"893JJ"))
    ,(110,(Taken,"99292"))
    ]
```

Maintenant, cherchons le code de quelques casiers.

```
ghci> lockerLookup 101 lockers
Right "JAH3I"
ghci> lockerLookup 100 lockers
Left "Locker 100 is already taken!"
ghci> lockerLookup 102 lockers
Left "Locker number 102 doesn't exist!"
ghci> lockerLookup 110 lockers
Left "Locker 110 is already taken!"
ghci> lockerLookup 105 lockers
Right "QOTSA"
```

On aurait pu utiliser un `Maybe a` pour représenter le résultat, mais alors on ne saurait pas pour quelle raison on n'a pas pu obtenir le code. Ici, l'information en cas d'échec est dans le type de retour.

Structures de données récursives

Comme on l'a vu, un constructeur de type de données algébrique peut avoir plusieurs (ou aucun) champs et chaque champ doit avoir un type concret. Avec cela en tête, on peut créer des types dont les constructeurs ont des champs qui sont de ce même type ! Ainsi, on peut créer des types de données récursifs, où une valeur d'un type peut contenir des valeurs de ce même type, qui à leur tour peuvent contenir encore plus de valeurs de ce type, et ainsi de suite.

Pensez à cette liste : `[5]`. C'est juste un sucre syntaxique pour `5:[]`. À gauche de `:`, il y a une valeur, et à droite, il y a une liste. Dans ce cas, c'est une liste vide. Maintenant, qu'en est-il de `[4, 5]` ? Eh bien, ça se désucre en `4:(5:[])`. En considérant le premier `:`, on voit qu'il prend aussi un élément à gauche et une liste (ici `5:[]`) à droite. Il en va de même pour `3:(4:(5:(6:[])))`, qui peut aussi être écrit `3:4:5:6:[]` (parce que `:` est associatif à droite) ou `[3, 4, 5, 6]`.

On peut dire qu'une liste peut être la liste vide ou composée d'un élément adjoint via `:` à une autre liste (qui peut être vide ou non).

Utilisons un type de données algébrique pour implémenter nos propres listes dans ce cas !

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

On peut lire ça comme la définition des listes donnée dans un paragraphe ci-dessus. Une liste est soit vide, soit une combinaison d'une tête qui a une valeur et d'une autre liste. Si cela vous laisse perplexe, peut-être que vous serez plus à l'aise avec une syntaxe d'enregistrements.

```
data List a = Empty | Cons { listHead :: a, listTail :: List a} deriving (Show, Read, Eq, Ord)
```

Vous serez peut-être aussi surpris par le nom du constructeur `Cons` ici. *cons* est un autre nom de `:`. Vous voyez, pour les listes, `:` est en réalité un constructeur qui prend une valeur et une autre liste, pour retourner une liste. On peut d'ores et déjà utiliser notre nouveau type de listes ! Il a deux champs. Le premier de type `a` et le second de type `List a`.

```
ghci> Empty
Empty
ghci> 5 `Cons` Empty
Cons 5 Empty
ghci> 4 `Cons` (5 `Cons` Empty)
Cons 4 (Cons 5 Empty)
ghci> 3 `Cons` (4 `Cons` (5 `Cons` Empty))
Cons 3 (Cons 4 (Cons 5 Empty))
```

On a appelé notre constructeur `Cons` de façon infixe pour souligner sa similarité avec `:`. `Empty` est similaire à `[]` et `4 `Cons` (5 `Cons` Empty)` est similaire à `4:(5:[])`.

On peut définir des fonctions comme automatiquement infixes en ne les nommant qu'avec des caractères spéciaux. On peut aussi faire de même avec les constructeurs, puisque ce sont des fonctions qui retournent un type de données. Regardez ça !

```
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)
```

Tout d'abord, on remarque une nouvelle construction syntaxique, la déclaration de fixité. Lorsqu'on définit des fonctions comme opérateurs, on peut leur donner une fixité (ce n'est pas nécessaire). Une fixité indique avec quelle force un opérateur lie, et s'il est associatif à droite ou à gauche. Par exemple, la fixité de `*` est `infixl 7 *`, et la fixité de `+` est `infixl 6`. Cela signifie qu'ils sont tous deux associatifs à gauche (`4 * 3 * 2` est équivalent à `(4 * 3) * 2)`, mais `*` lie plus fortement que `+`, car il a une plus grande fixité, et ainsi `5 * 4 + 3` est équivalent à `(5 * 4) + 3`.

À part ce détail, on a juste écrit `a :-: (List a)` à la place de `Cons a (List a)`. Maintenant, on peut écrire des listes qui ont notre type de listes de la sorte :

```
ghci> 3 :-: 4 :-: 5 :-: Empty
(:-:) 3 ((:-:) 4 ((:-:) 5 Empty))
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> 100 :-: a
(:-:) 100 ((:-:) 3 ((:-:) 4 ((:-:) 5 Empty)))
```

Lorsqu'on dérive `Show` pour notre type, Haskell va toujours afficher le constructeur comme une fonction préfixe, d'où le parenthésage autour de l'opérateur (rappelez-vous, `4 + 3` est juste `(+) 4 3`).

Créons une fonction qui somme deux de nos listes ensemble. `++` est défini ainsi pour des listes normales :

```
infixr 5  ++
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

On va juste voler cette définition pour nos listes. On nommera la fonction `.++`.

```
infixr 5  .++
(.++) :: List a -> List a -> List a
Empty .++ ys = ys
(x :-: xs) .++ ys = x :-: (xs .++ ys)
```

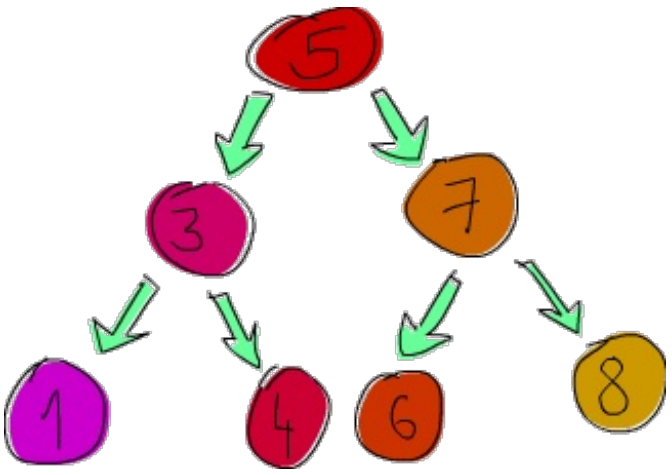
Voyons si ça a marché…

```
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
```

```
ghci> let b = 6 :-: 7 :-: Empty
ghci> a .++ b
(:-:) 3 ((:-:) 4 ((:-:) 5 ((:-:) 6 ((:-:) 7 Empty))))
```

Bien. Très bien. Si l'on voulait, on pourrait implémenter toutes les fonctions qui opèrent sur des listes pour notre propre type liste.

Notez comme on a filtré sur le motif `(x :-: xs)`. Cela fonctionne car le filtrage par motif n'est en fait basé que sur la correspondance des constructeurs. On peut filtrer sur `:-:` parce que c'est un constructeur de notre type de liste, tout comme on pouvait filtrer sur `:` pour des listes normales car c'était un de leurs constructeurs. Pareil pour `[]`. Puisque le filtrage par motif marche (uniquement) sur les constructeurs, on peut filtrer sur des choses comme des constructeurs normaux préfixes, ou bien comme `8` ou `'a'`, qui sont simplement des constructeurs de types numériques ou de caractères respectivement.



À présent, implémentons un **arbre binaire de recherche**. Si vous ne connaissez pas les arbres binaires de recherche, voici en quoi ils consistent : chaque élément pointe vers deux éléments, son fils gauche et son fils droit. Le fils gauche a une valeur inférieure à celle de l'élément, le fils droit a une valeur supérieure. Chacun des fils peut à son tour pointer vers zéro, un ou deux éléments. Chaque élément a ainsi au plus deux sous-arbres. Et la propriété intéressante de ces arbres est que tous les nœuds du sous-arbre gauche d'un nœud donné, mettons 5, ont une valeur inférieure à 5. Quant aux nœuds du sous-arbre droit, ils sont tous supérieurs à 5. Ainsi, si l'on cherche 8 dans un arbre dont la racine vaut 5, on va chercher seulement dans le sous-arbre droit, puisque 8 est plus grand que 5. Si le nœud suivant est 7, on cherche encore à droite car 8 est plus grand que 7. Et voilà ! On a trouvé notre élément en seulement 3 coups ! Si l'on cherchait dans une liste, ou dans un arbre mal construit, cela pourrait nous prendre jusqu'à 8 coups pour savoir si 8 est là ou pas.

Les ensembles de `Data.Set` et les maps de `Data.Map` sont implémentés à l'aide d'arbres binaires de recherche équilibrés, qui sont toujours bien équilibrés. Mais pour l'instant, nous allons simplement implémenter des arbres de recherches binaires standards.

Voilà ce qu'on va dire : un arbre est soit vide, soit un élément contenant une valeur et deux arbres. Ça sent le type de données algébrique à plein nez !

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

Ok, bien, c'est très bien. Plutôt que de construire des arbres à la main, on va créer une fonction qui prend un arbre, un élément, et insère cet élément. Cela se fait en comparant la valeur de l'élément que l'on souhaite insérer avec le nœud racine, s'il est plus petit on part à gauche, s'il est plus grand on part à droite. On fait de même avec tous les nœuds suivants jusqu'à ce qu'on arrive à un arbre vide. C'est ici qu'on doit insérer notre nœud qui va simplement remplacer ce vide.

Dans des langages comme C, on fait ceci en modifiant des pointeurs et des valeurs dans l'arbre. En Haskell, on ne peut pas vraiment modifier l'arbre, donc on recrée un nouveau sous-arbre chaque fois qu'on décide d'aller à gauche ou à droite et au final, la fonction d'insertion construit un tout nouvel arbre, puisqu'Haskell n'a pas de concept de pointeurs mais seulement de valeurs. Ainsi, le type de la fonction d'insertion sera de la forme `a -> Tree a -> Tree a`. Elle prend un élément et un arbre et retourne un nouvel arbre qui contient cet élément. Ça peut paraître inefficace, mais la paresse se charge de ce problème.

Voici donc deux fonctions. L'une est une fonction auxiliaire pour créer un arbre singleton (qui ne contient qu'un nœud) et l'autre est une fonction d'insertion.

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a  = Node a (treeInsert x left) right
    | x > a  = Node a left (treeInsert x right)
```

La fonction `singleton` est juste un raccourci pour créer un nœud contenant une valeur et deux sous-arbres vides. Dans la fonction d'insertion, on a d'abord notre cas de base sous forme d'un motif. Si l'on atteint un arbre vide et qu'on veut y insérer notre valeur, cela veut dire qu'il faut renvoyer l'arbre singleton qui contient cette valeur. Si l'on insère dans un arbre non vide, il faut vérifier quelques choses. Si l'élément qu'on veut insérer est égal à la racine, alors on retourne le même arbre. S'il est plus petit, on retourne un arbre qui a la même racine, le même sous-arbre droit, mais qui a pour sous-arbre gauche le même qu'avant auquel on a ajouté l'élément à ajouter. Le raisonnement est symétrique si l'élément à ajouter est plus grand que l'élément à la racine.

Ensuite, on va créer une fonction qui vérifie si un élément est dans l'arbre. Définissons d'abord le cas de base. Si on cherche un élément dans l'arbre vide, il n'est certainement pas là. Ok. Voyez comme le cas de base est similaire au cas de base lorsqu'on cherche un élément dans une liste. Si on cherche un élément dans une liste vide, il n'est sûrement pas là. Enfin bon, si l'arbre n'est pas vide, quelques vérifications. Si l'élément à la racine est celui qu'on cherche, trouvé ! Sinon, eh bien ? Puisqu'on sait que les éléments à gauche sont plus petits que lui, si on cherche un élément plus petit, on va le chercher à gauche. Symétriquement, on cherchera à droite un élément plus grand que celui à la racine.

```
treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
```

```
treeElem x (Node a left right)
    | x == a = True
    | x < a  = treeElem x left
    | x > a  = treeElem x right
```

Tout ce qu'on a eu à faire, c'est réécrire le paragraphe précédent en code. Amusons-nous avec nos arbres ! Plutôt que d'en créer un à la main (bien que ce soit possible), créons-en un à l'aide d'un pli sur une liste. Souvenez-vous, à peu près tout ce qui traverse une liste et retourne une valeur peut être implémenté comme un pli ! On va commencer avec un arbre vide, puis approcher une liste par la droite en insérant les éléments un à un dans l'arbre accumulateur.

```
ghci> let nums = [8,6,4,1,7,3,5]
ghci> let numsTree = foldr treeInsert EmptyTree nums
ghci> numsTree
Node 5 (Node 3 (Node 1 EmptyTree EmptyTree) (Node 4 EmptyTree EmptyTree)) (Node 7 (Node 6 EmptyTree EmptyTree) (Node 8 EmptyTre
```

Dans ce **`foldr`**, **`treeInsert`** était la fonction de pliage (elle prend un arbre, un élément d'une liste et produit un nouvel arbre) et **`EmptyTree`** était l'accumulateur initial. **`nums`**, évidemment, était la liste qu'on pliait.

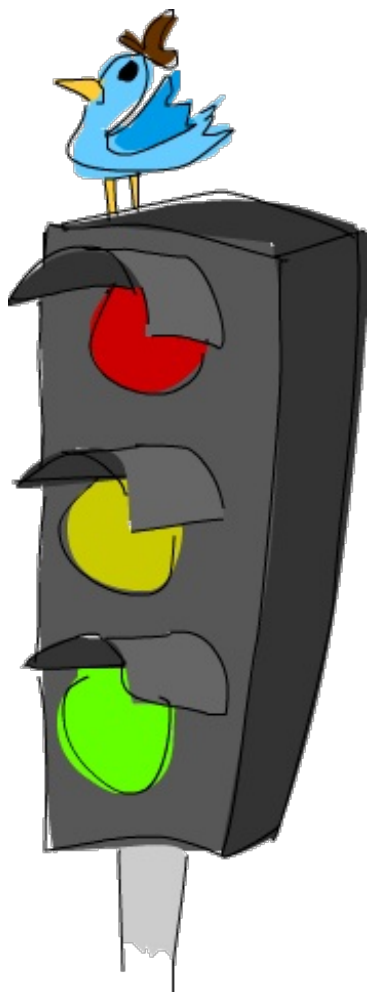
Lorsqu'on affiche notre arbre à la console, il n'est pas très lisible, mais en essayant, on peut voir sa structure. On peut voir que le nœud racine est 5 et qu'il a deux sous-arbres, un qui a pour nœud racine 3, l'autre qui a pour nœud racine 7, etc.

```
ghci> 8 `treeElem` numsTree
True
ghci> 100 `treeElem` numsTree
False
ghci> 1 `treeElem` numsTree
True
ghci> 10 `treeElem` numsTree
False
```

Tester l'appartenance fonctionne aussi correctement. Cool.

Comme vous le voyez, les structures de données algébriques sont des concepts cool et puissants en Haskell. On peut les utiliser pour tout faire, des booléens et énumérations des jours de la semaine jusqu'aux arbres binaires de recherche, et encore plus !

Classes de types 102



Jusqu'ici, on a découvert des classes de types standard d'Haskell et on a vu quels types les habitaient. On a aussi appris à créer automatiquement des instances de ces classes en demandant à Haskell de les dériver pour nous. Dans cette section, on va créer nos propres classes de types, et nos propres instances, le tout à la main.

Bref récapitulatif sur les classes de types : elles sont comme des interfaces. Une classe de types définit un comportement (tester l'égalité, comparer l'ordre, énumérer), puis les types qui peuvent se comporter de la sorte sont fait instances de ces classes. Le comportement des classes de types est défini en définissant des fonctions ou juste des déclarations de types que les instances doivent implémenter. Ainsi, lorsqu'on dit qu'un type est une instance d'une classe de types, on veut dire qu'on peut utiliser les fonctions que cette classe définit sur des éléments de ce type.

Les classes de types n'ont presque rien à voir avec les types dans des langages comme Java ou Python. Cela désoriente beaucoup de gens, aussi je veux que vous oubliiez tout ce que vous savez des classes dans les langages impératifs à partir de maintenant.

Par exemple, la classe **`Eq`** est pour les choses qui peuvent être testées égales. Elle définit les fonctions **`==`** et **`/=`**. Si on a un type (mettons, **`Car`**) et que comparer deux voitures avec **`==`** a un sens, alors il est sensé de rendre **`Car`** instance d'**`Eq`**.

Voici comment **`Eq`** est définie dans le prélude :

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Wow, wow, wow ! Quels nouveaux syntaxe et mot-clés étranges ! Pas d'inquiétude, tout cela va s'éclaircir bientôt. Tout d'abord, quand on écrit **`class Eq a where`**, cela signifie qu'on définit une nouvelle classe **`Eq`**. Le **`a`** est une variable de types et indique que **`a`** jouera le rôle du type qu'on souhaitera bientôt rendre instance d'**`Eq`**. Cette variable n'a pas à s'appeler **`a`**, ou même à être une lettre, elle doit juste être un mot en minuscules. Ensuite, on définit plusieurs fonctions. Il n'est pas obligatoire d'implémenter le corps de ces fonctions, on doit juste spécifier les déclarations de type des fonctions.

Certaines personnes comprendraient peut-être mieux si l'on avait écrit `class Eq equatable where` et ensuite spécifié les déclarations de type comme `(==) :: equatable -> equatable -> Bool`.

Toutefois, on *a implémenté* le corps des fonctions que `Eq` définit, mais on les a définies en termes de récursion mutuelle. On a dit que deux instances d'`Eq` sont égales si elles ne sont pas différentes, et différentes si elles ne sont pas égales. Ce n'était pas nécessaire, mais on l'a fait, et on va bientôt voir en quoi cela nous aide.

Si nous avons, disons, `class Eq a where` et qu'on définit une déclaration de type dans cette classe comme `(==) :: a -> a -> Bool`, alors si on examine le type de cette fonction, celui-ci sera `(Eq a) => a -> a -> Bool`.

Maintenant qu'on a une classe, que faire avec ? Eh bien, pas grand chose, vraiment. Mais une fois qu'on a des instances, on commence à bénéficier de fonctionnalités sympathiques. Regardez donc ce type :

```
data TrafficLight = Red | Yellow | Green
```

Il définit les états d'un feu de signalisation. Voyez comme on n'a pas dérivé d'instances de classes. C'est parce qu'on va les écrire à la main, bien qu'on aurait pu dériver celles-ci pour des classes comme `Eq` et `Show`. Voici comment on fait de notre type une instance d'`Eq`.

```
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

On l'a fait en utilisant le mot-clé *instance*. *class* définit de nouvelles classes de types, et *instance* définit de nouvelles instances d'une classe de types. Quand nous définissions `Eq`, on a écrit `class Eq a where` et on a dit que `a` jouait le rôle du type qu'on voudra rendre instance de la classe plus tard. On le voit clairement ici, puisque quand on crée une instance, on écrit `instance Eq TrafficLight where`. On a remplacé le `a` par le type réel de cette instance particulière.

Puisque `==` était défini en termes de `/=` et vice versa dans la déclaration de *classe*, il suffit d'en définir un dans l'instance pour obtenir l'autre automatiquement. On dit que c'est une définition complète minimale d'une classe de types - un des plus petits ensembles de fonctions à implémenter pour que le type puisse se comporter comme une instance de la classe. Pour remplir la définition complète minimale d'`Eq`, il faut redéfinir soit `==`, soit `/=`. Cependant, si `Eq` était définie seulement ainsi :

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

alors nous aurions dû implémenter ces deux fonctions lors de la création d'une instance, car Haskell ne saurait pas comment elles sont reliées. La définition complète minimale serait alors : `==` et `/=`.

Vous pouvez voir qu'on a implémenté `==` par un simple filtrage par motif. Puisqu'il y a beaucoup de cas où deux lumières sont différentes, on a spécifié les cas d'égalité, et utilisé un motif attrape-tout pour dire que dans tous les autres cas, les lumières sont différentes.

Créons une instance de `Show` à la main également. Pour satisfaire une définition complète minimale de `Show`, il suffit d'implémenter `show`, qui prend une valeur et retourne une chaîne de caractères.

```
instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

Encore une fois, nous avons utilisé le filtrage par motif pour arriver à nos fins. Voyons cela en action :

```
ghci> Red == Red
True
ghci> Red == Yellow
False
ghci> Red `elem` [Red, Yellow, Green]
True
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

Joli. On aurait pu simplement dériver `Eq` et obtenir la même chose (on ne l'a pas fait, dans un but éducatif). Cependant, dériver `Show` aurait simplement traduit les constructeurs en chaînes de caractères. Si on veut afficher `"Reg light"`, il faut faire la déclaration d'instance à la main.

Vous pouvez également créer des classes de types qui sont des sous-classes d'autres classes de types. La déclaration de classe de `Num` est un peu longue, mais elle débute ainsi :

```
class (Eq a) => Num a where
   ...
```

Comme on l'a mentionné précédemment, il y a beaucoup d'endroits où l'on peut glisser des contraintes de classe. Ici c'est comme écrire `class Num a where`, mais on déclare que `a` doit être une instance d'`Eq` au préalable. Ainsi, un type doit être une instance d'`eq` avant de pouvoir prétendre être une instance de `Num`. Il est logique qu'avant qu'un type soit considéré comme numérique, on puisse attendre de lui qu'il soit testable pour l'égalité. C'est tout pour le sous-typage, c'est seulement une contrainte de classes sur une déclaration de *classe* ! À partir de là, quand on définit le corps des fonctions, que ce soit dans la déclaration de *classe* ou bien dans une déclaration d'*instance*, on peut toujours présumer que `a` est membre d'`Eq` et ainsi utiliser `==` sur des valeurs de ce type.

Mais comment est-ce que `Maybe`, ou le type des listes, sont rendus instances de classes de types ? Ce qui rend `Maybe` différent de, par exemple, `TrafficLight`, c'est que `Maybe` tout seul n'est pas un type concret, mais un constructeur de types qui prend un type en paramètre (comme `Char` ou un autre) pour produire un type concret (comme `Maybe Char`). Regardons à nouveau la classe de types `Eq` :

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

De ces déclarations de types, on voit que `a` est utilisé comme un type concret car tous les types entre les flèches d'une fonction doivent être concrets (souvenez-vous, on ne peut avoir de fonction de type `a -> Maybe`, mais on peut avoir des fonctions `a -> Maybe a` ou `Maybe Int -> Maybe String`). C'est pour cela qu'on ne peut pas faire :

```
instance Eq Maybe where
    ...
```

Parce que, comme on l'a vu, `a` doit être un type concret, et `Maybe` ne l'est pas. C'est un constructeur de types qui prend un paramètre pour produire un type concret. Il serait également très fastidieux d'écrire des instances `instance Eq (Maybe Int) where`, `instance Eq (Maybe Char) where`, etc. pour chaque type qu'on utilise. Ainsi, on peut écrire :

```
instance Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

C'est-à-dire qu'on déclare tous les types de la forme `Maybe something` comme instances d'`Eq`. On aurait à vrai dire pu écrire `(Maybe something)`, mais on opte généralement pour des identifiants en une lettre pour rester proche du style Haskell. Le `(Maybe m)` ici joue le rôle du `a` de `class Eq a where`. Alors que `Maybe` n'était pas un type concret, `Maybe m` l'est. En spécifiant un paramètre de type (`m`, qui est en minuscule), on dit qu'on souhaite parler de tous les types de la forme `Maybe m`, où `m` est n'importe quel type, afin d'en faire une instance d'`Eq`.

Il y a un problème à cela tout de même. Le voyez-vous ? On utilise `==` sur le contenu de `Maybe`, mais on n'a pas de garantie que ce que contient `Maybe` est membre d'`Eq` ! C'est pourquoi nous devons modifier la déclaration d'*instance* ainsi :

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

On a dû ajouter une contrainte de classe ! Avec cette déclaration d'*instance*, on dit ceci : nous voulons que tous les types de la forme `Maybe m` soient membres de la classe `Eq`, à condition que le type `m` (celui dans le `Maybe`) soit lui-même membre d'`Eq`. C'est ce qu'Haskell dériverait d'ailleurs.

La plupart du temps, les contraintes de classe dans les déclarations de *classes* sont utilisées pour faire d'une classe de types une sous-classe d'une autre classe de types, alors que les contraintes de classe dans les déclarations d'*instance* sont utilisées pour exprimer des pré-requis sur le contenu de certains types. Par exemple, ici nous avons requis que le contenu de `Maybe` soit membre de la classe `Eq`.

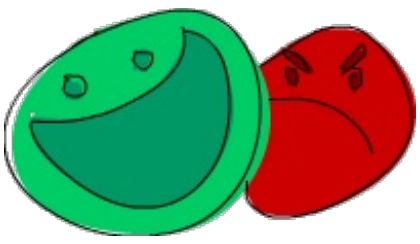
Quand vous faites des instances, si vous voyez qu'un type est utilisé comme un type concret dans les déclarations de type de la classe (comme le `a` dans

`a -> a -> Bool`), alors il faut fournir à l'instance un type concret en fournissant si besoin est des paramètres de type et en parenthésant le tout, de manière à obtenir un type concret.

Prenez en compte le fait que le type dont vous essayez de faire une instance remplacera le paramètre dans la déclaration de *classe*. Le `a` de `class Eq a where` sera remplacé par un type réel lorsque vous écrirez une instance, donc essayez de mettre mentalement le type dans les déclarations de types des fonctions. `(==) :: Maybe -> Maybe -> Bool` ne veut ainsi par dire grand chose, alors que `(==) :: (Eq m) => Maybe m -> Maybe m -> Bool` est sensé. C'est juste pour mieux y voir dans votre tête, en réalité, `==` conservera toujours son type `(==) :: (Eq a) => a -> a -> Bool`, peu importe le nombre d'instances que l'on crée.

Oh, encore un truc, regardez ça ! Si vous voulez connaître les instances d'une classe de types, faites juste `:info YourTypeClass` dans GHCi. En tapant `:info Num`, vous verrez toutes les fonctions que définit cette classe de types, suivies d'une liste de tous les types qui habitent cette classe. `:info` marche aussi sur les types et les constructeurs de types. `:info Maybe` vous montre toutes les classes dont `Maybe` est une instance. `:info` peut aussi montrer la déclaration de type d'une fonction. Je trouve ça plutôt cool.

Une classe de types oui-non



En JavaScript et dans d'autres langages à typage faible, on peut mettre quasiment ce que l'on veut dans une expression *if*. Par exemple, tout ceci est valide : `if (0) alert ("YEAH") else alert("NO!")`, `if ("") alert ("YEAH") else alert("NO!")`, `if (false) alert ("YEAH") else alert("NO!")`, etc. et tout cela lancera une alerte `NO!`. Si vous faites `if ("WHAT") alert ("YEAH") else alert("NO!")`, une alerte `"YEAH!"` sera lancée parce que JavaScript considère toutes les chaînes de caractères non vides comme des sortes de valeurs vraies.

Bien qu'utiliser `Bool` pour une sémantique booléenne marche mieux en Haskell, implémentons un fonctionnement à la JavaScript pour le fun ! Commençons par une déclaration de *classe*.

```
class YesNo a where
    yesno :: a -> Bool
```

Plutôt simple. La classe `YesNo` définit une seule fonction. Cette fonction prend une valeur d'un type qui peut représenter le concept de vérité, et nous indique si cette valeur est vraie ou fausse. Remarquez que la façon dont on utilise `a` dans cette déclaration implique que `a` doit être un type concret.

Ensuite, définissons des instances. Pour les nombres, on va dire que tout nombre qui n'est pas 0 est vrai, et que 0 est faux (comme en JavaScript).

```
instance YesNo Int where
    yesno 0 = False
    yesno _ = True
```

Les listes vides (et par extension les chaînes de caractères vides) sont des valeurs fausses, alors que les listes ou chaînes non vides sont des valeurs vraies.

```
instance YesNo [a] where
    yesno [] = False
    yesno _ = True
```

Remarquez qu'on a ajouté un paramètre de type `a` ici pour faire de la liste un type concret, bien qu'on n'ait pas ajouté de contrainte sur ce que la liste contient. Quoi d'autre, hmmm… Je sais, `Bool` aussi contient la notion de vérité, et c'est plutôt évident de savoir lequel est quoi.

```
instance YesNo Bool where
    yesno = id
```

Hein ? Qu'est-ce qu'`id` ? C'est juste une fonction de la bibliothèque standard qui prend un paramètre et retourne la même chose, ce qui correspond à ce qu'on écrirait ici de toute façon.

Faisons de `Maybe a` une instance.

```
instance YesNo (Maybe a) where
    yesno (Just _) = True
    yesno Nothing = False
```

Pas besoin de contrainte de classe puisqu'on ne fait aucune supposition sur ce que contient le `Maybe`. On a juste considéré que toute valeur `Just` est vraie, alors que `Nothing` est faux. Il a encore fallu écrire `(Maybe a)` plutôt que `Maybe` tout court, parce qu'en y réfléchissant, une fonction `Maybe -> Bool` ne peut

pas exister (car `Maybe` n'est pas un type concret), alors que `Maybe a -> Bool` est bien et propre. Cependant, c'est plutôt cool puisqu'à présent, tout type de la forme `Maybe something` est membre de `YesNo`, peu importe ce que `something` est.

Précédemment, on a défini un type `Tree a`, qui représentait un arbre binaire de recherche. On peut dire qu'un arbre vide est faux, alors qu'un arbre non vide est vrai.

```
instance YesNo (Tree a) where
    yesno EmptyTree = False
    yesno _ = True
```

Est-ce qu'un feu tricolore peut être une valeur vraie ou fausse ? Bien sûr. Si c'est rouge, on s'arrête. Si c'est vert, on y va. Si c'est jaune ? Eh, personnellement je foncerai pour l'adrénaline.

```
instance YesNo TrafficLight where
    yesno Red = False
    yesno _ = True
```

Cool, on a plein d'instances, jouons avec !

```
ghci> yesno $ length []
False
ghci> yesno "haha"
True
ghci> yesno ""
False
ghci> yesno $ Just 0
True
ghci> yesno True
True
ghci> yesno EmptyTree
False
ghci> yesno []
False
ghci> yesno [0,0,0]
True
ghci> :t yesno
yesno :: (YesNo a) => a -> Bool
```

Bien, ça marche ! Faisons une fonction qui copie le comportement de *if*, mais avec des valeurs `YesNo`.

```
yesnoIf :: (YesNo y) => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult = if yesno yesnoVal then yesResult else noResult
```

Plutôt direct. Elle prend une valeur `YesNo` et deux choses. Si la valeur est plutôt vraie, elle retourne la première chose, sinon elle retourne la deuxième.

```
ghci> yesnoIf [] "YEAH!" "NO!"
"NO!"
ghci> yesnoIf [2,3,4] "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf True "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf (Just 500) "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf Nothing "YEAH!" "NO!"
"NO!"
```

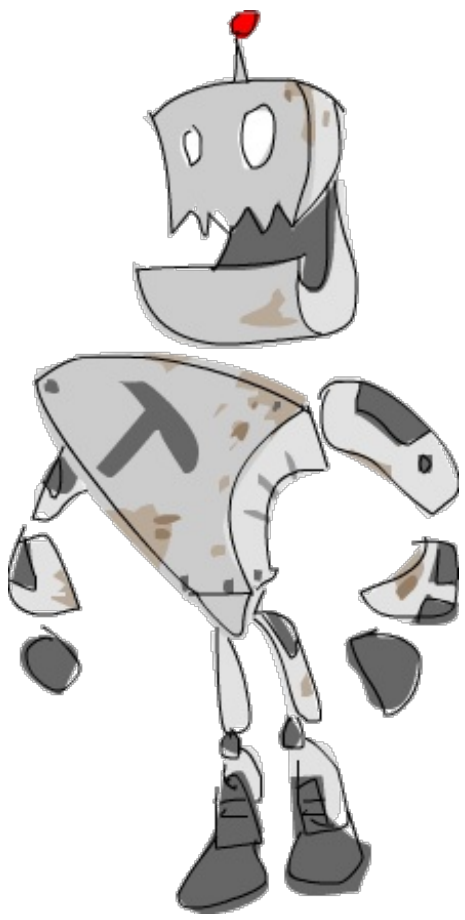
La classe de types Functor

Jusqu'ici, nous avons rencontré pas mal de classes de types de la bibliothèque standard. On a joué avec `Ord`, pour les choses qu'on peut ranger en ordre. On s'est familiarisé avec `Eq`, pour les choses dont on peut tester l'égalité. On a vu `Show`, qui présente une interface pour les choses qui peuvent être affichées sous une forme de chaîne de caractères. Notre bon ami `Read` est là lorsqu'on veut convertir des chaînes de caractères en des valeurs d'un certain type. Et maintenant, on va regarder la classe de types `Functor`, qui est simplement pour les choses sur lesquelles on peut mapper. Vous pensez probablement aux listes à l'instant, puisque mapper sur des listes est un idiome prédominant en Haskell. Et vous avez raison, le type des listes est bien membre de la classe `Functor`.

Quel meilleur moyen de connaître `Functor` que de regarder son implémentation ? Jetons un coup d'œil.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Parfait. On voit que la classe définit une fonction, `fmap`, et ne fournit pas d'implémentation par défaut. Le type de `fmap` est intéressant. Jusqu'ici, dans les définitions de classes de types, la variable de type qui jouait le rôle du type dans la classe de type était un type concret, comme le `a` dans `(==) :: (Eq a) => a -> a -> Bool`. Mais maintenant, le `f` n'est pas un type concret (un type qui peut contenir des valeurs, comme `Int`, `Bool` ou `Maybe String`), mais un constructeur de types qui prend un paramètre de type. Petit rafraîchissement par l'exemple : `Maybe Int` est un type concret, mais `Maybe` est un constructeur qui prend un type en paramètre. Bon, on voit que `fmap` prend une fonction d'un type vers un autre type et un foncteur appliqué au premier type, et retourne un foncteur appliqué au second type.



Si cela semble un peu confus, ne vous inquiétez pas. Tout sera bientôt révélé avec des exemples. Hmm, cette déclaration de type de `fmap` me rappelle quelque chose. Si vous ne connaissez pas la signature de `map`, c'est `map :: (a -> b) -> [a] -> [b]`.

Ah, intéressant ! Elle prend une fonction d'un type vers un autre type, une liste du premier type, et retourne une liste du second type. Mes amis, je crois que nous avons là un foncteur ! En fait, `map` est juste le `fmap` des listes. Voici l'instance de `Functor` pour les listes.

```
instance Functor [] where
    fmap = map
```

C'est tout ! Remarquez qu'on n'a pas écrit `instance Functor [a] where`, parce qu'en regardant `fmap :: (a -> b) -> f a -> f b`, on voit que `f` doit être un constructeur de types qui prend un type. `[a]` est déjà un type concret (d'une liste contenant n'importe quel type), alors que `[]` est un constructeur de types qui prend un type et produit des types comme `[Int]`, `[String]` ou même `[[String]]`.

Puisque pour les listes, `fmap` est juste `map`, on obtient les mêmes résultats en utilisant l'un ou l'autre sur des listes.

```
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

Que se passe-t-il lorsqu'on `map` ou `fmap` sur une liste vide ? Eh bien, bien sûr, on obtient une liste vide. On transforme simplement une liste vide de type `[a]` en une liste vide de type `[b]`.

Tous les types qui peuvent se comporter comme des boîtes peuvent être des foncteurs. Vous pouvez imaginer une liste comme une boîte ayant une infinité de petits compartiments, et ils peuvent être soit tous vides, soit partiellement remplis. Donc, quoi d'autre qu'une liste peut être assimilé à une boîte ? Pour commencer, le type `Maybe a`. C'est comme une boîte qui peut contenir soit rien, auquel cas elle a la valeur `Nothing`, soit un élément, comme `"HAHA"`, auquel cas elle a la valeur `Just "HAHA"`. Voici comment `Maybe` est fait foncteur.

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

Encore une fois, remarquez qu'on a écrit `instance Functor Maybe where` et pas `instance Functor (Maybe m) where`, contrairement à ce qu'on avait fait avec `Maybe` dans `YesNo`. `Functor` veut un constructeur de types qui prend un paramètre, pas un type concret. Si vous remplacez mentalement les `f` par des `Maybe`, `fmap` agit comme `(a -> b) -> Maybe a -> Maybe b` pour ce type, ce qui semble correct. Par contre, si vous remplacez mentalement `f` par `(Maybe m)`, alors elle devrait agir comme `(a -> b) -> Maybe m a -> Maybe m b`, ce qui n'a pas de sens puisque `Maybe` ne prend qu'un seul paramètre de type.

Bon, l'implémentation de `fmap` est plutôt simple. Si c'est une valeur vide `Nothing`, alors on retourne `Nothing`. Si on mappe sur une boîte vide, on obtient une boîte vide. C'est sensé. Tout comme mapper sur une liste vide retourne une liste vide. Si ce n'est pas une valeur vide, mais plutôt une valeur emballée dans un `Just`, alors on applique la fonction sur le contenu du `Just`.

```
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") (Just "Something serious.")
Just "Something serious. HEY GUYS IM INSIDE THE JUST"
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") Nothing
Nothing
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```

Une autre chose sur laquelle on peut mapper, et donc en faire une instance de Functor, c'est notre type `Tree a`. On peut l'imaginer comme une boîte qui peut contenir zéro ou plusieurs valeurs, et le constructeur de type `Tree` prend exactement un paramètre de type. Si vous regardez `fmap` comme une fonction sur les

`Tree`, sa signature serait `(a -> b) -> Tree a -> Tree b`. On va utiliser de la récursivité pour celui-là. Mapper sur un arbre vide produira un arbre vide. Mapper sur un arbre non vide donnera un arbre où notre fonction sera appliquée sur le nœud racine, et les deux sous-arbres correspondront aux sous-arbres du nœud de départ sur lesquels on aura mappé la même fonction.

```
instance Functor Tree where
    fmap f EmptyTree = EmptyTree
    fmap f (Node x leftsub rightsub) = Node (f x) (fmap f leftsub) (fmap f rightsub)
```

```
ghci> fmap (*2) EmptyTree
EmptyTree
ghci> fmap (*4) (foldr treeInsert EmptyTree [5,7,3,2,1,7])
Node 28 (Node 4 EmptyTree (Node 8 EmptyTree (Node 12 EmptyTree (Node 20 EmptyTree EmptyTree)))) EmptyTree
```

Joli ! Et pourquoi pas `Either a b` à présent ? Peut-on en faire un foncteur ? La classe de types `Functor` veut des constructeurs de types à un paramètre de type, mais `Either` en prend deux. Hmmm ! Je sais, on va partiellement appliquer `Either` en lui donnant un paramètre de type, de façon à ce qu'il n'ait plus qu'un paramètre libre. Voici comment `Either a` est fait foncteur dans la bibliothèque standard :

```
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x
```

Eh bien, qu'avons-nous fait là ? Vous voyez qu'on a fait d'`Either a` une instance, au lieu d'`Either`. C'est parce qu'`Either a` est un constructeur de types qui attend un paramètre, alors qu'`Either` en attend deux. Si `fmap` était spécifiquement pour `Either a`, elle aurait pour signature `(b -> c) -> Either a b -> Either a c`, parce que c'est identique à `(b -> c) -> (Either a) b -> (Either a) c`. Dans l'implémentation, on a mappé sur le constructeur de valeurs `Right`, mais pas sur `Left`. Pourquoi cela ? Eh bien, si on retourne à la définition d'`Either a b`, c'est un peu comme :

```
data Either a b = Left a | Right b
```

Si l'on voulait mapper une fonction sur les deux à la fois, `a` et `b` devraient avoir le même type. Ce que je veux dire, c'est que si on voulait mapper une fonction qui transforme une chaîne de caractères en chaîne de caractères, que `b` était une chaîne de caractères, mais que `a` était un nombre, ça ne marcherait pas. Aussi, en voyant ce que serait le type de `fmap` s'il n'opérait que sur des valeurs `Either`, on voit que le premier paramètre doit rester le même alors que le second peut changer, et c'est le constructeur de valeurs `Left` qui actualise ce premier paramètre.

Cela va bien de pair avec notre analogie de la boîte, si l'on imagine que `Left` est une boîte vide sur laquelle un message d'erreur indique pourquoi elle est vide.

Les maps de `Data.Map` peuvent aussi être faites foncteurs puisqu'elles contiennent (ou non !) des valeurs. Dans le cas de `Map k v`, `fmap` va mapper une fonction de type `v -> v'` sur une map de type `Map k v` et retourner une map de type `Map k v'`.

Notez que le `'` n'a pas de sémantique spéciale dans les noms des types, de même qu'il n'en avait pas dans les noms des valeurs. On l'utilise généralement pour dénoter des choses qui sont similaires mais un peu modifiées.

Essayez de deviner comment `Map k` est fait instance de `Functor` par vous-même !

Avec la classe `Functor`, on a vu comment les classes de types peuvent représenter des concepts d'ordre supérieur plutôt cool. On a aussi pratiqué un peu plus l'application partielle sur les types et la création d'instances. Dans un des chapitres suivants, on verra quelques lois qui s'appliquent sur les foncteurs.

Encore une chose ! Les foncteurs doivent suivre certaines lois afin qu'ils bénéficient de propriétés dont on dépendra sans trop s'en soucier. Si on fait `fmap (+1)` sur la liste `[1, 2, 3, 4]`, on obtient `[2, 3, 4, 5]`, et pas la liste renversée `[5, 4, 3, 2]`. Si on utilise `fmap (\a -> a)` (la fonction identité qui retourne son paramètre inchangé) sur une liste, on s'attend à retrouver la même liste en retour. Par exemple, si on donnait une mauvaise instance de foncteur à notre type `Tree`, alors en utilisant `fmap` sur un arbre où le sous-arbre gauche n'a que des éléments inférieurs au nœud racine et le sous-arbre droit n'a que des éléments supérieurs, on risquerait de produire un arbre qui n'a plus cette propriété. On verra les lois des foncteurs plus en détail dans un des chapitres suivants.

Sortes et un peu de type-fu

Les constructeurs de types prennent d'autres types en paramètres pour finalement produire des types concrets. Ça me rappelle un peu les fonctions, qui prennent des valeurs en paramètre pour produire des valeurs. On a vu que les constructeurs de types peuvent être appliqués partiellement (`Either String` est un constructeur de

types qui prend un type et produit un type concret, comme `Either String Int`), tout comme le peuvent les fonctions. C'est effectivement très intéressant. Dans cette section, nous allons voir comment définir formellement la manière dont les constructeurs de types sont appliqués aux types, tout comme nous avions défini formellement comment les fonctions étaient appliquées à des valeurs en utilisant des déclarations de type. **Vous n'avez pas nécessairement besoin de lire cette section pour continuer votre quête magique d'Haskell**, et si vous ne la comprenez pas, ne vous inquiétez pas. Cependant, comprendre ceci vous donnera une compréhension très poussée du système de types.

Donc, les valeurs comme `3`, `"YEAH"`, ou `takeWhile` (les fonctions sont aussi des valeurs, puisqu'on peut les passer et les retourner) ont chacune un type. Les types sont des petites étiquettes que les valeurs transportent afin qu'on puisse raisonner sur leur valeur. Mais les types ont eux-même leurs petites étiquettes, appelées **sortes**. Une sorte est plus ou moins le type d'un type. Ça peut sembler bizarre et déroutant, mais c'est en fait un concept très cool.

Que sont les sortes et à quoi servent-elles ? Eh bien, examinons la sorte d'un type à l'aide de la commande `:k` dans GHCi.

```
ghci> :k Int
Int :: *
```

Une étoile ? Comme c'est bizarre. Qu'est-ce que cela signifie ? Une `*` signifie que le type est un type concret. Un type concret est un type qui ne prend pas de paramètre de types, et les valeurs ne peuvent avoir que des types concrets. Si je devais lire `*` tout haut (ce qui n'a jamais été le cas), je dirais juste *étoile* ou *type*.

Ok, voyons maintenant le type de `Maybe`.

```
ghci> :k Maybe
Maybe :: * -> *
```

Le constructeur de types `Maybe` prend un type concret (comme `Int`) et retourne un type concret comme `Maybe Int`. Et c'est ce que nous dit cette sorte. Tout comme `Int -> Int` signifie qu'une fonction prend un `Int` et retourne un `Int`, `* -> *` signifie qu'un constructeur de types prend un type concret et retourne un type concret. Appliquons `Maybe` à un paramètre de type et voyons la sorte du résultat.

```
ghci> :k Maybe Int
Maybe Int :: *
```

Comme prévu ! On a appliqué `Maybe` à un paramètre de type et obtenu un type concret (c'est ce que `* -> *` signifie). Un parallèle (bien que non équivalent, les types et les sortes étant des choses différentes) à cela est, lorsque l'on fait `:t isUpper` et `:t isUpper 'A'`. `isUpper` a pour type `Char -> Bool` et `isUpper 'A'` a pour type `Bool`, parce que sa valeur est simplement `True`. Ces deux types ont tout de même pour sorte `*`.

On a utilisé `:k` sur un type pour obtenir sa sorte, comme on a utilisé `:t` sur une valeur pour connaître son type. Comme dit précédemment, les types sont les étiquettes des valeurs, et les sortes les étiquettes des types, et on peut voir des parallèles entre les deux.

Regardons une autre sorte.

```
ghci> :k Either
Either :: * -> * -> *
```

Aha, cela nous indique qu'`Either` prend deux types concrets en paramètres pour produire un type concret. Ça ressemble aussi à la déclaration de type d'une fonction qui prend deux choses et en retourne une troisième. Les constructeurs de types sont curryfiés (comme les fonctions), donc on peut les appliquer partiellement.

```
ghci> :k Either String
Either String :: * -> *
ghci> :k Either String Int
Either String Int :: *
```

Lorsqu'on voulait faire d'`Either` une instance de `Functor`, on a dû l'appliquer partiellement parce que `Functor` voulait des types qui attendent un paramètre, alors qu'`Either` en prend deux. En d'autres mots, `Functor` veut des types de sorte `* -> *` et nous avons dû appliquer partiellement `Either` pour obtenir un type de sorte `* -> *` au lieu de sa sorte originale `* -> * -> *`. Si on regarde à nouveau la définition de `Functor` :

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

on peut voir que `f` est utilisé comme un type qui prend un type concret et produit un type concret. On sait qu'il doit produire un type concret parce que ce type est utilisé comme valeur dans une fonction. Et de ça, on peut déduire que les types qui peuvent être amis avec `Functor` doivent avoir pour sorte `* -> *`.

Maintenant, faisons un peu de type-fu. Regardez cette classe de types que je vais inventer maintenant :

```
class Tofu t where
    tofu :: j a -> t a j
```

Wow, ça a l'air bizarre. Comment ferions-nous un type qui soit une instance de cette étrange classe de types ? Eh bien, regardons quelle sorte ce type devrait avoir. Puisque `j a` est utilisé comme le type d'une valeur que la fonction `tofu` prend en paramètre, `j a` doit avoir pour sorte `*`. En supposant que `a` a pour sorte `*`, alors `j` doit avoir pour sorte `* -> *`. On voit que `t` doit aussi produire un type concret et doit prendre deux types. Et sachant que `a` a pour sorte `*` et que `j` a pour sorte `* -> *`, on infère que `t` doit avoir pour sorte `* -> (* -> *) -> *`. Ainsi, il prend un type concret (`a`), un constructeur de types qui prend un type concret (`j`) et produit un type concret. Wow.

OK, faisons alors un type qui a pour sorte `* -> (* -> *) -> *`. Voici une façon d'y arriver.

```
data Frank a b  = Frank {frankField :: b a} deriving (Show)
```

Comment sait-on que ce type a pour sorte `* -> (* -> *) -> *` ? Eh bien, les champs des TAD (types abstraits de données) doivent contenir des valeurs, donc doivent avoir pour sorte `*`, évidemment. On suppose `*` pour `a`, ce qui veut dire que `b` prend un paramètre de type et donc sa sorte est `* -> *`. On connaît les sortes de `a` et `b`, et puisqu'ils sont les paramètres de `Frank`, on voit que `Frank` a pour sorte `* -> (* -> *) -> *`. Le premier `*` représente `a` et le `(* -> *)` représente `b`. Créons quelques valeurs de type `Frank` et vérifions leur type.

```
ghci> :t Frank {frankField = Just "HAHA"}
Frank {frankField = Just "HAHA"} :: Frank [Char] Maybe
ghci> :t Frank {frankField = Node 'a' EmptyTree EmptyTree}
Frank {frankField = Node 'a' EmptyTree EmptyTree} :: Frank Char Tree
ghci> :t Frank {frankField = "YES"}
Frank {frankField = "YES"} :: Frank Char []
```

Hmm. Puisque `frankField` a un type de la forme `a b`, ses valeurs doivent avoir des types de forme similaire. Ainsi, elles peuvent être `Just "HAHA"`, qui a pour type `Maybe [Char]` ou une valeur `['Y', 'E', 'S']`, qui a pour type `[Char]` (si on utilisait notre propre type liste, ce serait `List Char`). Et on voit que les types des valeurs `Frank` correspondent bien à la sorte de `Frank`. `[Char]` a pour sorte `*` et `Maybe` a pour sorte `* -> *`. Puisque pour être une valeur, elle doit avoir un type concret et donc entièrement appliqué, chaque valeur `Frank blah blaah` a pour sorte `*`.

Faire de `Frank` une instance de `Tofu` est plutôt facile. On voit que `tofu` prend un `j a` (par exemple un `Maybe Int`) et retourne un `t a j`. Si l'on remplaçait `j` par `Frank`, le type résultant serait `Frank Int Maybe`.

```
instance Tofu Frank where
    tofu x = Frank x
```

```
ghci> tofu (Just 'a') :: Frank Char Maybe
Frank {frankField = Just 'a'}
ghci> tofu ["HELLO"] :: Frank [Char] []
Frank {frankField = ["HELLO"]}
```

Pas très utile, mais bon pour l'entraînement de nos muscles de types. Encore un peu de type-fu. Mettons qu'on ait ce type de données :

```
data Barry t k p = Barry { yabba :: p, dabba :: t k }
```

Et maintenant, on veut créer une instance de `Functor`. `Functor` prend des types de sorte `* -> *` mais `Barry` n'a pas l'air de cette sorte. Quelle est la sorte de `Barry` ? Eh bien, on voit qu'il prend trois paramètres de types, donc ça va être `something -> something -> something -> *`. Il est prudent de considérer que `p` est un type concret et a pour sorte `*`. Pour `k`, on suppose `*`, et par extension, `t` a pour sorte `* -> *`. Remplaçons les *something* plus tôt par ces sortes, et l'on obtient `(* -> *) -> * -> * -> *`. Vérifions avec GHCi.

```
ghci> :k Barry
Barry :: (* -> *) -> * -> * -> *
```

Ah, on avait raison. Comme c'est satisfaisant. Maintenant, pour rendre ce type membre de `Functor` il nous faut appliquer partiellement les deux premiers paramètres de type de façon à ce qu'il nous reste un `* -> *`. Cela veut dire que le début de la déclaration d'instance sera : `instance Functor (Barry a b) where`. Si on regarde `fmap` comme si elle était faite spécifiquement pour `Barry`, elle aurait pour type

`fmap :: (a -> b) -> Barry c d a -> Barry c d b`, parce qu'on remplace juste les `f` de `Functor` par `Barry c d`. Le troisième paramètre de type de `Barry` devra changer, et on voit qu'il est convenablement placé dans son propre champ.

```
instance Functor (Barry a b) where
    fmap f (Barry {yabba = x, dabba = y}) = Barry {yabba = f x, dabba = y}
```

Et voilà ! On vient juste de mapper `f` sur le premier champ.

Dans cette section, on a vu un tour détaillé du fonctionnement des paramètres de types et on les a en quelque sorte formalisés avec les sortes, comme on avait formalisé les paramètres de fonctions avec des déclarations de type. On a vu qu'il existe des parallèles intéressants entre les fonctions et les constructeurs de types. Cependant, ce sont deux choses complètement différentes. En faisant du vrai Haskell, vous n'aurez généralement pas à travailler avec des sortes et à inférer des sortes à la main comme on a fait ici. Généralement, vous devez juste appliquer partiellement `* -> *` ou `*` à votre type personnalisé pour en faire une instance d'une des classes de types standard, mais il est bon de savoir comment et pourquoi cela fonctionne. Il est aussi intéressant de s'apercevoir que les types ont leur propre type. Encore une fois, vous n'avez pas à comprendre tout ce qu'on vient de faire pour continuer à lire, mais si vous comprenez comment les sortes fonctionnent, il y a des chances pour que vous ayez développé une compréhension solide du système de types d'Haskell.

← Modules | Table des matières | Entrées et sorties →

Entrées et Sorties

⇐ Créer nos propres types et classes de types | Table des matières | Résoudre des problèmes fonctionnellement ⇒

Nous avons mentionné qu'Haskell était un langage fonctionnel pur. Alors que dans des langages impératifs, on parvient généralement à faire quelque chose en donnant à l'ordinateur une série d'étapes à exécuter, en programmation fonctionnelle on définit plutôt ce que les choses sont. En Haskell, une fonction ne peut pas changer un état, comme par exemple le contenu d'une variable (quand une fonction change un état, on dit qu'elle a des *effets de bord*). La seule chose qu'une fonction peut faire en Haskell, c'est renvoyer un résultat basé sur les paramètres qu'on lui a donnés. Si une fonction est appelée deux fois avec les mêmes paramètres, elle doit retourner le même résultat. Bien que ça puisse paraître un peu limitant lorsqu'on vient d'un monde impératif, on a vu que c'est en fait plutôt sympa. Dans un langage impératif, vous n'avez aucune garantie qu'une fonction qui est sensée calculer des nombres ne va pas brûler votre maison, kidnapper votre chien et rayer votre voiture avec une patate pendant qu'elle calcule ces nombres. Par exemple, quand on a fait un arbre binaire de recherche, on n'a pas inséré un élément dans un arbre en modifiant l'arbre à sa place. Notre fonction pour insérer dans un arbre binaire retournait en fait un nouvel arbre, parce qu'elle ne peut pas modifier l'ancien.

Bien qu'avoir des fonctions incapables de changer d'état soit bien puisque cela nous aide à raisonner sur nos programmes, il y a un problème avec ça. Si une fonction ne peut rien changer dans le monde, comment est-elle sensée nous dire ce qu'elle a calculé ? Pour nous dire ce qu'elle a calculé, elle doit pouvoir changer l'état d'un matériel de sortie (généralement, l'état de notre écran), qui va ensuite émettre des photons qui voyageront jusqu'à notre cerveau pour changer l'état de notre esprit, mec.

Ne désespérez pas, tout n'est pas perdu. Il s'avère qu'Haskell a en fait un système très malin pour gérer ces fonctions qui ont des effets de bord, qui sépare proprement les parties de notre programme qui sont pures de celles qui sont impures, font tout le sale boulot comme parler au clavier ou à l'écran. Avec ces deux parties séparées, on peut toujours raisonner sur la partie pure du programme, et bénéficier de toutes les choses que la pureté offre, comme la paresse, la robustesse et la modularité, tout en communiquant efficacement avec le monde extérieur.

Hello, world!

Jusqu'ici, nous avions toujours chargé nos fonctions dans GHCi pour les tester et jouer avec elles. On a aussi exploré les fonctions de la bibliothèque standard de cette façon. Mais à présent, après environ huit chapitres, on va finalement écrire notre premier *vrai* programme Haskell ! Yay ! Et pour sûr, on va se faire ce bon vieux `"hello, world"`.

Hey ! Pour ce chapitre, je vais supposer que vous disposez d'un environnement à la Unix pour apprendre Haskell. Si vous êtes sous Windows, je suggère d'utiliser Cygwin, un environnement à la Linux pour Windows, autrement dit, juste ce qu'il vous faut.

Pour commencer, tapez ceci dans votre éditeur de texte favori :

```
main = putStrLn "hello, world"
```

On vient juste de définir un nom `main`, qui consiste à appeler `putStrLn` avec le paramètre `"hello, world"`. Ça semble plutôt banal, mais ça ne l'est pas, comme on va le voir bientôt. Sauvegardez ce fichier comme `helloworld.hs`.

Et maintenant, on va faire quelque chose sans précédent. On va réellement compiler notre programme ! Je suis tellement ému ! Ouvrez votre terminal et naviguez jusqu'au répertoire où `helloworld.hs` est placé et faites :

```
$ ghc --make helloworld
[1 of 1] Compiling Main             ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

Ok ! Avec de la chance, vous avez quelque chose comme ça et vous pouvez à présent lancer le programme en faisant `./helloworld`.

```
$ ./helloworld
```

```
hello, world
```

Et voilà, notre premier programme compilé qui affichait quelque chose dans le terminal. Comme c'est extraordinaire(ment ennuyeux) !

Examinons ce qu'on vient d'écrire. D'abord, regardons le type de la fonction `putStrLn`.

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

On peut lire le type de `putStrLn` ainsi : `putStrLn` prend une chaîne de caractères et retourne une **action I/O** qui a pour type de retour `()` (c'est-à-dire le tuple vide, aussi connu comme *unit*). Une action I/O est quelque chose qui, lorsqu'elle sera exécutée, va effectuer une action avec des effets de bord (généralement, lire en entrée ou afficher à l'écran) et contiendra une valeur de retour. Afficher quelque chose à l'écran n'a pas vraiment de valeur de retour significative, alors une valeur factice `()` est utilisée.

Le tuple vide est une valeur `()` qui a pour type `()`.

Donc, quand est-ce que cette action sera exécutée ? Eh bien, c'est ici que le `main` entre en jeu. Une action I/O sera exécutée lorsqu'on lui donne le nom `main` et qu'on lance le programme ainsi créé.

Que votre programme entier ne soit qu'une action I/O semble un peu limitant. C'est pourquoi on peut utiliser la notation *do* pour coller ensemble plusieurs actions I/O en une seule. Regardez l'exemple suivant :

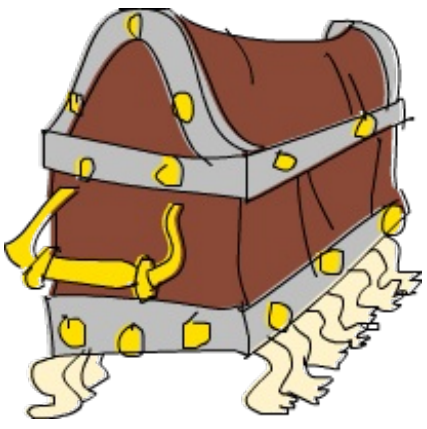
```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

Ah, intéressant, une nouvelle syntaxe ! Et celle-ci se lit presque comme un programme impératif. Si vous compilez cela et l'essayez, cela se comportera certainement conformément à ce que vous attendez. Remarquez qu'on a dit *do*, puis on a aligné une série d'étapes, comme en programmation impérative. Chacune de ces étapes est une action I/O. En les mettant ensemble avec la syntaxe *do*, on les a collées en une seule action I/O. L'action obtenue a pour type `IO ()`, parce que c'est le type de la dernière action à l'intérieur du collage.

À cause de ça, `main` a toujours la signature de type `main :: IO something`, où `something` est un type concret. Par convention, on ne spécifie généralement pas la déclaration de type de `main`.

Une chose intéressante qu'on n'a pas rencontrée avant est à la troisième ligne, qui dit `name <- getLine`. On dirait qu'elle lit une ligne depuis l'entrée et la stocke dans une variable `name`. Est-ce le cas ? Examinons le type de `getLine`.

```
ghci> :t getLine
getLine :: IO String
```



Aha, o-kay. `getLine` est une action I/O qui contient un résultat de type `String`. Ça tombe sous le sens, parce qu'elle attendra que l'utilisateur tape quelque chose dans son terminal, et ensuite, cette frappe sera représentée comme une chaîne de caractères. Mais qu'est-ce qui se passe dans `name <- getLine` alors ? Vous pouvez lire ceci ainsi : **effectue l'action I/O `getLine` puis lie la valeur résultante au nom `name`**. `getLine` a pour type `IO String`, donc `name` aura pour type `String`. Vous pouvez imaginer une action I/O comme une boîte avec des petits pieds qui sortirait dans le monde réel et irait faire quelque chose là-bas (comme des graffitis sur les murs) et reviendrait peut-être avec une valeur. Une fois qu'elle a attrapé une valeur pour vous, le seul moyen d'ouvrir la boîte pour en récupérer le contenu est d'utiliser la construction `<-`. Et si l'on sort une valeur d'une action I/O, on ne peut le faire qu'à l'intérieur d'une autre action I/O. C'est ainsi qu'Haskell parvient à séparer proprement les parties pure et impure du code. `getLine` est en un sens impure, parce que sa valeur de retour n'est pas garantie d'être la même lorsqu'on l'exécute deux fois. C'est pourquoi elle est en quelque sorte *tachée* par le constructeur de types `IO`, et on ne peut récupérer cette donnée que dans du code I/O. Et puisque le code I/O est taché aussi, tout calcul qui dépend d'une valeur tachée I/O renverra un résultat taché.

Quand je dis *taché*, je ne veux pas dire taché de façon à ce que l'on ne puisse plus jamais utiliser le résultat contenu dans l'action I/O dans un code pur. Non, on dé-tache temporairement la donnée dans l'action I/O lorsqu'on la lie à un nom. Quand on fait `name <- getLine`, `name` est une chaîne de caractères normale, parce qu'elle représente ce qui est dans la boîte. On peut avoir une fonction très compliquée qui, mettons, prend votre nom (une chaîne de caractères normale) et un paramètre, et vous donne votre fortune et votre futur basé sur votre nom. On peut faire cela :

```
main = do
```

```
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ "Read this carefully, because this is your future: " ++ tellFortune name
```

et `tellFortune` (ou n'importe quelle fonction à laquelle on passe `name`) n'a pas besoin de savoir quoi que ce soit à propos d'I/O, c'est une simple fonction `String -> String` !

Regardez ce bout de code. Est-il valide ?

```
nameTag = "Hello, my name is " ++ getLine
```

Si vous avez répondu non, offrez-vous un cookie. Si vous avez dit oui, buvez un verre de lave en fusion. Non, je blague ! La raison pour laquelle ça ne marche pas, c'est parce que `++` requiert que ses deux paramètres soient des listes du même type. Le premier paramètre a pour type `String` (ou `[Char]` si vous voulez), alors que `getLine` a pour type `IO String`. On ne peut pas concaténer une chaîne de caractères et une action I/O. On doit d'abord récupérer le résultat de l'action I/O pour obtenir une valeur de type `String`, et le seul moyen de faire ceci c'est de faire `name <- getLine` dans une autre action I/O. Si l'on veut traiter des données impures, il faut le faire dans un environnement impur. Ainsi, la trace de l'impureté se propage comme le fléau des morts-vivants, et il est dans notre meilleur intérêt de restreindre les parties I/O de notre code autant que faire se peut.

Chaque action I/O effectuée encapsule son résultat en son sein. C'est pourquoi notre précédent programme aurait pu s'écrire ainsi :

```
main = do
    foo <- putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

Cependant, `foo` aurait juste pour valeur `()`, donc écrire ça serait un peu discutable. Remarquez qu'on n'a pas lié le dernier `putStrLn`. C'est parce que, dans un bloc *do*, **la dernière action ne peut pas être liée à un nom** contrairement aux précédentes. Nous verrons pourquoi c'est le cas un peu plus tard quand nous nous aventurerons dans le monde des monades. Pour l'instant, vous pouvez imaginer que le bloc *do* extrait automatiquement la valeur de la dernière action et la lie à son propre résultat.

À part la dernière ligne, toute ligne d'un bloc *do* qui ne lie pas peut être réécrite avec une liaison. Ainsi, `putStrLn "BLAH"` peut être réécrit comme `_ <- putStrLn "BLAH"`. Mais ça ne sert à rien, donc on enlève le `<-` pour les actions I/O dont le résultat ne nous importe pas, comme `putStrLn something`.

Les débutants pensent parfois que faire

```
name = getLine
```

va lire l'entrée et lier la valeur de cela à `name`. Eh bien non, tout ce que cela fait, c'est de donner un autre nom à l'action I/O `getLine`, ici, `name`. Rappelez-vous, pour obtenir une valeur d'une action I/O, vous devez le faire de l'intérieur d'une autre action I/O et la liant à un nom via `<-`.

Les actions I/O ne seront exécutées que si elles ont pour nom `main` ou lorsqu'elles sont dans une grosse action I/O composée par un bloc *do* en train d'être exécutée. On peut utiliser les blocs *do* pour coller des actions I/O, puis utiliser cette action I/O dans un autre bloc *do*, et ainsi de suite. De toute façon, elles ne seront exécutées que si elles finissent par se retrouver dans `main`.

Oh, j'oubliais, il y a un autre cas où une action I/O est exécutée. C'est lorsque l'on tape cette action I/O dans GHCi et qu'on presse Entrée.

```
ghci> putStrLn "HEEY"
HEEY
```

Même lorsque l'on tape juste des nombres ou qu'on appelle une fonction dans GHCi et qu'on tape Entrée, il va l'évaluer (autant que nécessaire) et appeler `show` sur le résultat, puis afficher cette chaîne de caractères sur le terminal en appelant implicitement `putStrLn`.

Vous vous souvenez des liaisons *let* ? Si ce n'est pas le cas, rafraîchissez-vous l'esprit en lisant cette section. Elles sont de la forme `let bindings in expression`, où `bindings` sont les noms à donner aux expressions et `expression` est l'expression évaluée qui peut voir ces liaisons. On a aussi dit que dans les listes en compréhension, la partie *in* n'était pas nécessaire. Eh bien, on peut aussi les utiliser dans les blocs *do* comme on le faisait dans les listes en compréhension. Regardez :

```
import Data.Char

main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
```

```
        bigLastName = map toUpper lastName
    putStrLn $ "hey " ++ bigFirstName ++ " " ++ bigLastName ++ ", how are you?"
```

Remarquez comme les actions I/O dans le bloc *do* sont alignées. Notez également comme le *let* est aligné avec les actions I/O, et les noms du *let* sont alignés les uns aux autres. C'est important, parce que l'indentation importe en Haskell. Ici, on a fait `map toUpper firstName`, qui transforme quelque chose comme `"John"` en une chaîne de caractères bien plus cool comme `"JOHN"`. On a lié cette chaîne de caractères majuscules à un nom, puis utilisé ce nom dans une chaîne de caractères qu'on a affichée sur le terminal plus tard.

Vous vous demandez peut-être quand utiliser `<-` et quand utiliser des liaisons *let* ? Eh bien, souvenez-vous, `<-` sert (pour l'instant) à effectuer des actions I/O et lier leur résultat à des noms. `map toUpper firstName`, cependant, n'est pas une action I/O. C'est une expression pure en Haskell. Donc, utilisez `<-` quand vous voulez lier le résultat d'une action I/O à un nom et utiliser une liaison *let* pour lier une expression pure à un nom. Si on avait fait quelque chose comme `let firstName = getLine`, on aurait juste donné à l'action I/O `getLine` un nouveau nom, sans avoir exécuté cette action.

À présent, on va faire un programme qui lit continuellement une ligne et l'affiche avec les mots renversés. Le programme s'arrêtera si on entre une ligne vide. Voici le programme :

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

Pour vous rendre compte de ce qu'il fait, essayez de l'exécuter avant qu'on s'intéresse au code.

Astuce : Pour lancer un programme, vous pouvez soit le compiler puis lancer l'exécutable produit en faisant `ghc --make helloworld` puis `./helloworld`, ou bien vous pouvez utiliser la commande `runhaskell` ainsi : `runhaskell helloworld.hs` et votre programme sera exécuté à la volée.

Tout d'abord, regardons la fonction `reverseWords`. C'est une fonction normale qui prend une chaîne de caractères comme `"hey there man"` et appelle `words` pour produire une liste de mots comme `["hey", "there", "man"]`. Puis, on mappe `reverse` sur la liste, résultant en `["yeh", "ereht", "nam"]`, puis on regroupe cette liste en une chaîne en utilisant `unwords` et le résultat final est `"yeh ereht nam"`. Voyez comme on a utilisé la composition de fonctions ici. Sans cela, on aurait dû écrire `reverseWords st = unwords (map reverse (words st))`.

Quid de `main` ? D'abord, on récupère une ligne du terminal avec `getLine` et on nomme cette ligne `line`. Maintenant, on a une expression conditionnelle. Souvenez-vous, en Haskell, tout *if* doit avoir un *else* parce que chaque expression est une valeur. On a fait le *if* de sorte que lorsque la condition est vraie (dans notre cas, la ligne est non vide), on exécute une action I/O, et lorsqu'elle est vide, l'action I/O sous le *else* est exécutée à la place. C'est pour cela que dans des blocs *do* I/O, les *if* doivent avoir la forme `if condition then I/O action else I/O action`.

Regardons d'abord du côté de la clause *else*. Puisqu'on doit avoir une action I/O après le *else*, on crée un bloc *do* pour coller des actions en une. Vous pouvez aussi écrire cela :

```
else (do
    putStrLn $ reverseWords line
    main)
```

Cela rend plus explicite le fait que le bloc *do* peut être vu comme une action I/O, mais c'est plus moche. Peu importe, dans le bloc *do*, on appelle `reverseWords` sur la ligne obtenue de `getLine`, puis on l'affiche au terminal. Après cela, on exécute `main`. C'est un appel récursif, et c'est bon, parce que `main` est bien une action I/O. En un sens, on est de retour au début du programme.

Maintenant, que se passe-t-il lorsque `null line` est vrai ? Dans ce cas, ce qui suit le *then* est exécuté. Si on regarde, on voit qu'il y a `then return ()`. Si vous avez utilisé des langages impératifs comme C, Java ou Python, vous vous dites certainement que vous savez déjà ce que `return` fait, et il se peut que vous ayez déjà sauté ce long paragraphe. Eh bien, voilà le détail qui tue : **le `return` de Haskell n'a vraiment rien à voir avec le `return` de la plupart des autres langages !** Il a le même nom, ce qui embrouille beaucoup de monde, mais en réalité, il est bien différent. Dans les langages impératifs, `return` termine généralement l'exécution de la méthode ou sous-routine, en rapportant une valeur à son appelant. En Haskell (plus spécifiquement, dans les action I/O), il crée une action I/O à partir d'une valeur pure. Si vous repensez à l'analogie de la boîte faite précédemment, il prend une valeur et la met dans une boîte. L'action I/O résultante ne fait en réalité rien, mais encapsule juste cette valeur comme son résultat. Ainsi, dans un contexte d'I/O, `return "HAHA"` aura pour type `IO String`. Quel est le but de transformer une valeur pure en une action I/O qui ne fait rien ? Pourquoi salir notre programme d'`IO` plus que nécessaire ? Eh bien, il nous fallait une action I/O à exécuter dans le cas où la ligne en entrée était vide. C'est pourquoi on a créé une fausse action I/O qui ne fait rien, en écrivant

`return ()`.

Utiliser `return` ne cause pas la fin de l'exécution du bloc *do* I/O ou quoi que ce soit du genre. Par exemple, ce programme va gentiment s'exécuter jusqu'au bout de la dernière ligne :

```
main = do
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
    putStrLn line
```

Tout ce que ces `return` font, c'est créer des actions I/O qui ne font rien de spécial à part encapsuler un résultat, et les résultats sont ici jetés puisqu'on ne les lie pas à des noms. On peut utiliser `return` en combinaison avec `<-` pour lier des choses à des noms.

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

Comme vous voyez, `return` est un peu l'opposé de `<-`. Alors que `return` prend une valeur et l'enveloppe dans une boîte, `<-` prend une boîte, l'exécute, et en extrait la valeur pour la lier à un nom. Faire cela est un peu redondant, puisque l'on dispose des liaisons *let* dans les blocs *do*, donc on préfère :

```
main = do
    let a = "hell"
        b = "yeah"
    putStrLn $ a ++ " " ++ b
```

Quand on fait des blocs *do* I/O, on utilise principalement `return` soit parce que l'on a besoin de créer une action I/O qui ne fait rien, ou bien parce qu'on ne veut pas que l'action créée par le bloc ait la valeur de la dernière action qui la compose, auquel cas on place un `return` tout à la fin avec le résultat qu'on veut obtenir de cette action composée.

Un bloc *do* peut aussi avoir une seule action I/O. Dans ce cas, c'est équivalent à écrire seulement l'action. Certains vont préférer écrire `then do return ()` parce que le *else* avait un *do*.

Avant de passer aux fichiers, regardons quelques fonctions utiles pour faire des I/O.

`putStr` est un peu comme `putStrLn` puisqu'il prend une chaîne de caractères en paramètre et retourne une action I/O qui affiche cette chaîne sur le terminal, seulement `putStr` ne va pas à la ligne après avoir affiché la chaîne, alors que `putStrLn` va à la ligne.

```
main = do   putStr "Hey, "
            putStr "I'm "
            putStrLn "Andy!"
```

```
$ runhaskell putstr_test.hs
Hey, I'm Andy!
```

Sa signature de type est `putStr :: String -> IO ()`, donc le résultat encapsulé est *unit*. Une valeur inutile, donc ça ne sert à rien de la lier.

`putChar` prend un caractère et retourne une action I/O qui l'affiche sur le terminal.

```
main = do   putChar 't'
            putChar 'e'
            putChar 'h'
```

```
$ runhaskell putchar_test.hs
teh
```

`putStr` est en fait défini récursivement à l'aide de `putChar`. Le cas de base de `putStr` est la chaîne vide, donc si on affiche une chaîne vide, elle retourne juste une action I/O qui ne fait rien à l'aide de `return ()`. Si la chaîne n'est pas vide, elle affiche son premier caractère avec `putChar`, puis affiche le reste de la chaîne avec `putStr`.

```haskell
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
    putChar x
    putStr xs
```

Voyez comme on peut utiliser la récursivité dans les I/O, comme dans du code pur. Comme pour un code pur, on définit un cas de base, et on réfléchit à ce que le résultat doit être. C'est une action qui affiche le première caractère puis affiche le reste de la chaîne.

`print` prend une valeur de n'importe quel type instance de `Show` (donc, qu'on sait représenter sous forme de chaîne de caractères), et appelle `show` sur cette valeur pour la transformer en chaîne, puis affiche cette chaîne sur le terminal. En gros, c'est seulement `putStrLn . show`. Elle lance d'abord `show` sur la valeur, puis nourrit `putStrLn` du résultat, qui va alors retourner une action I/O qui affichera notre valeur.

```haskell
main = do   print True
            print 2
            print "haha"
            print 3.2
            print [3,4,3]
```

```
$ runhaskell print_test.hs
True
2
"haha"
3.2
[3,4,3]
```

Comme vous le voyez, cette fonction est très pratique. Vous vous souvenez qu'on a dit que les actions I/O n'étaient exécutées que lorsqu'elles sont sous `main` ou quand on essaie de les évaluer dans l'invite GHCi ? Quand on tape une valeur (comme `3` ou `[1, 2, 3]`) et qu'on tape Entrée, GHCi utilise en fait `print` sur cette valeur pour l'afficher dans notre terminal !

```
ghci> 3
3
ghci> print 3
3
ghci> map (++"!") ["hey","ho","woo"]
["hey!","ho!","woo!"]
ghci> print (map (++"!") ["hey","ho","woo"])
["hey!","ho!","woo!"]
```

Quand on veut afficher des chaînes de caractères, on utilise généralement `putStrLn` parce qu'on ne veut pas des guillemets autour d'elles, mais pour afficher toutes les autres valeurs, `print` est généralement utilisé.

`getChar` est une action I/O qui lit un caractère de l'entrée. Ainsi, sa signature de type est `getChar :: IO Char`, parce que le résultat contenu dans l'action I/O a pour type `Char`. Notez que les caractères sont mis en tampon, ainsi la lecture des caractères n'aura pas lieu tant que l'utilisateur ne tape pas Entrée.

```haskell
main = do
    c <- getChar
    if c /= ' '
        then do
            putChar c
            main
        else return ()
```

Ce programme semble lire un caractère et vérifier si c'est un espace. Si c'est le cas, il s'arrête, sinon il affiche le caractère et recommence. C'est un peu ce qu'il fait, mais pas forcément comme on s'y attend. Regardez :

```
$ runhaskell getchar_test.hs
hello sir
hello
```

La seconde ligne est l'entrée. On tape `hello sir` et on appuie sur Entrée. À cause de la mise en tampon, l'exécution du programme ne démarre que lorsqu'on tape Entrée, et pas à chaque caractère tapé. Mais une fois qu'on presse Entrée, il agit sur ce qu'on vient de taper. Essayez de jouer avec ce programme pour vous rendre compte !

La fonction `when` est dans `Control.Monad` (pour l'utiliser, faites `import Control.Monad`). Elle est intéressante parce que, dans un bloc *do*, on dirait qu'elle contrôle le flot du programme, alors qu'en fait c'est une fonction tout à fait normale. Elle prend une valeur booléenne et une action I/O. Si la valeur booléenne est

`True`, elle retourne l'action I/O qu'on lui a passée. Si c'est `False`, elle retourne `return ()`, donc une action I/O qui ne fait rien. Voici comment on pourrait réécrire le code précédent dans lequel on présentait `getChar`, en utilisant `when` :

```
import Control.Monad

main = do
    c <- getChar
    when (c /= ' ') $ do
        putChar c
        main
```

Comme vous voyez, c'est utile pour encapsuler le motif `if something then do some I/O action else return ()`.

`sequence` prend une liste d'actions I/O et retourne une action I/O qui exécutera ces actions l'une après l'autre. Le résultat contenu dans cette action I/O sera la liste de tous les résultats de toutes les actions I/O exécutées. Sa signature de type est `sequence :: [IO a] -> IO [a]`. Faire ceci :

```
main = do
    a <- getLine
    b <- getLine
    c <- getLine
    print [a,b,c]
```

est équivalent à faire :

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

Donc `sequence [getLine, getLine, getLine]` crée une action I/O qui va exécuter `getLine` trois fois. Si on lie cette action à un nom, le résultat sera une liste de tous les résultats, dans notre cas, une liste de trois choses que l'utilisateur a tapées dans l'invite.

Un motif courant avec `sequence` consiste à mapper des fonctions comme `print` ou `putStrLn` sur des listes. Faire `map print [1, 2, 3, 4]` ne créera pas une action I/O. Cela va créer une liste d'actions I/O, car c'est équivalent à `[print 1, print 2, print 3, print 4]`. Si on veut transformer une liste d'actions I/O en une action I/O, il faut la séquencer.

```
ghci> sequence (map print [1,2,3,4,5])
1
2
3
4
5
[(),(),(),(),()]
```

C'est quoi ce `[(), (), (), (), ()]` à la fin ? Eh bien, quand on évalue une action I/O dans GHCi, elle est exécutée et le résultat est affiché, à moins que ce ne soit `()`, auquel cas il n'est pas affiché. C'est pourquoi évaluer `putStrLn "hehe"` dans GHCi affiche seulement `hehe` (parce que le résultat contenu dans cette action est `()`). Mais quand on fait `getLine` dans GHCi, le résultat de cette action est affiché dans GHCi, parce que `getLine` a pour type `IO String`.

Puisque mapper une fonction qui retourne une action I/O sur une liste puis séquencer cette liste est tellement commun, les fonctions utilitaires `mapM` et `mapM_` ont été introduites. `mapM` prend une fonction et une liste, mappe la fonction sur la liste et séquence le tout. `mapM_` fait pareil, mais jette le résultat final. On utilise généralement `mapM_` lorsqu'on se fiche du résultat de nos actions séquencées.

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
ghci> mapM_ print [1,2,3]
1
2
3
```

`forever` prend une action I/O et retourne une action I/O qui répète la première indéfiniment. Elle est dans `Control.Monad`. Ce programme va demander indéfiniment à l'utilisateur une entrée, et va la recracher en MAJUSCULES :

```
import Control.Monad
import Data.Char
```

```
main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

`forM` (dans `Control.Monad`) est comme `mapM`, mais avec les paramètres dans l'ordre inverse. Le premier paramètre est la liste, et le second la fonction à mapper sur cette liste, qui sera finalement séquencée. Pourquoi est-ce utile ? Eh bien, en étant un peu créatif sur l'utilisation des lambdas et de la notation *do*, on peut faire des choses comme :

```
import Control.Monad

main = do
    colors <- forM [1,2,3,4] (\a -> do
        putStrLn $ "Which color do you associate with the number " ++ show a ++ "?"
        color <- getLine
        return color)
    putStrLn "The colors that you associate with 1, 2, 3 and 4 are: "
    mapM putStrLn colors
```

Le `(\a -> do ...)` est une fonction qui prend un nombre et retourne une action I/O. On l'a mise entre parenthèses, sinon le lambda croit que les deux dernières actions I/O sur les deux dernières lignes lui appartiennent également. Remarquez qu'on fait `return color` dans le bloc *do*. On fait cela de manière à ce que l'action I/O définie par ce bloc *do* contienne pour résultat notre couleur. En fait on n'avait pas besoin de faire ça ici, puisque `getLine` contient déjà ce résultat. Faire `color <- getLine` suivi de `return color` consiste seulement à sortir le résultat de `getLine` et à le remettre dans la boîte juste après, donc il suffit de faire juste `getLine`. Le `forM` (appelé avec ses deux paramètres) produit une action I/O dont on lie le résultat à `colors`. `colors` est une liste tout ce qu'il y a de plus normale, qui contient des chaînes de caractères. À la fin, on affiche toutes ces couleurs en faisant `mapM putStrLn colors`.

Vous pouvez imaginer `forM` comme signifiant : crée une action I/O pour tous les éléments de cette liste. Ce que l'action I/O fait peut dépendre de la valeur de l'élément de la liste à partir duquelle elle a été créée. Finalement, exécute toutes ces actions et lie leur résultat à quelque chose. Notez qu'on n'est pas forcé de le lier, on pourrait juste le jeter.

```
$ runhaskell form_test.hs
Which color do you associate with the number 1?
white
Which color do you associate with the number 2?
blue
Which color do you associate with the number 3?
red
Which color do you associate with the number 4?
orange
The colors that you associate with 1, 2, 3 and 4 are:
white
blue
red
orange
```

On aurait pu le faire sans `forM`, mais c'est plus lisible avec. Généralement, on utilise `forM` pour mapper et séquencer des actions que l'on définit au vol avec la notation *do*. Dans la même veine, on aurait pu remplacer la dernière ligne par `forM colors putStrLn`.

Dans cette section, on a vu les bases de l'entrée-sortie. On a aussi vu ce que les actions I/O sont, comment elles nous permettent de faire des entrées-sorties, et quand elles sont réellement exécutées. Pour en remettre une couche, les actions I/O sont des valeurs presque comme les autres en Haskell. On peut les passer en paramètre à des fonctions, et des fonctions peuvent en retourner comme résultat. Ce qui est spécial à leur propos, c'est que si elles se retrouvent sous la fonction `main` (ou dans GHCi), elles sont exécutées. Et c'est alors qu'elles se retrouvent à écrire des choses sur votre écran ou à jouer Yakety Sax dans vos haut-parleurs. Chaque action I/O peut encapsuler un résultat qui vous dira ce qu'elle a trouvé dans le monde réel.

Ne pensez pas que `putStrLn` est une fonction qui prend une chaîne de caractères et l'affiche à l'écran. Pensez-y comme une fonction qui prend une chaîne de caractères et retourne une action I/O. Cette action I/O affichera, lorsqu'elle sera exécutée, de magnifiques poèmes dans votre terminal.

Fichiers et flots

`getChar` est une action I/O qui lit un caractère du terminal. `getLine` est une action I/O qui lit une ligne du terminal. Ces deux sont plutôt simples, et la plupart des langages de programmation ont des fonctions ou des instructions semblables à ces actions I/O. Mais à présent, rencontrons `getContents`. `getContents` est une action I/O qui lit toute l'entrée standard jusqu'à rencontrer un caractère de fin de fichier. Son type est `getContents :: IO String`. Ce qui est cool avec `getContents`, c'est qu'elle effectue une entrée paresseuse. Quand on fait `foo <- getContents`, elle ne va pas lire toute l'entrée d'un coup, la stocker en

mémoire, puis la lier à `foo`. Non, elle est paresseuse ! Elle dira : "*Ouais ouais, je lirai l'entrée du terminal plus tard, quand tu en auras besoin !*".

`getContents` est très utile quand on veut connecter la sortie d'un programme à l'entrée de notre programme. Si vous ne savez pas comment cette connexion (à base de *tubes*) fonctionne dans les systèmes type Unix, voici un petit aperçu. Créons un fichier texte qui contient ce haiku :

```
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
```

Ouais, le haiku est pourri, mais bon ? Si quelqu'un connaît un tutoriel de haikus, je suis preneur.

Bien, souvenez-vous de ce programme qu'on a écrit en introduisant la fonction `forever`. Il demandait à l'utilisateur une ligne, et lui retournait en MAJUSCULES, puis recommençait, indéfiniment. Pour vous éviter de remonter tout là-haut, je vous la remets ici :

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

Sauvegardons ce programme comme `capslocker.hs` ou ce que vous voulez, et compilons-le. À présent, on va utiliser un tube Unix pour donner notre fichier texte à manger à notre petit programme. On va utiliser le programme GNU *cat*, qui affiche le fichier donné en argument. Regardez-moi ça, booyaka !

```
$ ghc --make capslocker
[1 of 1] Compiling Main             ( capslocker.hs, capslocker.o )
Linking capslocker ...
$ cat haiku.txt
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
$ cat haiku.txt | ./capslocker
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
capslocker <stdin>: hGetLine: end of file
```

Comme vous voyez, connecter la sortie d'un programme (dans notre cas c'était *cat*) à l'entrée d'un autre (ici *capslocker*) est fait à l'aide du caractère `|`. Ce qu'on a fait ici est à peu près équivalent à lancer *capslocker*, puis taper notre haiku dans le terminal, avant d'envoyer le caractère de fin de fichier (généralement en tapant Ctrl-D). C'est comme lancer *cat haiku.txt* et dire : "Attends, n'affiche pas ça dans le terminal, va le dire à *capslocker* plutôt !".

Donc ce qu'on fait principalement avec cette utilisation de `forever` c'est prendre l'entrée et la transformer en une sortie. C'est pourquoi on peut utiliser `getContents` pour rendre ce programme encore plus court et joli :

```
import Data.Char

main = do
    contents <- getContents
    putStr (map toUpper contents)
```

On lance l'action I/O `getContents` et on nomme la chaîne produite `contents`. Puis on mappe `toUpper` sur cette chaîne, et on l'affiche sur le terminal. Gardez en tête que puisque les chaînes de caractères sont simplement des listes, qui sont paresseuses, et que `getContents` est aussi paresseuse en entrée-sortie, cela ne va pas essayer de lire tout le contenu et de le stocker en mémoire avant d'afficher la version en majuscules. Plutôt, cela va afficher la version en majuscule au fur et à mesure de la lecture, parce que cela ne lira une ligne de l'entrée que lorsque ce sera nécessaire.

```
$ cat haiku.txt | ./capslocker
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
```

Cool, ça marche. Et si l'on essaie de lancer *capslocker* et de taper les lignes nous-même ?

```
$ ./capslocker
```

```
hey ho
HEY HO
lets go
LETS GO
```

On a quitté le programme en tapant Ctrl-D. Plutôt joli ! Comme vous voyez, cela affiche nos caractères en majuscules ligne après ligne. Quand le résultat de **`getContents`** est lié à **`contents`**, ce n'est pas représenté en mémoire comme une vraie chaîne de caractères, mais plutôt comme une promesse de produire cette chaîne de caractères en temps voulu. Quand on mappe **`toUpper`** sur **`contents`**, c'est aussi une promesse de mapper la fonction sur le contenu une fois qu'il sera disponible. Et finalement, quand **`putStr`** est exécuté, il dit à la promesse précédente : "*Hé, j'ai besoin des lignes en majuscules !*". Celle-ci n'a pas encore les lignes, alors elle demande à **`contents`** : "*Hé, pourquoi tu n'irais pas chercher ces lignes dans le terminal à présent ?* ". C'est à ce moment que **`getContents`** va vraiment lire le terminal et donner une ligne au code qui a demandé d'avoir quelque chose de tangible. Ce code mappe alors **`toUpper`** sur la ligne et la donne à **`putStr`**, qui l'affiche. Puis **`putStr`** dit : "*Hé, j'ai besoin de la ligne suivante, allez !*" et tout ceci se répète jusqu'à ce qu'il n'y ait plus d'entrée, comme l'indique le caractère de fin de fichier.

Faisons un programme qui prend une entrée et affiche seulement les lignes qui font moins de 10 caractères de long. Observez :

```
main = do
    contents <- getContents
    putStr (shortLinesOnly contents)

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

On a fait la partie I/O du programme la plus courte possible. Puisque notre programme est censé lire une entrée, et afficher quelque chose en fonction de l'entrée, on peut l'implémenter en lisant le contenu d'entrée, puis en exécutant une fonction pure sur ce contenu, et en affichant le résultat que la fonction a renvoyé.

La fonction **`shortLinesOnly`** fonctionne ainsi : elle prend une chaîne de caractères comme **`"short\nlooooooooooooooong\nshort again"`**. Cette chaîne a trois lignes, dont deux courtes et une au milieu plus longue. La fonction lance **`lines`** sur cette chaîne, ce qui la convertit en **`["short", "looooooooooooooong", "short again"]`**, qu'on lie au nom **`allLines`**. Cette liste de chaînes est ensuite filtrée pour ne garder que les lignes de moins de 10 caractères, produisant **`["short", "short again"]`**. Et finalement, **`unlines`** joint cette liste en une seule chaîne, donnant **`"short\nshort again"`**. Testons cela.

```
i'm short
so am i
i am a looooooooooong line!!!
yeah i'm long so what hahahaha!!!!!!
short line
looooooooooooooooooooooooooong
short
```

```
$ ghc --make shortlinesonly
[1 of 1] Compiling Main             ( shortlinesonly.hs, shortlinesonly.o )
Linking shortlinesonly ...
$ cat shortlines.txt | ./shortlinesonly
i'm short
so am i
short
```

On connecte le contenu de *shortlines.txt* à l'entrée de *shotlinesonly* et en sortie, on obtient les lignes courtes.

Ce motif qui récupère une chaîne en entrée, la transforme avec une fonction, puis écrit le résultat est tellement commun qu'il existe une fonction qui rend cela encore plus simple, appelée **`interact`**. **`interact`** prend une fonction de type **`String -> String`** en paramètre et retourne une action I/O qui va lire une entrée, lancer cette fonction dessus, et afficher le résultat. Modifions notre programme en conséquence :

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

Juste pour montrer que ceci peut être fait en beaucoup moins de code (bien que ce soit moins lisible) et pour démontrer notre maîtrise de la composition de fonctions, on va retravailler cela.

```
main = interact $ unlines . filter ((<10) . length) . lines
```

Wow, on a réduit cela à juste une ligne, c'est plutôt cool !

`interact` peut être utilisé pour faire des programmes à qui l'on connecte un contenu en entrée et qui affichent un résultat en conséquence, ou bien pour faire des programmes qui semblent attendre une entrée de l'utilisateur, et rendent un résultat basé sur la ligne entrée, puis prend une autre ligne, etc. Il n'y a en fait pas de réelle distinction entre les deux, ça dépend juste de la façon dont l'utilisateur veut utiliser le programme.

Faisons un programme qui lit continuellement une ligne et nous dit si la ligne était un palindrome ou pas. On pourrait utiliser `getLine` pour lire une ligne, dire à l'utilisateur si c'est un palindrome, puis lancer `main` à nouveau. Mais il est plus simple d'utiliser `interact`. Quand vous utilisez `interact`, pensez à tout ce que vous avez besoin de faire pour transformer une entrée en la sortie désirée. Dans notre cas, on doit remplacer chaque ligne de l'entrée par soit `"palindrome"`, soit `"not a palindrome"`. Donc on doit écrire une fonction qui transforme quelque chose comme `"elephant\nABCBA\nwhatever"` en `"not a palindrome\npalindrome\nnot a palindrome"`. Faisons ça !

```
respondPalindromes contents = unlines (map (\xs -> if isPalindrome xs then "palindrome" else "not a palindrome") (lines content
    where   isPalindrome xs = xs == reverse xs
```

Écrivons ça sans point.

```
respondPalindromes = unlines . map (\xs -> if isPalindrome xs then "palindrome" else "not a palindrome") . lines
    where   isPalindrome xs = xs == reverse xs
```

Plutôt direct. D'abord, on change quelque chose comme `"elephant\nABCBA\nwhatever"` en `["elephant", "ABCBA", "whatever"]`, puis on mappe la lambda sur ça, donnant `["not a palindrome", "palindrome", "not a palindrome"]` et enfin `unlines` joint cette liste en une seule chaîne de caractères. À présent, on peut faire :

```
main = interact respondPalindromes
```

Testons ça :

```
$ runhaskell palindromes.hs
hehe
not a palindrome
ABCBA
palindrome
cookie
not a palindrome
```

Bien qu'on ait fait un programme qui transforme une grosse chaîne de caractères en une autre, ça se passe comme si on avait fait un programme qui faisait cela ligne par ligne. C'est parce qu'Haskell est paresseux et veut afficher la première ligne de la chaîne de caractères en sortie, mais ne peux pas parce qu'il ne l'a pas encore. Dès qu'on lui donne une ligne en entrée, il va l'afficher en sortie. On termine le programme en envoyant un caractère de fin de fichier.

On peut aussi utiliser ce programme en connectant simplement un fichier en entrée. Disons qu'on ait ce fichier :

```
dogaroo
radar
rotor
madam
```

et qu'on le sauvegarde comme `words.txt`. Voici ce qu'on obtient en le connectant en entrée de notre programme :

```
$ cat words.txt | runhaskell palindromes.hs
not a palindrome
palindrome
palindrome
palindrome
```

Encore une fois, on obtient la même sortie que si l'on avait tapé les mots dans le programme nous-même. Seulement, on ne voit pas l'entrée affichée puisqu'elle est venue du programme et qu'on ne l'a pas tapée ici.

Vous voyez probablement comment les entrées-sorties paresseuses fonctionnent à présent, et comment on peut en tirer parti. Vous pouvez penser simplement à ce que doit être la sortie en fonction de l'entrée, et écrire une fonction qui effectue cette transformation. En entrée-sortie paresseuse, rien n'est consommé en entrée tant que ce n'est pas absolument nécessaire, par exemple parce que l'on souhaite afficher un résultat qui dépend de cette entrée.

Jusqu'ici, nous avons travaillé avec les entrées-sorties en affichant des choses dans le terminal, et en lisant des choses depuis ce dernier. Mais pourquoi pas lire et écrire des fichiers ? En quelque sorte on a déjà fait ça. Une manière de penser au terminal est de se dire que c'est un fichier (un peu spécial). On peut appeler le fichier de ce qu'on tape dans le terminal `stdin`, et le fichier qui s'affiche dans notre terminal `stdout`, pour *entrée standard* et *sortie standard*, respectivement. En gardant cela à l'esprit, on va voir qu'écrire ou lire dans un fichier est très similaire à écrire ou lire dans l'entrée ou la sortie standard.

On va commencer avec un programme très simple qui ouvre un fichier nommé *girlfriend.txt*, qui contient un vers du tube d'Avril Lavigne numéro 1 *Girlfriend*, et juste afficher cela dans le terminal. Voici *girlfriend.txt* :

```
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

Et voici notre programme :

```
import System.IO

main = do
    handle <- openFile "girlfriend.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

En le lançant, on obtient le résultat attendu :

```
$ runhaskell girlfriend.hs
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

Regardons cela ligne par ligne. La première ligne contient juste quatre exclamations, pour attirer notre attention. Dans la deuxième ligne, Avril nous indique qu'elle n'apprécie pas notre partenaire romantique actuelle. La troisième ligne sert à mettre en emphase cette désapprobation, alors que la quatrième ligne suggère que nous devrions chercher une nouvelle petite amie.

Hum, regardons plutôt le programme ligne par ligne ! Notre programme consiste en plusieurs actions I/O combinées ensemble par un bloc *do*. Dans la première ligne du bloc *do*, on remarque une fonction nouvelle nommée *openFile*. Voici sa signature de type : `openFile :: FilePath -> IOMode -> OI Handle`. Si vous lisez ceci tout haut, cela donne : `openFile` prend un chemin vers un fichier et un `IOMode` et retourne une action I/O qui va ouvrir le fichier et encapsule pour résultat la poignée vers le fichier associé.

`FilePath` est juste un synonyme du type `String`, simplement défini comme :

```
type FilePath = String
```

`IOMode` est défini ainsi :

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Tout comme notre type qui représente sept valeurs pour les jours de la semaine, ce type est une énumération qui représente ce qu'on veut faire avec le fichier ouvert. Très simple. Notez bien que le type est `IOMode` et non pas `IO Mode`. Ce dernier serait le type d'une action I/O qui contient une valeur d'un type `Mode`, alors qu'`IOMode` est juste une simple énumération.

Finalement, la fonction retourne une action I/O qui ouvre le fichier spécifié dans le mode indiqué. Si l'on lie cette action à un nom, on obtient un `Handle`. Une valeur de type `Handle` représente notre fichier ouvert. On utilise cette poignée pour savoir de quel fichier on parle. Il serait stupide d'ouvrir un fichier mais ne pas lier cette poignée à un nom, puisqu'on ne pourrait alors pas savoir quel fichier on a ouvert. Dans notre cas, on lie la poignée au nom `handle`.

À la prochaine ligne, on voit une fonction nommée `hGetContents`. Elle prend un `Handle`, afin de savoir de quel fichier on veut récupérer le contenu, et retourne une `IO String` - une action I/O qui contient en résultat le contenu du fichier.

Cette fonction est proche de `getContents`. La seule différence, c'est que `getContents` lit automatiquement depuis l'entrée standard (votre terminal), alors que `hGetContents` prend une poignée pour savoir quel fichier elle doit lire. Pour le reste, elles font la même chose. Et tout comme `getContents`, `hGetContents` ne va pas lire tout le fichier et le stocker en mémoire, mais lire ce qui sera nécessaire pour progresser. C'est très cool parce qu'on peut traiter `contents` comme le contenu de tout le fichier, alors qu'il n'est pas réellement chargé en entier dans la mémoire. Donc, même pour un énorme fichier, faire `hGetContents` ne va pas étouffer notre mémoire, parce que le fichier est lu seulement quand c'est nécessaire.

Notez bien la différence entre la poignée, utilisée pour identifier le fichier, et le contenu de ce fichier, liés respectivement dans ce programme aux noms `handle` et `contents`. La poignée est juste quelque chose qui indique quel est notre fichier. Si vous imaginez que votre système de fichiers est un énorme livre, dont les fichiers sont des chapitres, la poignée est une sorte de marque-page qui montre quel chapitre vous souhaitez lire (ou écrire), alors que le contenu est celui du chapitre.

Avec `putStr contents`, on affiche seulement le contenu sur la sortie standard, puis on fait `hClose`, qui prend une poignée et retourne une action I/O qui ferme le fichier. Il faut fermer le fichier vous-même après l'avoir ouvert avec `openFile` !

Un autre moyen de faire tout ça consiste à utiliser la fonction `withFile`, qui a pour type `withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a`. Elle prend un chemin vers un fichier, un `IOMode` et une fonction qui prend une poignée et retourne une action I/O. Elle retourne alors une action I/O qui ouvre le fichier, applique notre fonction, puis ferme le fichier. Le résultat retourné par cette action I/O est le même que le résultat retourné par la fonction qu'on lui fournit. Ça peut vous sembler compliqué, mais c'est en fait très simple, surtout avec des lambdas, voici le précédent exemple réécrit avec `withFile` :

```
import System.IO

main = do
    withFile "girlfriend.txt" ReadMode (\handle -> do
        contents <- hGetContents handle
        putStr contents)
```

Comme vous le voyez, c'est très similaire au code précédent. `(\handle -> ...)` est une fonction qui prend une poignée et retourne une action I/O, et on le fait généralement comme ça avec une lambda. La raison pour laquelle `withFile` doit prendre une fonction qui retourne une action I/O, plutôt que de prendre directement une action I/O, est que l'action I/O ne saurait pas sur quel fichier elle doit agir autrement. Ainsi, `withFile` ouvre le fichier et passe la poignée à la fonction qu'on lui a donnée. Elle récupère ainsi une action I/O, et crée à son tour une action I/O qui est comme la précédente mais ferme le fichier à la fin. Voici comment coder notre propre fonction `withFile` :

```
withFile' :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile' path mode f = do
    handle <- openFile path mode
    result <- f handle
    hClose handle
    return result
```

On sait que le résultat sera une action I/O, donc on peut commencer par écrire un *do*. D'abord, on ouvre le fichier pour récupérer une poignée. Puis, on applique notre fonction sur cette poignée pour obtenir une action I/O qui fait son travail sur ce fichier. On lie le résultat de cette action à `result`, on ferme la poignée, et on fait `return result`. En retournant le résultat encapsulé dans l'action I/O obtenue par `f`, notre action I/O composée encapsule le même résultat que celui de `f handle`. Ainsi, si `f handle` retourne une action I/O qui lit un nombre de lignes de l'entrée standard, les écrit dans un fichier, et encapsule pour résultat le nombre de lignes qu'elle a lues, alors en l'utilisant avec `withFile'`, l'action I/O résultante aurait également pour résultat le nombre de lignes lues.

Tout comme on a `hGetContents` qui fonctionne comme `getContents` mais pour un fichier, il y a aussi `hGetLine`, `hPutStr`, `hPutStrLn`, `hGetChar`, etc. Elles fonctionnent toutes comme leur équivalent sans h, mais prennent une poignée en paramètre et opèrent sur le fichier correspondant plutôt que l'entrée ou la sortie standard. Par exemple : `putStrLn` est une fonction qui prend une chaîne de caractères et retourne une action I/O qui affiche cette chaîne dans le terminal avec un retour à la ligne à la fin. `hPutStrLn` prend une poignée et une chaîne et retourne une action I/O qui écrit cette chaîne dans le fichier associé à la poignée, suivie d'un retour à la ligne. Dans la même veine, `hGetLine` prend une poignée et retourne une action I/O qui lit une ligne de ce fichier.

Charger des fichiers et traiter leur contenu comme des chaînes de caractères est tellement commun qu'on a ces trois fonctions qui facilitent le travail :

`readFile` a pour signature `readFile :: FilePath -> IO String`. Souvenez-vous que `FilePath` est juste un nom plus joli pour `String`. `readFile` prend un chemin vers un fichier et retourne une action I/O qui lit ce fichier (paresseusement bien sûr) et lie son contenu à une chaîne de caractères. C'est généralement plus pratique que de faire `openFile`, de lier le retour à une poignée, puis de faire `hGetContents`. Ainsi, le précédent exemple se réécrit :

```
import System.IO

main = do
    contents <- readFile "girlfriend.txt"
    putStr contents
```

Puisqu'on ne récupère pas de poignée, on ne peut pas fermer le fichier manuellement, donc Haskell le fait tout seul quand on utilise **`readFile`**.

`writeFile` a pour type **`writeFile :: FilePath -> String -> IO ()`**. Elle prend un chemin vers un fichier et une chaîne de caractères à écrire dans ce fichier et retourne une action I/O qui effectuera l'écriture. Si le fichier existe déjà, il sera écrasé complètement avant que l'écriture ne commence. Voici comment transformer *girlfriend.txt* en une version en MAJUSCULES et écrire cette version dans *girlfriendcaps.txt*.

```
import System.IO
import Data.Char

main = do
    contents <- readFile "girlfriend.txt"
    writeFile "girlfriendcaps.txt" (map toUpper contents)
```

```
$ runhaskell girlfriendtocaps.hs
$ cat girlfriendcaps.txt
HEY! HEY! YOU! YOU!
I DON'T LIKE YOUR GIRLFRIEND!
NO WAY! NO WAY!
I THINK YOU NEED A NEW ONE!
```

`appendFile` a la même signature de type que **`writeFile`**, mais elle ne tronque pas le fichier s'il existe déjà, à la place elle écrit à la suite du contenu du fichier existant.

Mettons qu'on ait un fichier *todo.txt* qui a une tâche par ligne de chose qu'on doit penser à faire. Faisons un programme qui prend une ligne de l'entrée standard et l'ajoute à notre liste de tâches.

```
import System.IO

main = do
    todoItem <- getLine
    appendFile "todo.txt" (todoItem ++ "\n")
```

```
$ runhaskell appendtodo.hs
Iron the dishes
$ runhaskell appendtodo.hs
Dust the dog
$ runhaskell appendtodo.hs
Take salad out of the oven
$ cat todo.txt
Iron the dishes
Dust the dog
Take salad out of the oven
```

On a dû ajouter le **`"\n"`** à la fin de chaque ligne, parce que **`getLine`** ne met pas de caractère pour aller à la ligne à la fin.

Ooh, encore une chose. On a dit que **`contents <- hGetContents handle`** ne lisait pas tout le fichier d'un coup pour le stocker en mémoire. C'est une entrée-sortie paresseuse, donc faire :

```
main = do
    withFile "something.txt" ReadMode (\handle -> do
        contents <- hGetContents handle
        putStr contents)
```

c'est comme connecter un tube depuis le fichier vers la sortie. Comme on peut penser aux listes comme à des flots, on peut aussi penser les fichiers comme des flots. Ceci va lire une ligne à la fois et l'afficher sur le terminal au passage. Vous vous demandez peut-être quelle est la taille de ce tube alors ? Combien de fois va-t-on accéder au disque ? Eh bien, pour les fichiers textes, le tampon par défaut est par ligne généralement. Cela veut dire que la plus petite unité du fichier lue à la fois est une ligne. C'est pourquoi dans ce cas il lit une ligne, affiche le résultat, lit la prochaine ligne, affiche le résultat, etc. Pour des fichiers binaires, la mise en tampon par défaut est d'un bloc. Cela veut dire que le fichier sera lu bout par bout. La taille de ces bouts de fichiers dépend de ce que votre système d'exploitation considère comme une taille cool.

Vous pouvez contrôler comment la mise en tampon est effectuée en utilisant la fonction **`hSetBuffering`**. Elle prend une poignée et un **`BufferMode`** et retourne

une action I/O qui définit le mode de mise en tampon. `BufferMode` est un simple type de données énuméré, et les valeurs possibles sont : `NoBuffering`, `LineBuffering` ou `BlockBuffering (Maybe Int)`. Le `Maybe Int` permet de préciser la taille des blocs, en octets. Si c'est `Nothing`, alors le système d'exploitation détermine la taille du bloc. `NoBuffering` signifie que les caractères sont lus un par un. `NoBuffering` n'est généralement pas efficace, parce qu'il doit accéder le disque tout le temps.

Voici notre code précédent, sauf qu'il ne lit pas ligne par ligne, mais en blocs de 2048 octets.

```
main = do
    withFile "something.txt" ReadMode (\handle -> do
        hSetBuffering handle $ BlockBuffering (Just 2048)
        contents <- hGetContents handle
        putStr contents)
```

Lire des fichiers en plus gros morceaux peut aider à minimiser les accès au disque, ou lorsque notre fichier est en fait une ressource réseau lente.

On peut aussi utiliser `hFlush`, qui est une fonction qui prend une poignée et retourne une action I/O qui va vider le tampon du fichier associé à la poignée. Quand on est en mise en tampon par ligne, le tampon est vidé à chaque nouvelle ligne. Quand on est en mise en tampon par bloc, le tampon est vidé à chaque bloc. Il est aussi vidé lorsqu'on ferme la poignée. Cela signifie que lorsqu'on atteint un caractère de nouvelle ligne, le mécanisme de lecture (ou d'écriture) rapporte toutes les données qu'il a lues jusqu'ici. Mais on peut utiliser `hFlush` pour le forcer à rapporter ce qu'il a lu à n'importe quel instant. Après avoir vidé le tampon en écriture, les données sont disponibles pour n'importe quel autre programme qui accède au fichier en lecture en même temps.

Pensez à la mise en tampon par bloc comme cela : votre cuvette de toilettes est faite pour se vider dès qu'elle contient un litre d'eau. Vous pouvez donc commencer à la remplir d'eau, et dès que vous atteignez un litre, l'eau est automatiquement vidée, et les données que vous aviez mises dans cette eau sont lues. Mais vous pouvez aussi vider la cuvette manuellement en actionnant la chasse d'eau. Cela force la cuvette à se vider, et toute l'eau (les données) est lue. Au cas où vous n'auriez pas saisi, tirer la chasse d'eau est une métaphore pour `hFlush`. Ce n'est pas une analogie très sensationnelle en termes de critères d'analogies dans la programmation, mais je voulais une analogie avec le monde réel d'un objet qu'on peut vider, pour ma chute.

On a déjà fait un programme pour ajouter un nouvel élément à une liste de tâches dans *todo.txt*, à présent, faisons un programme qui retire une tâche. Je vais coller le code, et on va le regarder en détail par la suite ensemble, pour que vous voyiez que c'est très simple. On va utiliser quelques nouvelles fonctions sorties de `System.Directory` et une nouvelle fonction de `System.IO`, mais j'expliquerai cela en temps voulu.

Bon, voici le programme qui retire un élément de *todo.txt* :

```
import System.IO
import System.Directory
import Data.List

main = do
    handle <- openFile "todo.txt" ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStrLn "These are your TO-DO items:"
    putStr $ unlines numberedTasks
    putStrLn "Which one do you want to delete?"
    numberString <- getLine
    let number = read numberString
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile "todo.txt"
    renameFile tempName "todo.txt"
```

Au départ, on ouvre seulement *todo.txt* en lecture, et on lie sa poignée au nom `handle`.

Ensuite, on utilise une fonction que nous n'avons pas encore rencontrée, qui vient de `System.IO` - `openTempFile`. Son nom est assez expressif. Elle prend un chemin vers un répertoire temporaire, et un préfixe de nom, et ouvre un fichier temporaire. On a utilisé `"."` pour le dossier temporaire, parce que `.` indique le répertoire courant sur à peu près tous les systèmes d'exploitation. On a utilisé `"temp"` pour le préfixe de nom du fichier temporaire, ce qui signifie que le fichier sera nommé *temp* suivi de caractères aléatoires. Cette fonction retourne une action I/O qui crée le fichier temporaire, et le résultat de cette action est une paire : le nom du fichier créé et sa poignée. On pouvait simplement ouvrir un fichier *todo2.txt* ou quelque chose comme ça, mais il vaut mieux ouvrir un fichier temporaire à l'aide d'`openTempFile` pour être certain de ne pas en écraser un autre.

On n'a pas utilisé `getCurrentDirectory` pour récupérer le dossier courant avant de le passer à `openTempFile` parce que `.` indique le répertoire courant sous les systèmes Unix ainsi que sous Windows.

Puis, on lie le contenu de *todo.txt* au nom `contents`. Ensuite, on coupe cette chaîne de caractères en une liste de chaînes de caractères, chacune contenant

une ligne. `todoTasks` est à présent sous la forme `["Iron the dishes", "Dust the dog", "Take salad out of the oven"]`. On zippe les nombres 0 et suivants avec cette liste à l'aide d'une fonction qui prend un nombre, comme `3`, et une chaîne comme `"hey"` et retourne `"3 - hey"`, ainsi, `numberedTasks` est `["0 - Iron the dishes", "1 - Dust the dog" ...`. On joint cette liste de chaînes en une simple chaîne avec `unlines` et on affiche cette chaîne sur le terminal. Notez qu'on aurait pu faire `mapM putStrLn numberedTasks` ici.

On demande ensuite à l'utilisateur laquelle il souhaite effacer. Mettons qu'il souhaite effacer la numéro 1, `Dust the dog`, donc il tape `1`. `numberString` est à présent `"1"`, et puisqu'on veut un nombre, et pas une chaîne de caractères, on utilise `read` sur cela pour obtenir `1` et on le lie à `number`.

Souvenez-vous de `delete` et `!!` de `Data.List`. `!!` retourne l'élément d'une liste à l'indice donné, et `delete` supprime la première occurence d'un élément dans une liste et retourne une nouvelle liste sans cette occurence. `(todoTasks !! number)` (`number` étant `1`) retourne `"Dust the dog"`. On lie `todoTasks` à laquelle on a supprimé la première occurence de `"Dust the dog"` à `newTodoItems`, puis on joint cela en une seule chaîne avec `unlines` avant de l'écrire dans notre fichier temporaire. L'ancien fichier n'est pas modifié et le fichier temporaire contient toutes les lignes de l'ancien, à l'exception de celle qu'on a effacée.

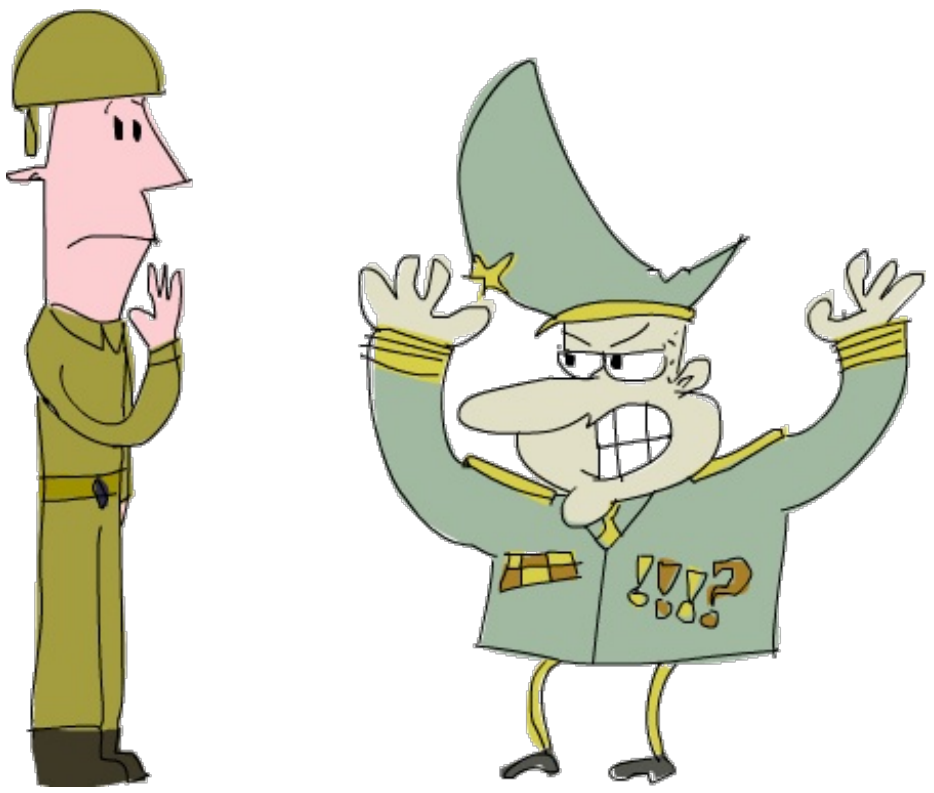
Après cela, on ferme les deux fichiers, on supprime l'ancien avec `removeFile`, qui, comme vous pouvez le voir, prend un chemin vers un fichier et le supprime. Après avoir supprimé le vieux *todo.txt*, on utilise `renameFile` pour renommer le fichier temporaire en *todo.txt*. Faites attention, `removeFile` et `renameFile` (qui sont dans `System.Directory`) prennent des chemins de fichiers, et pas des poignées.

Et c'est tout ! On aurait pu faire cela encore plus court, mais on a fait très attention à ne pas écraser de fichier existant et à demander poliment au système d'exploitation de nous dire où l'on pouvait mettre notre fichier temporaire. Essayons à présent !

```
$ runhaskell deletetodo.hs
These are your TO-DO items:
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
Which one do you want to delete?
1

$ cat todo.txt
Iron the dishes
Take salad out of the oven

$ runhaskell deletetodo.hs
These are your TO-DO items:
0 - Iron the dishes
1 - Take salad out of the oven
Which one do you want to delete?
0

$ cat todo.txt
Take salad out of the oven
```

Arguments de ligne de commande

Gérer les arguments de ligne de commande est plutôt nécessaire si vous voulez faire un script ou une application à lancer depuis le terminal. Heureusement, la bibliothèque standard Haskell a un moyen très sympa pour récupérer les arguments de ligne de commande du programme.



Dans la section précédente, on a fait un programme qui ajoute un élément à une liste de tâches et un programme pour retirer une tâche. Il y a deux problèmes avec l'approche choisie. La première, c'est qu'on a codé en dur le nom du fichier. On a en quelque sorte décidé que le fichier serait nommé *todo.txt* et que l'utilisateur n'aura jamais à gérer plus d'une liste de tâches.

Un moyen de résoudre ce problème consiste à demander à chaque fois quel fichier l'utilisateur veut modifier lorsqu'il utilise nos programmes. On a utilisé cette approche quand on a voulu savoir quel élément l'utilisateur voulait supprimer. Ça marche, mais ce n'est pas optimal, parce que cela requiert que l'utilisateur lance le programme, attende que le programme lui demande quelque chose, et ensuite indique le fichier au programme. Ceci est appelé un programme interactif, et le problème est le suivant : que faire si l'on souhaite automatiser l'exécution du programme, comme avec un script ? Il est plus dur de faire un script qui interagit avec un programme que de faire un script qui appelle un ou même plusieurs programmes.

C'est pourquoi il est parfois préférable que l'utilisateur dise au programme ce qu'il veut lorsqu'il lance le programme, plutôt que le programme demande à l'utilisateur des choses une fois lancé. Et quel meilleur moyen pour l'utilisateur d'indiquer ce qu'il veut au programme que de donner des arguments en ligne de commande !

Le module **`System.Environment`** a deux actions I/O cool. L'une est **`getArgs`**, qui a pour type **`getArgs :: IO [String]`** et est une action I/O qui va récupérer les arguments du programme et les encapsuler dans son résultat sous forme d'une liste. **`getProgName`** a pour type **`getProgName :: IO String`** et est une action I/O qui contient le nom du programme.

Voici un petit programme qui démontre leur fonctionnement :

```
import System.Environment
import Data.List

main = do
   args <- getArgs
   progName <- getProgName
   putStrLn "The arguments are:"
   mapM putStrLn args
   putStrLn "The program name is:"
   putStrLn progName
```

On lie **`getArgs`** et **`getProgName`** à **`args`** et **`progName`**. On dit **`The arguments are:`** et ensuite, pour chaque argument dans **`args`**, on applique **`putStrLn`**. Finalement, on affiche aussi le nom du programme. Compilons cela comme **`arg-test`**.

```
$ ./arg-test first second w00t "multi word arg"
The arguments are:
first
second
w00t
multi word arg
The program name is:
arg-test
```

Bien. Armé de cette connaissance, vous pouvez créer des applications en ligne de commande plutôt cool. En fait, créons-en une dès maintenant. Dans la section précédente, nous avions créé deux programmes séparés pour ajouter des tâches et en supprimer. Maintenant, on va faire cela en un seul programme, et ce qu'il fera dépendra des arguments. On va aussi le faire de manière à ce qu'il puisse opérer sur différents fichiers, pas seulement *todo.txt*.

On va l'appeler *todo* ("à faire") et il servira à faire (haha !) trois choses différentes :

- Voir des tâches
- Ajouter des tâches
- Supprimer des tâches

On se fichera un peu des erreurs d'entrée pour l'instant.

Notre programme sera fait de façon à ce que si l'on souhaite ajouter la tâche **`Find the magic sword of power`** au fichier *todo.txt*, on ait simplement à faire **`todo add todo.txt "Find the magic sword of power"`** dans notre terminal. Pour voir les tâches, on fera **`todo view todo.txt`** et pour supprimer la tâche qui a pour indice 2, on fera **`todo remove todo.txt 2`**.

On va commencer par créer une liste associative de résolution. Ce sera une simple liste associative qui aura pour clés des arguments de ligne de commande, et pour valeurs une fonction correspondante. Ces fonctions auront toute pour type **`[String] -> IO ()`**. Elles prendront en paramètre la liste des arguments, et retourneront une action I/O qui fera l'affichage, l'ajout, la suppression, etc.

```
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: [(String, [String] -> IO ())]
dispatch =  [ ("add", add)
            , ("view", view)
            , ("remove", remove)
            ]
```

Il nous reste encore à définir **`main`**, **`add`**, **`view`** et **`remove`**, commençons par **`main`** :

```
main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    action args
```

D'abord, on récupère les arguments et on les lie à **`(command:args)`**. Si vous vous souvenez de votre filtrage par motif, cela signifie que le premier argument est

lié à `command` et que le reste est lié à `args`. Si on appelle notre programme en faisant `todo add todo.txt "Spank the monkey"`, `command` sera `"add"` et `args` sera `["todo.xt", "Spank the monkey"]`.

À la prochaine ligne, on cherche notre commande dans la liste de résolution. Puisque `"add"` pointe vers `add`, on récupère `Just add` en résultat. On utilise à nouveau un filtrage par motifs pour sortir notre fonction du `Maybe`. Que se passe-t-il si notre commande ne fait pas partie de la liste de résolution ? Eh bien la résolution retournera `Nothing`, mais comme on a dit qu'on ne se souciait pas trop de ça, le filtrage va échouer et notre programme va lancer une exception.

Finalement, on appelle notre fonction `action` avec le reste de la liste des arguments. Cela retourne une action I/O qui ajoute un élément, affiche les éléments ou supprime un élément, et puisque cette action fait partie du bloc *do* de `main`, elle sera exécutée. Si on suit notre exemple concret, jusqu'ici `action` vaut `add`, et sera appelée avec `args` (donc `["todo.txt", "Spank the monkey"]`) et retournera une action I/O qui ajoute `Spank the monkey` à *todo.txt*.

Génial ! Il ne reste plus qu'à implémenter `add`, `view` et `remove`. Commençons par `add` :

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
```

Si on appelle notre programme avec `todo add todo.txt "Spank the monkey"`, le `"add"` sera lié à `command` dans le premier filtrage du bloc `main`, alors que `["todo.txt", "Spank the monkey"]` sera passé à la fonction obtenue dans la liste de résolution. Donc, puisqu'on ne se soucie pas des mauvaises entrées, on filtre simplement cela contre une liste à deux éléments, et on retourne une action I/O qui ajoute cette ligne à la fin du fichier, avec un caractère de retour à la ligne.

Ensuite, implémentons la fonctionnalité d'affichage. Si on veut voir les éléments d'un fichier, on fait `todo view todo.txt`. Donc dans le premier filtrage par motif, `command` sera `"view"` et `args` sera `["todo.txt"]`.

```
view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStr $ unlines numberedTasks
```

On a déjà fait à peu près la même chose dans le programme qui n'effaçait les tâches que quand on les avait affichées afin que l'utilisateur en choisisse une à supprimer, seulement ici, on ne fait qu'afficher.

Et enfin, on va implémenter `remove`. Ce sera très similaire au programme qui effaçait une tâche, donc si vous ne comprenez pas comment la suppression se passe ici, allez relire l'explication sous ce programme. La différence principale, c'est qu'on ne code pas le nom *todo.txt* en dur mais qu'on l'obtient en argument. Aussi, on ne demande pas à l'utilisateur de choisir un numéro puisqu'on l'obtient également en argument.

```
remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

On a ouvert le fichier basé sur `fileName` et ouvert un fichier temporaire, supprimé la ligne avec l'indice donné par l'utilisateur, écrit cela dans le fichier temporaire, supprimé le fichier original et renommé le fichier temporaire en `fileName`.

Voilà le programme final en entier, dans toute sa splendeur !

```
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: [(String, [String] -> IO ())]
dispatch =  [ ("add", add)
            , ("view", view)
            , ("remove", remove)
            ]

main = do
```

```
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    action args

add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")

view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStr $ unlines numberedTasks

remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

Pour résumer notre solution : on a fait une liste associative de résolution qui associe à chaque commande une fonction qui prend des arguments de la ligne de commande et retourne une action I/O. On regarde ce qu'est la commande, et en fonction de ça on résout l'appel sur la fonction appropriée de la liste de résolution. On appelle cette fonction avec le reste des arguments pour obtenir une action I/O qui fera l'action attendue et ensuite on exécute cette action !

Dans d'autres langages, on aurait pu implémenter ceci avec un gros *switch case* ou ce genre de structure, mais utiliser des fonctions d'ordre supérieure nous permet de demander à la liste de résolution de nous donner une fonction, puis de demander à cette fonction une action I/O correspondant à nos arguments.

Testons cette application !

```
$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven

$ ./todo add todo.txt "Pick up children from drycleaners"

$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
3 - Pick up children from drycleaners

$ ./todo remove todo.txt 2

$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Pick up children from drycleaners
```

Une autre chose cool à propos de ça, c'est qu'il est très facile d'ajouter de nouvelles fonctionnalités. Ajoutez simplement une entrée dans la liste de résolution et implémentez la fonction correspondante, les doigts dans le nez ! Comme exercice, vous pouvez implémenter la fonction `bump` qui prend un fichier et un numéro de tâche, et retourne une action I/O qui pousse cette tâche en tête de la liste de tâches.

Vous pourriez aussi faire échouer ce programme plus gracieusement dans le cas d'une entrée mal formée (par exemple, si quelqu'un lance `todo UP YOURS HAHAHAHA`) en faisant une action I/O qui rapporte une erreur (disons, `errorExit :: IO ()`), puis en vérifiant si l'entrée est erronée, et le cas échéant, en utilisant cette action. Un autre moyen est d'utiliser les exceptions, qu'on va bientôt rencontrer.

Aléatoire

Souvent quand on programme, on a besoin de données aléatoires. Par exemple, lorsque vous faites un jeu où un dé doit être lancé, ou quand vous voulez générer des entrées pour tester votre programme. Des données aléatoires peuvent avoir plein d'utilisations en programmation. En fait, je devrais dire pseudo-aléatoire, puisqu'on sait tous que la seule source de vrai aléatoire est un singe

sur un monocycle tenant un fromage dans une main et ses fesses dans l'autre. Dans cette section, on va voir comment Haskell génère des données quasiment aléatoires.

Dans la plupart des langages de programmation, vous avez des fonctions qui vous donnent un nombre aléatoire. Chaque fois que vous appelez cette fonction, vous obtenez (on l'espère) un nombre aléatoire différent. Quid d'Haskell ? Eh bien, souvenez-vous, Haskell est un langage fonctionnel pur. Cela signifie qu'il a la propriété de transparence référentielle. Ce que CELA signifie, c'est qu'une fonction à laquelle on donne les mêmes paramètres plusieurs fois retournera toujours le même résultat. C'est très cool, parce que ça nous permet de raisonner différemment à propos de nos programmes, et de retarder l'évaluation jusqu'à ce qu'elle soit nécessaire. Si j'appelle une fonction, je peux être certain qu'elle ne va pas faire des choses folles avant de me donner son résultat. Tout ce qui importe, c'est son résultat. Cependant, cela rend un peu difficile les choses dans le cas des nombres aléatoires. Si j'ai une fonction comme ça :

```
randomNumber :: (Num a) => a
randomNumber = 4
```

Ce n'est pas très utile comme fonction de nombre aléatoire parce qu'elle retourne toujours `4`, bien que je puisse vous garantir que ce 4 est totalement aléatoire puisque j'ai lancé un dé pour le déterminer.

Comment les autres langages font-ils des nombres apparemment aléatoires ? Eh bien, ils prennent différentes informations de l'ordinateur, comme l'heure actuelle, combien et comment vous avez déplacé votre souris, quels genres de bruits vous faites devant votre ordinateur, et mélangent tout ça pour vous donner un nombre qui a l'air aléatoire. La combinaison de ces facteurs (l'aléatoire) est probablement différente à chaque instant, donc vous obtenez un nombre aléatoire différent.

Ah. Donc en Haskell, on peut faire un nombre aléatoire en faisant une fonction qui prend en paramètre cet aléatoire et, basé sur ceci, retourne un nombre (ou un autre type de données).

Pénétrez dans le module `System.Random`. Il a toutes les fonctions qui satisfont notre besoin d'aléatoire. Plongeons donc dans l'une des fonctions qu'il exporte, j'ai nommé `random`. Voici son type : `random :: (RandomGen g, Random a) => g -> (a, g)`. Ouah ! Que de nouvelles classes de types dans cette déclaration ! La classe `RandomGen` est pour les types qui peuvent agir comme des sources d'aléatoire. La classe de types `Random` est pour les choses qui peuvent avoir une valeur aléatoire. Un booléen peut prendre une valeur aléatoire, soit `True` soit `False`. Un nombre peut prendre une pléthore de différentes valeurs aléatoires. Est-ce qu'une fonction peut prendre une valeur aléatoire ? Je ne pense pas, probablement pas ! Si l'on essaie de traduire la déclaration de `random`, cela donne quelque chose comme : elle prend un générateur aléatoire (c'est notre source d'aléatoire) et retourne une valeur aléatoire et un nouveau générateur aléatoire. Pourquoi retourne-t-elle un nouveau générateur en plus de la valeur aléatoire ? Eh bien, on va voir ça dans un instant.

Pour utiliser notre fonction `random`, il nous faut obtenir un de ces générateurs aléatoires. Le module `System.Random` exporte un type cool, `StdGen`, qui est une instance de la classe `RandomGen`. On peut soit créer notre `StdGen` nous-même, soit demander au système de nous en faire un basé sur une multitude de choses aléatoires.

Pour créer manuellement un générateur aléatoire, utilisez la fonction `mkStdGen`. Elle a pour type `mkStdGen :: Int -> StdGen`. Elle prend un entier et en fonction de celui-ci, nous donne un générateur aléatoire. Ok, essayons d'utiliser `random` et `mkStdGen` en tandem pour obtenir un nombre (pas très aléatoire…).

```
ghci> random (mkStdGen 100)
```

```
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Random a' arising from a use of `random' at <interactive>:1:0-20
    Probable fix: add a type signature that fixes these type variable(s)
```

De quoi ? Ah, oui, la fonction `random` peut retourner une valeur de n'importe quel type membre de la classe `Random`, on doit donc informer Haskell du type qu'on désire. N'oublions tout de même pas qu'elle retourne une valeur aléatoire et un générateur aléatoire sous forme de paire.

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Enfin ! Un nombre qui a l'air un peu aléatoire ! La première composante du tuple est notre nombre, alors que la seconde est la représentation textuelle du nouveau générateur. Que se passe-t-il si l'on appelle `random` à nouveau, avec le même générateur ?

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Bien sûr. Le même résultat pour les mêmes paramètres. Essayons de donner un générateur aléatoire différent comme paramètre.

```
ghci> random (mkStdGen 949494) :: (Int, StdGen)
(539963926,466647808 1655838864)
```

Bien, cool, super, un nombre différent. On peut utiliser les annotations de type pour obtenir différents types de cette fonction.

```
ghci> random (mkStdGen 949488) :: (Float, StdGen)
(0.8938442,1597344447 1655838864)
ghci> random (mkStdGen 949488) :: (Bool, StdGen)
(False,1485632275 40692)
ghci> random (mkStdGen 949488) :: (Integer, StdGen)
(1691547873,1597344447 1655838864)
```

Créons une fonction qui simule trois lancers de pièce. Si `random` ne retournait pas un nouveau générateur avec la valeur aléatoire, on devrait donner à cette fonction trois générateurs aléatoires en paramètre, et retourner le jet de pièce pour chacun des trois. Mais ça semble mauvais, parce que si un générateur peut retourner une valeur aléatoire de type `Int` (qui peut prendre énormément de valeurs), il devrait pouvoir renvoyer un jet de trois pièces (qui ne peut prendre que huit valeurs précisément). C'est ici que le fait que `random` retourne un nouveau générateur s'avère très utile.

On va représenter une pièce avec un simple `Bool`. `True` pour pile, `False` pour face.

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in  (firstCoin, secondCoin, thirdCoin)
```

On appelle `random` avec le générateur passé en paramètre pour obtenir un jet et un nouveau générateur. On l'appelle à nouveau avec le nouveau générateur, pour obtenir un deuxième jet. On fait de même pour le troisième. Si on avait appelé à chaque fois avec le même générateur, toutes les pièces auraient eu la même valeur, et on aurait seulement pu avoir `(False, False, False)` ou `(True, True, True)` comme résultat.

```
ghci> threeCoins (mkStdGen 21)
(True,True,True)
ghci> threeCoins (mkStdGen 22)
(True,False,True)
ghci> threeCoins (mkStdGen 943)
(True,False,True)
ghci> threeCoins (mkStdGen 944)
(True,True,True)
```

Remarquez que l'on n'a pas eu à faire `random gen :: (Bool, StdGen)`. C'est parce que nous avions déjà spécifié que l'on voulait des booléens dans la déclaration de type de la fonction. C'est pourquoi Haskell peut inférer qu'on veut une valeur booléenne dans ce cas.

Et si l'on veut lancer quatre pièces ? Ou cinq ? Eh bien, il y a une fonction `randoms` qui prend un générateur, et retourne une liste infinie de valeurs basées sur ce générateur.

```
ghci> take 5 $ randoms (mkStdGen 11) :: [Int]
[-1807975507,545074951,-1015194702,-1622477312,-502893664]
ghci> take 5 $ randoms (mkStdGen 11) :: [Bool]
[True,True,True,True,False]
ghci> take 5 $ randoms (mkStdGen 11) :: [Float]
[7.904789e-2,0.62691015,0.26363158,0.12223756,0.38291094]
```

Pourquoi `randoms` ne retourne pas de nouveau générateur avec la liste ? On peut implémenter `randoms` très facilement ainsi :

```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen in value:randoms' newGen
```

Une définition récursive. On obtient une valeur aléatoire et un nouveau générateur à partir du générateur courant, et on crée une liste qui contient la valeur aléatoire en tête, et une liste de nombres aléatoires obtenue par le nouveau générateur en queue. Puisqu'on doit potentiellement générer une liste infinie de nombres, on ne peut pas renvoyer le nouveau générateur.

On pourrait faire une fonction qui génère un flot fini de nombres aléatoires et un nouveau générateur ainsi :

```
finiteRandoms :: (RandomGen g, Random a, Num n) => n -> g -> ([a], g)
finiteRandoms 0 gen = ([], gen)
```

```haskell
finiteRandoms n gen =
    let (value, newGen) = random gen
        (restOfList, finalGen) = finiteRandoms (n-1) newGen
    in  (value:restOfList, finalGen)
```

Encore, une définition récursive. On dit que si l'on ne veut aucun nombre, on retourne la liste vide et le générateur qu'on nous a donné. Pour tout autre nombre de valeurs aléatoires, on récupère une première valeur aléatoire et un nouveau générateur. Ce sera la tête. Et on dit que la queue sera composée de *n - 1* nombres générés à partir du nouveau générateur. Enfin, on renvoie la tête jointe à la queue, ainsi que le générateur obtenu par l'appel récursif.

Et si l'on voulait une valeur aléatoire dans un certain intervalle ? Tous les entiers aléatoires rencontrés jusqu'à présent étaient outrageusement grands ou petits. Si l'on voulait lancer un dé ? On utilise `randomR` à cet effet. Elle a pour type `randomR :: (RandomGen g, Random a) => (a, a) -> g -> (a, g)`, signifiant qu'elle est comme `random`, mais prend en premier paramètre une paire de valeurs qui définissent une borne inférieure et une borne supérieure pour la valeur produite.

```haskell
ghci> randomR (1,6) (mkStdGen 359353)
(6,1494289578 40692)
ghci> randomR (1,6) (mkStdGen 35935335)
(3,1250031057 40692)
```

Il y a aussi `randomRs`, qui produit un flot de valeurs aléatoires dans l'intervalle spécifié. Regardez ça :

```haskell
ghci> take 10 $ randomRs ('a','z') (mkStdGen 3) :: [Char]
"ndkxbvmomg"
```

Super, on dirait un mot de passe ultra secret ou quelque chose comme ça.

Vous vous demandez peut-être ce que cette section vient faire dans le chapitre sur les entrées-sorties ? On n'a pas fait d'I/O jusqu'ici. Eh bien, jusqu'ici, on a créé notre générateur manuellement avec un entier arbitraire. Le problème si l'on fait ça dans nos vrais programmes, c'est qu'ils retourneront toujours les mêmes suites de nombres aléatoires, ce qui ne nous convient pas. C'est pourquoi `System.Random` offre l'action I/O `getStdGen`, qui a pour type `IO StdGen`. Lorsque votre programme débute, il demande au système un bon générateur aléatoire et le stocke comme un générateur global. `getStdGen` vous récupère ce générateur lorsque vous le liez à un nom.

Voici un simple programme qui génère une chaîne aléatoire.

```haskell
import System.Random

main = do
    gen <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen)
```

```
$ runhaskell random_string.hs
pybphhzzhuepknbykxhe
$ runhaskell random_string.hs
eiqgcxykivpudlsvvjpg
$ runhaskell random_string.hs
nzdceoconysdgcyqjruo
$ runhaskell random_string.hs
bakzhnnuzrkgvesqplrx
```

Attention cependant, faire `getStdGen` deux fois vous donnera le même générateur global deux fois. Donc si vous faites :

```haskell
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen2 <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen2)
```

vous obtiendrez la même chaîne deux fois ! Un moyen d'obtenir deux chaînes de longueur 20 est de mettre en place un flot infini de caractères, de prendre les 20 premiers, les afficher sur une ligne, puis prendre les 20 suivants et les afficher sur une seconde ligne. Pour cela, on peut utiliser `splitAt` de `Data.List`, qui coupe une liste à un indice donné et retourne le tuple formé par la partie coupée en première composante, et le reste en seconde composante.

```haskell
import System.Random
import Data.List

main = do
```

```
    gen <- getStdGen
    let randomChars = randomRs ('a','z') gen
        (first20, rest) = splitAt 20 randomChars
        (second20, _) = splitAt 20 rest
    putStrLn first20
    putStr second20
```

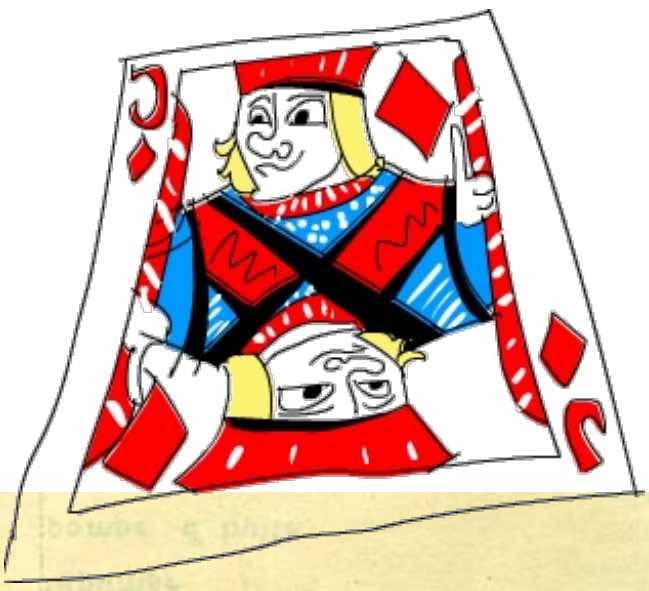
Un autre moyen est d'utiliser l'action **`newStdGen`**, qui coupe notre générateur courant en deux générateurs. Elle remplace le générateur global par l'un deux, et encapsule l'autre comme son résultat.

```
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen' <- newStdGen
    putStr $ take 20 (randomRs ('a','z') gen')
```

Non seulement on obtient un nouveau générateur quand on lie **`newStdGen`** à un nom, mais en plus le générateur global est changé, donc si on fait **`getStdGen`** à nouveau et qu'on le lie à un nom, on obtiendra un générateur différent de **`gen`**.

Voici un petit programme qui fait deviner à l'utilisateur le numéro auquel il pense.

```
import System.Random
import Control.Monad(when)

main = do
    gen <- getStdGen
    askForNumber gen

askForNumber :: StdGen -> IO ()
askForNumber gen = do
    let (randNumber, newGen) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
        askForNumber newGen
```



On crée une fonction **`askForNumber`**, qui prend un générateur aléatoire et retourne une action I/O qui demande un nombre à l'utilisateur et lui dit s'il a bien deviné. Dans cette fonction, on génère d'abord un nombre aléatoire et un nouveau générateur basés sur le générateur obtenu en paramètre, et on les nomme respectivement **`randNumber`** et **`newGen`**. Disons que le nombre généré était **`7`**. On demande ensuite à l'utilisateur de deviner à quel nombre on pense. On fait **`getLine`** et lie le résultat à **`numberString`**. Quand l'utilisateur tape **`7`**, **`numberString`** devient **`"7"`**. Ensuite, on utilise **`when`** pour vérifier si la chaîne entrée par l'utilisateur est vide. Si elle l'est, une action I/O vide **`return ()`** est exécutée, qui termine le programme. Sinon, l'action combinée de ce bloc *do* est effectuée. On utilise **`read`** sur **`numberString`** pour la convertir en nombre, donc **`number`** est **`7`**.

Excusez-moi ! Si l'utilisateur nous donne ici une entrée que **`read`** ne peut pas lire (comme **`"haha"`**), notre programme va planter avec un message d'erreur horrible. Si vous ne voulez pas que votre programme plante sur une entrée erronée, utilisez **`reads`**, qui retourne une liste vide quand elle n'arrive pas à lire la chaîne. Quand elle y parvient, elle retourne une liste singleton avec notre valeur en première composante, et le reste de la chaîne qu'elle n'a pas consommé dans l'autre composante.

On vérifie si le nombre qu'on a entré est égal à celui généré aléatoirement et on donne à l'utilisateur le message approprié. Puis on appelle **`askForNumber`** récursivement, seulement avec le nouveau générateur qu'on a obtenu, ce qui nous donne une nouvelle action I/O qui sera comme celle qu'on vient d'exécuter, mais dépendra d'un générateur différent.

`main` consiste en la récupération d'un générateur aléatoire du système, suivie d'un appel à **`askForNumber`** avec celui-ci pour obtenir l'action initiale.

Voici notre programme en action !

```
$ runhaskell guess_the_number.hs
```

```
Which number in the range from 1 to 10 am I thinking of? 4
Sorry, it was 3
Which number in the range from 1 to 10 am I thinking of? 10
You are correct!
Which number in the range from 1 to 10 am I thinking of? 2
Sorry, it was 4
Which number in the range from 1 to 10 am I thinking of? 5
Sorry, it was 10
Which number in the range from 1 to 10 am I thinking of?
```

Une autre manière d'écrire le même programme est comme suit :

```
import System.Random
import Control.Monad(when)

main = do
    gen <- getStdGen
    let (randNumber, _) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
        newStdGen
        main
```

C'est très similaire à la version précédente, mais plutôt que de faire une fonction qui prend un générateur et s'appelle récursivement avec un nouveau générateur, on fait tout le travail dans `main`. Après avoir dit à l'utilisateur s'il a correctement deviné ou pas, on met à jour le générateur global et on appelle `main` à nouveau. Les deux approches sont valides, mais je préfère la première parce qu'elle fait moins de choses dans `main` et nous offre une fonction facilement réutilisable.

Chaînes d'octets

Les listes sont des structures de données cool et utiles. Jusqu'ici, on les a utilisées à peu près partout. Il y a une multitude de fonctions opérant sur elles et la paresse d'Haskell nous permet d'échanger les boucles *for* et *while* des autres langages contre des filtrages et des mappages sur des listes, parce que l'évaluation n'aura lieu que lorsque cela sera nécessaire, donc des choses comme des listes infinies (et même des listes infinies de listes infinies !) ne nous posent pas de problème. C'est pourquoi les listes peuvent être utilisées pour réprésenter les flots, que ce soit en lecture depuis l'entrée standard ou un fichier. On peut juste ouvrir un fichier, et le lire comme une chaîne de caractères, alors qu'en réalité il ne sera accédé que quand ce sera nécessaire.

Cependant, traiter les fichiers comme des listes a un inconvénient : ça a tendance à être assez lent. Comme vous le savez, `String` est un synonyme de type pour `[Char]`. Les `Char` n'ont pas une taille fixe, parce qu'il faut plusieurs octets pour représenter un caractère, par exemple Unicode. De plus, les listes sont vraiment paresseuses. Si vous avez une liste comme `[1, 2, 3, 4]`, elle ne sera évaluée que lorsque ce sera vraiment nécessaire. La liste entière est une promesse de liste. Rappelez-vous que `[1, 2, 3, 4]` est un sucre syntaxique pour `1:2:3:4:[]`. Lorsque le premier élément de la liste est forcé à être évalué (par exemple, en l'affichant), le reste de la liste `2:3:4:[]` est toujours une promesse de liste, et ainsi de suite. Vous pouvez donc imaginer les listes comme des promesses que l'élément suivant sera délivré quand on en aura besoin, ainsi que la promesse que la suite fera pareil. Il ne faut pas se creuser l'esprit bien longtemps pour se dire que traiter une simple liste de nombres comme une série de promesses n'est peut-être pas la manière la plus efficace au monde.

Ce coût supplémentaire ne nous dérange pas la plupart du temps, mais il s'avère handicapant lorsque l'on lit et manipule des gros fichiers. C'est pourquoi Haskell a des **chaînes d'octets**. Les chaînes d'octets sont un peu comme des listes, seulement chaque élément fait un octet (ou 8 bits) de taille. La manière dont elles sont paresseuses est aussi différente.

Les chaînes d'octets viennent sous deux déclinaisons : les strictes et les paresseuses. Les chaînes d'octets strictes résident dans `Data.ByteString` et elles abandonnent complètement la paresse. Plus de promesses impliquées, une chaîne d'octets stricte est une série d'octets dans un tableau. Vous ne pouvez pas avoir de chaîne d'octets stricte infinie. Si vous évaluez le premier octet d'une chaîne stricte, elle est évaluée en entier. Le bon côté des choses, c'est qu'il y a moins de coût supplémentaire puisqu'il n'y a plus de glaçons (le terme technique pour les promesses) impliqués. Le mauvais côté, c'est qu'elles risquent plus de remplir votre mémoire, parce qu'elles sont lues entièrement dans la mémoire d'un coup.

L'autre variété de chaîne d'octets réside dans `Data.ByteString.Lazy`. Elles sont paresseuses, mais pas autant que les listes. Comme on l'a dit plus tôt, il y a autant de glaçons dans une liste que d'éléments dans la liste. C'est ce qui les rend un peu lentes pour certaines opérations. Les chaînes d'octets paresseuses prennent une approche différente - elles sont stockées dans des morceaux (à ne pas confondre avec des glaçons !), chaque morceau ayant une taille de 64K.

Donc, lorsque vous évaluez un octet d'une chaîne d'octets paresseuse (en l'affichant ou autre), les premiers 64K sont évalués. Après cela, c'est juste une promesse pour le reste des morceaux. Les chaînes d'octets sont un peu comme des listes de chaînes d'octets strictes de taille 64K. Quand vous traitez un fichier avec des chaînes d'octets paresseuses, il sera lu morceau par morceau. C'est cool parce que cela ne causera pas une montée en flèche de l'utilisation mémoire et les 64K tiennent probablement correctement dans le cache L2 de votre CPU.

Si vous lisez la documentation de `Data.ByteString.Lazy`, vous verrez qu'il a beaucoup de fonctions qui ont les mêmes noms que celles de `Data.List`, seulement les signatures de type ont `ByteString` au lieu de `[a]` et `Word8` au lieu de `a`. Les fonctions ayant le même nom fonctionnent principalement identiquement à celles sur les listes. Puisque les noms sont identiques, on va faire un import qualifié dans un script et charger ce script dans GHCi pour jouer avec les chaînes d'octets.

```
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
```

`B` contient les chaînes d'octets paresseuses, alors que `S` contient les strictes. On utilisera principalement les paresseuses.

La fonction `pack` a pour signature de type `pack :: [Word8] -> ByteString`. Cela signifie qu'elle prend une liste d'octets de type `Word8` et retourne une `ByteString`. Vous pouvez l'imaginer comme prenant une liste, qui est paresseuse, et la rendant moins paresseuse, c'est-à-dire seulement paresseuse à chaque intervalle de 64K.

C'est quoi ce type `Word8` au fait ? Eh bien, c'est comme `Int`, mais avec un plus petit intervalle de valeurs, c'est à dire 0 à 255. Cela représente un nombre sur 8 bits. Tout comme `Int`, il est dans la classe de types `Num`. Par exemple, on sait que la valeur `5` est polymorphe et peut se faire passer pour n'importe quel type numérique. Eh bien elle peut avoir pour type `Word8`.

```
ghci> B.pack [99,97,110]
Chunk "can" Empty
ghci> B.pack [98..120]
Chunk "bcdefghijklmnopqrstuvwx" Empty
```

Comme vous le voyez, vous n'avez généralement pas trop à vous soucier de `Word8`, parce que le système de types peut faire choisir aux nombres ce type. Si vous essayez un nombre trop gros, comme `336`, il sera juste modulé à `80`.

On a juste placé une poignée de valeurs dans une `ByteString`, donc elles tenaient dans un seul morceau. `Empty` est comme `[]` des listes.

`unpack` est la fonction inverse de `pack`. Elle prend une chaîne d'octets et la transforme en liste d'octets.

`fromChunks` prend une liste de chaînes d'octets strictes et la convertit en une chaîne d'octets paresseuse. `toChunks` prend une chaîne d'octets paresseuse et la convertit en une liste de chaînes d'octets strictes.

```
ghci> B.fromChunks [S.pack [40,41,42], S.pack [43,44,45], S.pack [46,47,48]]
Chunk "()*" (Chunk "+,-" (Chunk "./0" Empty))
```

C'est bien lorsque vous avez beaucoup de petites chaînes d'octets strictes et que vous voulez les traiter efficacement sans les joindre en une grosse chaîne stricte en mémoire.

La version chaîne d'octets de `:` est appelée `cons`. Elle prend un octet et une chaîne d'octets et place l'octet au début de la chaîne. Ceci est paresseux, donc elle va créer un nouveau morceau, même si le morceau précédent n'était pas rempli. C'est pourquoi il vaut mieux utiliser la version stricte de `cons`, `cons'` si vous voulez insérez plein d'octets au début d'une chaîne d'octets.

```
ghci> B.cons 85 $ B.pack [80,81,82,84]
Chunk "U" (Chunk "PQRT" Empty)
ghci> B.cons' 85 $ B.pack [80,81,82,84]
Chunk "UPQRT" Empty
ghci> foldr B.cons B.empty [50..60]
Chunk "2" (Chunk "3" (Chunk "4" (Chunk "5" (Chunk "6" (Chunk "7" (Chunk "8" (Chunk "9" (Chunk ":" (Chunk ";" (Chunk "<"
Empty))))))))))
ghci> foldr B.cons' B.empty [50..60]
Chunk "23456789:;<" Empty
```

Comme vous pouvez le constater, `empty` crée une chaîne d'octets vide. Vous voyez la différence entre `cons` et `cons'` ? Avec `foldr`, on est parti d'une chaîne d'octets vide, et on a ajouté tous les nombres d'une liste en partant de la droite au début de cette chaîne. Quand on a utilisé `cons`, on s'est retrouvé avec un morceau pour chaque octet, ce qui détruit l'utilité de la chaîne.

Autrement, les modules pour les chaînes d'octets ont une pelletée de fonctions analogues à celles de `Data.List`, incluant, mais non limitées à, `head`, `tail`, `init`, `null`, `length`, `map`, `reverse`, `foldl`, `foldr`, `concat`, `takeWhile`, `filter`, etc.

Ils ont aussi des fonctions qui ont le même nom que certaines des fonctions de `System.IO`, mais avec `ByteString` à la place de `String`. Par exemple, `readFile` de `System.IO` a pour type `readFile :: FilePath -> IO String`, alors que `readFile` des modules de chaînes d'octets a pour type `readFile :: FilePath -> IO ByteString`. Attention, si vous utilisez des chaînes d'octets strictes et que vous essayez de lire un fichier, il sera lu en entier en mémoire d'un coup ! Avec des chaînes d'octets paresseuses, il sera lu par morceaux.

Créons un simple programme qui prend deux noms de fichiers en arguments et copie le premier fichier dans le second. Notez que `System.Directory` a déjà une fonction `copyFile`, mais on va implémenter notre propre fonction de copie pour ce programme de toute façon.

```
import System.Environment
import qualified Data.ByteString.Lazy as B

main = do
    (fileName1:fileName2:_) <- getArgs
    copyFile fileName1 fileName2

copyFile :: FilePath -> FilePath -> IO ()
copyFile source dest = do
    contents <- B.readFile source
    B.writeFile dest contents
```

On crée notre propre fonction qui prend deux `FilePath` (rappelez-vous, `FilePath` est un synonyme de `String`) et retourne une action I/O qui copie un fichier sur l'autre en utilisant des chaînes d'octets. Dans la fonction `main`, on récupère seulement les arguments et on appelle notre fonction avec ceux-ci pour obtenir l'action I/O, qu'on exécute ensuite.

```
$ runhaskell bytestringcopy.hs something.txt ../../something.txt
```

Remarquez qu'un programme qui n'utiliserait pas de chaîne d'octets paresseuses ressemblerait exactement à celui-ci, à part le fait qu'on ait utilisé `B.readFile` et `B.writeFile` au lieu de `readFile` et `writeFile`. Beaucoup de fois, vous pouvez convertir un programme qui utilise des chaînes de caractères normales en un programme qui utilise des chaînes d'octets en faisant les imports nécessaires et en qualifiant les bonnes fonctions par le nom des modules appropriés. Parfois, vous devez tout de même convertir certaines fonctions que vous avez écrites pour qu'elle fonctionne sur des chaînes d'octets, mais c'est assez facile.

Chaque fois que vous avez besoin de meilleures performances dans un programme qui lit beaucoup de données, essayez les chaînes d'octets, il se peut que vous obteniez de bons gains de performance sans trop d'efforts de votre part. J'écris généralement mes programmes avec des chaînes de caractères, et je les convertis en chaînes d'octets si les performances sont insatisfaisantes.

Exceptions

Tous les langages ont des procédures, des fonctions et des bouts de code qui peuvent échouer d'une certaine façon. C'est la vie. Différents langages ont différentes manières de gérer ces erreurs. En C, on utilise généralement une valeur anormale (comme `-1` ou un pointeur nul) pour indiquer que ce que la fonction a retourné ne devrait pas être traité comme une valeur normale. Java et C#, d'un autre côté, tendent à utiliser les exceptions pour gérer les échecs. Quand une exception est levée, le flot de contrôle saute jusqu'à un code qu'on a défini pour nettoyer un peu et possiblement lever une autre exception de manière à ce qu'un autre gestionnaire d'exceptions s'occupe d'autre chose.

Haskell a un très bon système de types. Les types de données algébriques permettent d'utiliser des types comme `Maybe` ou `Either` et des valeurs de ces types pour représenter des choses qui peuvent être présentes ou non. En C, retourner, disons, `-1` en cas d'échec est un problème de convention. Cette valeur n'est spéciale que pour nous humains, et si l'on ne fait pas attention, on pourrait la traiter par erreur comme une valeur normale, et provoquer le chaos et le désarroi dans notre code. Le système de types d'Haskell nous offre une sûreté bien nécessaire sur cet aspect. Une fonction qui a pour type `a -> Maybe b` indique clairement qu'elle peut produire un `b` enveloppé dans un `Just` ou retourner `Nothing`. Le type est différent de `a -> b`, et si on essaie de remplacer l'une par l'autre, le compilateur se plaindra.

Bien qu'il ait un système de types expressif qui supporte les échecs de calculs, Haskell supporte quand même les exceptions, parce qu'elles sont plus sensées dans un contexte d'I/O. Beaucoup de choses peuvent mal tourner quand on traite avec le monde extérieur, car il est très imprévisible. Par exemple, quand on ouvre un fichier, beaucoup de choses peuvent mal se passer. Le fichier peut être verrouillé, ne pas être là, voire le disque dur lui-même peut ne pas être là. Ainsi, il est pratique de sauter à une partie du code qui s'occupe de gérer cela quand une telle erreur a lieu.

Ok, donc le code I/O (i.e. le code impur) peut lever des exceptions. C'est sensé. Mais qu'en est-il du code pur ? Eh bien, il peut aussi lever des exceptions. Pensez à `div` ou `head`. Elles ont respectivement pour type `(Integral a) => a -> a -> a` et `[a] -> a`. Pas de `Maybe` ou d'`Either` dans leur type de retour, et pourtant elles peuvent toute deux échouer ! `div` vous explose au visage lorsque vous essayez de diviser par zéro et `head` pique une crise de colère lorsqu'on lui donne une liste vide.

```
ghci> 4 `div` 0
*** Exception: divide by zero
ghci> head []
*** Exception: Prelude.head: empty list
```

Du code pur peut lancer des exceptions, mais elles ne peuvent être attrapées que dans du code impur (dans un bloc *do* sous `main`). C'est parce que l'on ne sait jamais quand (ou si) quelque chose sera évalué dans du code pur, puisqu'il est paresseux et n'a pas d'ordre d'exécution spécifié, contrairement au code d'entrée-sortie.

Plus tôt, on a parlé de passer le moins de temps possible de notre programme dans les entrées-sorties. La logique de notre programme doit résider principalement dans nos fonctions pures, parce que leur résultat ne dépend que des paramètres avec lesquelles elles sont appelées. Quand vous manipulez des fonctions pures, vous n'avez qu'à penser à ce qu'elles retournent, parce qu'elles ne peuvent rien faire d'autre. Bien qu'un peu de logique dans les I/O soit nécessaire (pour ouvrir des fichiers par exemple), elle devrait préférablement être restreinte au minimum. Les fonctions pures sont paresseuses par défaut, ce qui veut dire qu'on ne sait pas quand elles seront évaluées et que cela ne doit pas importer. Cependant, dès que des fonctions pures se mettent à lancer des exceptions, leur ordre d'évaluation devient important. C'est pourquoi on ne peut attraper ces exceptions que dans la partie impure de notre code. Et c'est mauvais, parce que l'on veut garder cette partie aussi petite que possible. Cependant, si on n'attrape pas ces erreurs, notre programme plante. La solution ? Ne pas mélanger les exceptions et le code pur. Tirez profit du puissant système de types d'Haskell et utilisez des types comme `Either` et `Maybe` pour représenter des résultats pouvant échouer.

C'est pourquoi nous n'allons regarder que les exceptions d'I/O pour l'instant. Les exceptions d'I/O sont des exceptions causées par quelque chose se passant mal lorsqu'on communique avec le monde extérieur dans une action I/O qui fait partie de notre `main`. Par exemple, on peut tenter d'ouvrir un fichier, puit s'apercevoir qu'il a été supprimé, ou quelque chose comme ça. Regardez ce programme qui ouvre un fichier dont le nom lui est donné en argument et nous dit combien de lignes le fichier contient.

```
import System.Environment
import System.IO

main = do (fileName:_) <- getArgs
          contents <- readFile fileName
          putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"
```

Un programme très simple. On effectue l'action I/O `getArgs` et lie la première chaîne de la liste retournée à `fileName`. Puis on appelle `contents` le contenu du fichier qui porte ce nom. Finalement, on applique `lines` sur ce contenu pour obtenir une liste de lignes, et on récupère la longueur de cette liste et on la donne à `show` pour obtenir une représentation textuelle de ce nombre. Cela fonctionne comme prévu, mais que se passe-t-il lorsqu'on donne le nom d'un fichier inexistant ?

```
$ runhaskell linecount.hs i_dont_exist.txt
linecount.hs: i_dont_exist.txt: openFile: does not exist (No such file or directory)
```

Aha, on obtient une erreur de GHC, nous disant que le fichier n'existe pas. Notre programme plante. Et si l'on voulait afficher un message plus joli quand le fichier n'existe pas ? Un moyen de faire cela est de vérifier l'existence du fichier à l'aide de `doesFileExist` de `System.Directory`.

```
import System.Environment
import System.IO
import System.Directory

main = do (fileName:_) <- getArgs
          fileExists <- doesFileExist fileName
          if fileExists
              then do contents <- readFile fileName
                      putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"
              else do putStrLn "The file doesn't exist!"
```

On a fait `fileExists <- doesFileExist fileName` parce que `doesFileExist` a pour type `doesFileExist :: FilePath -> IO Bool`, ce qui signifie qu'elle retourne une action I/O qui a pour résultat une valeur booléenne nous indiquant si le fichier existe. On ne peut pas utiliser simplement `doesFileExist` dans une expression *if* directement.

Une autre solution ici serait d'utiliser des exceptions. C'est parfaitement acceptable dans ce contexte. Un fichier inexistant est une exception qui est levée par une I/O, donc l'attraper dans une I/O est propre et correct.

Pour gérer ceci en utilisant des exceptions, on va tirer parti de la fonction `catch` de `System.IO.Error`. Son type est `catch :: IO a -> (IOError -> IO a) -> IO a`. Elle prend deux paramètres. Le premier est une action I/O. Par exemple, ça pourrait être une action I/O qui essaie d'ouvrir un fichier. Le second est le gestionnaire d'exceptions. Si l'action I/O passée en premier paramètre à `catch` lève une exception I/O, cette exception

sera passée au gestionnaire, qui décidera alors quoi faire. Le résultat final est une action I/O qui se comportera soit comme son premier paramètre, ou bien exécutera le gestionnaire en fonction de l'exception levée par la première action I/O.

Si vous êtes familier avec les blocs *try-catch* de langages comme Java ou Python, la fonction `catch` est similaire. Le premier paramètre est la chose à essayer, un peu comme ce qu'on met dans le bloc *try* dans d'autres langages impératifs. Le second paramètre est le gestionnaire d'exceptions, un peu comme la plupart des blocs *catch* qui reçoivent des exceptions que vous pouvez examiner pour savoir ce qui s'est mal passé. Le gestionnaire est invoqué lorsqu'une exception est levée.

Le gestionnaire prend une valeur de type `IOError`, qui est une valeur signifiant que l'exception qui a eu lieu était liée à une I/O. Elle contient aussi des informations sur le type de l'exception levée. La façon dont ce type est implémenté dépend de l'implémentation du langage, donc on ne peut pas inspecter les valeurs de type `IOError` par filtrage par motif, tout comme on ne peut pas filtrer par motif les valeurs de type `IO something`. On peut tout de même utiliser tout un tas de prédicats utiles pour savoir des choses à propos de valeurs de type `IOError` comme nous le verrons dans une seconde.

Mettons notre nouvelle amie `catch` à l'essai !

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e = putStrLn "Whoops, had some trouble!"
```

Tout d'abord, vous verrez qu'on a mis des apostrophes renversées autour de `catch` pour l'utiliser de manière infixe, parce qu'elle prend deux paramètres. L'utiliser de manière infixe la rend plus lisible. Donc, `toTry `catch` handler` est la même chose que `catch toTry handler`, qui correspond bien à son type. `toTry` est une action I/O qu'on essaie d'exécuter, et `handler` est la fonction qui prend une `IOError` et retourne une action à exécuter en cas d'exception.

Essayons :

```
$ runhaskell count_lines.hs i_exist.txt
The file has 3 lines!

$ runhaskell count_lines.hs i_dont_exist.txt
Whoops, had some trouble!
```

Dans le gestionnaire, nous n'avons pas vérifié de quel type d'`IOError` il s'agissait. On a juste renvoyé `"Whoops, had some trouble!"` quelque que soit le type d'erreur. Attraper tous les types d'erreur dans un seul gestionnaire est une mauvaise pratique en Haskell tout comme dans la plupart des autres langages. Et si une exception arrivait que l'on ne désirait pas attraper, comme une interruption du programme par l'utilisateur ? C'est pour cela qu'on va faire comme dans la plupart des autres langages : on va vérifier de quel type d'exception il s'agit. Si c'est celui qu'on attendait, on la traite. Sinon, on la lève à nouveau dans la nature. Modifions notre programme pour n'attraper que les exceptions liées à l'inexistence du fichier.

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e = putStrLn "The file doesn't exist!"
    | otherwise = ioError e
```

Tout reste pareil à part le gestionnaire, qu'on a modifié pour n'attraper qu'un certain groupe d'exceptions I/O. Ici, on a utilisé deux nouvelles fonctions de `System.IO.Error` - `isDoesNotExistError` et `ioError`. `isDoesNotExistError` est un prédicat sur les `IOError`, ce qui signifie que c'est une fonction

prenant une `IOError` et retournant `True` ou `False`, ayant donc pour type `isDoesNotExistError :: IOError -> Bool`. On utilise cette fonction sur l'exception que notre gestionnaire reçoit pour savoir si c'est une erreur causée par l'inexistence du fichier. On utilise la syntaxe des gardes ici, mais on aurait aussi pu utiliser un *if else*. Si ce n'était pas causé par un fichier inexistant, on lève à nouveau l'exception passée au gestionnaire à l'aide de la fonction `ioError`. Elle a pour type `ioError :: IOError -> IO a`, donc elle prend une `IOError` et produit une action I/O qui va lever cette exception. L'action I/O a pour type `IO a`, parce qu'elle ne retourne jamais de résultat, donc elle peut se faire passer pour une `IO anything`.

Donc, si l'exception levée dans l'action I/O `toTry` qu'on a collée avec un bloc *do* n'est pas causée par l'inexistence du fichier, `toTry `catch` handler` va attraper cette exception et la lever à nouveau. Plutôt cool, hein ?

Il y a plusieurs prédicats qui agissent sur des `IOError`, et lorsqu'une garde n'est pas évaluée comme `True`, l'évaluation passe à la prochaine garde. Les prédicats sur les `IOError` sont :

- `isAlreadyExistsError`
- `isDoesNotExistError`
- `isAlreadyInUseError`
- `isFullError`
- `isEOFError`
- `isIllegalOperation`
- `isPermissionError`
- `isUserError`

La plupart d'entre eux sont évidents. `isUserError` s'évalue à `True` quand on utilise la fonction `userError` pour lever l'exception, qui sert à utiliser nos propres exceptions en les accompagnant d'une chaîne de caractères. Par exemple, vous pouvez faire `ioError $ userError "remote computer unplugged!"`, bien qu'il soit préférable d'utiliser des types comme `Either` et `Maybe` pour exprimer des échecs plutôt que de lancer vous-même des exceptions avec `userError`.

Vous pourriez ainsi avoir un gestionnaire de la sorte :

```
handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e = putStrLn "The file doesn't exist!"
    | isFullError e = freeSomeSpace
    | isIllegalOperation e = notifyCops
    | otherwise = ioError e
```

Où `notifyCops` et `freeSomeSpace` sont des actions I/O que vous définissez. Soyez certain de lever à nouveau les exceptions si elles ne correspondent pas à vos critères, sinon votre programme échouera silencieusement là où il ne devrait pas.

`System.IO.Error` exporte aussi des fonctions qui nous premettent de demander à nos exceptions certains de leurs attributs, comme la poignée de fichier qui a causé l'erreur, ou le nom du fichier. Elles commencent par `ioe` et vous pouvez trouver la liste complète dans la documentation. Mettons qu'on veuille afficher le nom du fichier responsable de l'erreur. On ne peut pas afficher le `fileName` qu'on a reçu de `getArgs`, parce que seule l'`IOError` est passée au gestionnaire, et ce dernier ne connaît rien d'autre. Une fonction ne dépend que des paramètres avec lesquels elle a été appelée. C'est pourquoi on peut utiliser la fonction `ioeGetFileName`, qui a pour type `ioeGetFileName :: IOError -> Maybe FilePath`. Elle prend une `IOError` en paramètre et retourne éventuellement un `FilePath` (qui est un synonyme de `String`, souvenez-vous en). En gros, elle extrait le chemin du fichier de l'`IOError`, si elle le peut. Modifions notre programme pour afficher le chemin du fichier responsable de l'exception.

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e =
        case ioeGetFileName e of Just path -> putStrLn $ "Whoops! File does not exist at: " ++ path
                                 Nothing -> putStrLn "Whoops! File does not exist at unknown location!"
    | otherwise = ioError e
```

Dans la garde où `isDoesNotExistError` est `True`, on a utilisé une expression *case* pour appeler `ioeGetFileName` avec `e`, puis on a filtré par motif contre la valeur `Maybe` retournée. Utiliser une expression *case* se fait généralement pour filtrer par motif sur quelque chose sans introduire une nouvelle fonction.

Vous n'avez pas à utiliser un unique gestionnaire pour attraper (i.e. `catch`) toutes les exceptions de la partie I/O de votre code. Vous pouvez simplement protéger certaines parties de votre code avec `catch` ou vous pouvez couvrir plusieurs parties avec `catch` et utiliser différents gestionnaires pour chacune, ainsi :

```
main = do toTry `catch` handler1
          thenTryThis `catch` handler2
          launchRockets
```

Ici, `toTry` utilise le gestionnaire `handler1` et `thenTryThis` utilise le gestionnaire `handler2`. `launchRockets` n'est pas un paramètre de `catch`, donc toute exception qu'elle pourra lancer plantera probablement votre programme, à moins que `launchRockets` n'utilise un `catch` en interne pour gérer ses propres exceptions. Bien sûr, `toTry`, `thenTryThis` et `launchRockets` sont des actions I/O qui ont été collées ensemble avec la notation *do* et, hypothétiquement définies quelque part ailleurs. C'est un peu similaire aux blocs *try-catch* des autres langages, où l'on peut entourer notre programme entier d'un simple *try-catch*, ou bien utiliser une approche plus fine et utiliser différents *try-catch* sur différentes parties du code pour contrôler le type d'erreur qui peut avoir lieu à chaque endroit.

Maintenant, vous savez gérer les exceptions I/O ! Lever des exceptions depuis un code pur n'a pas encore été couvert, principalement parce que, comme je l'ai déjà dit, Haskell offre de bien meilleurs façons d'indiquer des erreurs, plutôt que s'en remettre aux I/O pour les attraper. Même en collant des actions I/O les unes aux autres, je préfère que leur type soit `IO (Either a b)`, c'est-à-dire des actions I/O normales mais dont le résultat peut être `Left a` ou `Right b` lorsqu'elles sont exécutées.

← Créer nos propres types et classes de types Table des matières Résoudre des problèmes fonctionnellement →

Résoudre des problèmes fonctionnellement

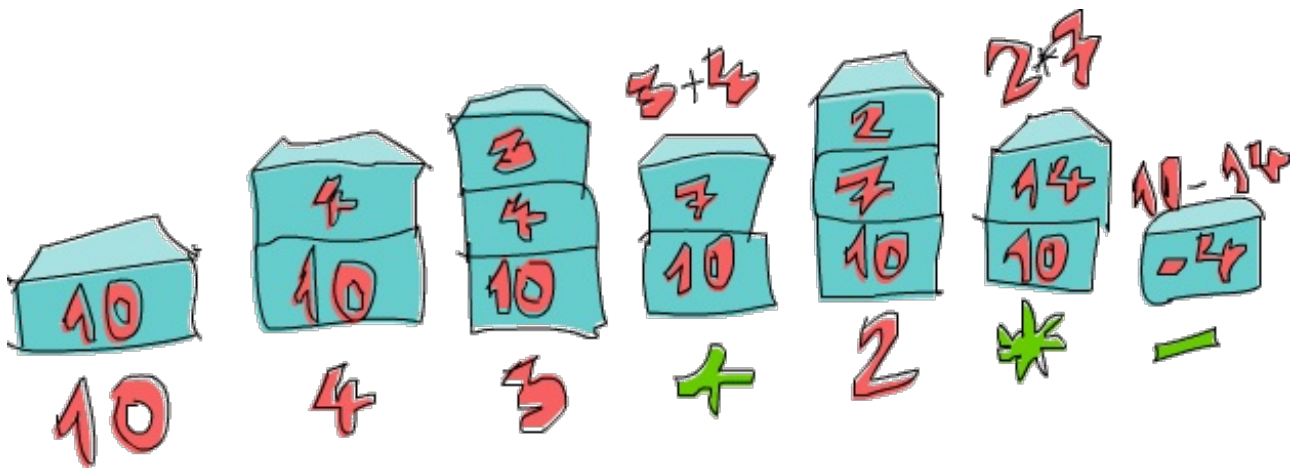
← Entrées et sorties | Table des matières | Foncteurs, foncteurs applicatifs et monoïdes →

Dans ce chapitre, nous allons regarder quelques problèmes intéressants et comment les résoudre fonctionnellement et aussi élégamment que possible. On ne découvrira probablement pas de nouveau concept, on va juste échauffer nos muscles Haskell tout fraîchement acquis et s'entraîner à coder. Chaque section présentera un problème différent. On commencera par décrire le problème, puis on essaiera de trouver le meilleur moyen de le résoudre (ou le moins mauvais).

Calculatrice de notation polonaise inverse

Généralement, lorsqu'on écrit des expressions mathématiques à l'école, on les écrit de manière infixe. Par exemple, on écrit `10 - (4 + 3) * 2`. `+`, `*` et `-` sont des opérateurs infixes, tout comme les fonctions infixes qu'on a rencontrées en Haskell (`+`, `` `elem` ``, etc.). C'est pratique puisqu'en tant qu'humains, il nous est facile de décomposer cette expression dans notre esprit. L'inconvénient, c'est qu'on a besoin de parenthèses pour indiquer la précédence.

La notation polonaise inverse est une autre façon d'écrire les expressions mathématiques. Au départ, ça semble un peu bizarre, mais c'est en fait assez simple à comprendre et à utiliser puisqu'il n'y a pas besoin de parenthèses, et parce que c'est très simple à taper dans une calculatrice. Bien que la plupart des calculatrices modernes utilisent la notation infixe, certaines personnes ne jurent toujours que par leur calculatrice NPI. Voici ce à quoi l'expression infixe précédente ressemble en NPI : `10 4 3 + 2 * -`. Comment calcule-t-on le résultat de cela ? Eh bien, imaginez une pile. Vous parcourez l'expression de gauche à droite. Chaque fois qu'un nombre est rencontré, vous l'empilez. Dès que vous rencontrez un opérateur, vous retirez les deux nombres en sommet de pile (on dit aussi qu'on les *dépile*), utilisez l'opérateur sur ces deux nombres, et empilez le résultat. Si l'expression est bien formée, en arrivant à la fin vous ne devriez plus avoir qu'un nombre dans la pile, et ce nombre est le résultat.

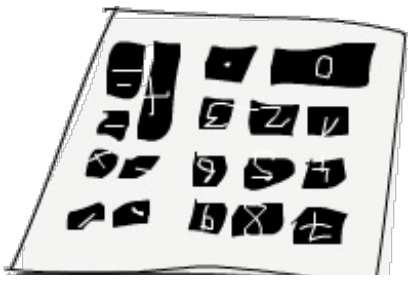


Parcourons l'expression `10 4 3 + 2 * -` ensemble ! D'abord, on empile `10`, et la pile est donc `10`. Le prochain élément est `4`, on l'empile également. La pile est maintenant `10, 4`. De même avec `3`, la pile est à présent `10, 4, 3`. Soudain, on rencontre un opérateur, j'ai nommé `+` ! On dépile les deux nombres en sommet de la pile (donc la pile devient `10`), on somme ces deux nombres, et on empile le résultat. La pile est désormais `10, 7`. On empile le `2`, la pile devient `10, 7, 2`. On rencontre un nouvel opérateur, on dépile donc `7` et `2`, on les multiplie et on empile le résultat. `7` fois `2` donne `14`, la pile est donc `10, 14`. Finalement, il y a un `-`. On dépile `10` et `14`, on soustrait `14` de `10` et on empile le résultat. La pile est maintenant `-4`, et puisqu'il n'y a plus de nombres ou d'opérateurs dans notre expression, c'est notre résultat !

Maintenant qu'on sait calculer n'importe quelle expression NPI à la main, réfléchissons à une manière d'écrire une fonction Haskell qui prendrait en paramètre une chaîne de caractères contenant une expression NPI, comme `"10 4 3 + 2 * -"`, et nous renvoie son résultat.

Quel serait le type d'une telle fonction ? On veut qu'elle prenne une chaîne de caractères en paramètre, et produise un nombre en résultat. Ce sera donc probablement quelque chose comme `solveRPN :: (Num a) => String -> a` (NDT : "RPN" pour "Reverse Polish Notation").

Astuce : cela aide beaucoup de réfléchir d'abord à la déclaration de type d'une fonction avant de s'inquiéter de l'implémentation, et d'écrire cette déclaration. En Haskell, une déclaration de type nous en dit beaucoup sur une fonction, grâce au système de types puissant.



Cool. Quand on implémente une solution à un problème en Haskell, il est aussi bien de penser à la façon dont vous le feriez à la main et d'en tirer des idées. Ici, on voit qu'on traite chaque nombre ou opérateur séparé par un espace comme un élément unique. Il serait donc peut-être utile de commencer par découper une chaîne comme `"10 4 3 + 2 * -"` en une liste d'éléments comme `["10","4","3","+","2","*","-"]`.

Ensuite, que faisions-nous avec cette liste d'éléments dans notre tête ? On la parcourait de la gauche vers la droite, et on

maintenait une pile tout du long. Est-ce que cette phrase vous rappelle quelque chose ? Souvenez-vous, dans la section sur les plis, on a dit que quasiment toute fonction qui traverse une liste de gauche à droite ou de droite à gauche, élément par élément, et construit (accumule) un résultat (que ce soit un nombre, une liste, une pile, peu importe), peut être implémentée comme un pli.

Dans ce cas, on va utiliser un pli gauche, puisqu'on traverse la liste de la gauche vers la droite. La valeur de l'accumulateur sera notre pile et ainsi, le résultat du pli sera aussi une pile, seulement, comme on l'a vu, elle ne contiendra qu'un élément.

Une autre chose à pondérer est, eh bien, comment va-t-on représenter cette pile ? Je propose d'utiliser une liste. Également, je propose de garder la somme de notre pile du côté de la tête de la liste. C'est parce qu'ajouter en tête (début) de liste est bien plus rapide que d'ajouter à la fin. Donc si l'on a une pile qui consiste en, mettons, `10, 4, 3`, nous la représenterons comme la liste `[3, 4, 10]`.

On a à présent assez d'informations pour ébaucher notre fonction. Elle va prendre une liste, comme `"10 4 3 + 2 * -"` et la décomposer en liste d'éléments en utilisant `words` pour obtenir `["10","4","3","+","2","*","-"]`. Ensuite, on va utiliser un pli gauche sur la liste et terminer avec une pile à un seul élément, `[-4]`. On sort cet élément de la liste, et c'est notre résultat final !

Voici donc l'esquisse de notre fonction :

```
import Data.List

solveRPN :: (Num a) => String -> a
solveRPN expression = head (foldl foldingFunction [] (words expression))
    where   foldingFunction stack item = ...
```

On prend l'expression et on la change en une liste d'éléments. Puis on plie la liste d'éléments avec la fonction de pli. Remarquez le `[]`, qui représente l'accumulateur initial. L'accumulateur est notre pile, donc `[]` représente une pile vide, avec laquelle on débute. Une fois qu'on récupère la pile finale qui ne contient qu'un élément, on appelle `head` sur cette liste pour obtenir cet élément, et on applique `read`.

Tout ce qu'il reste à faire consiste à implémenter une fonction de pli qui va prendre une pile, comme `[4, 10]`, et un élément, comme `"3"`, et retourner une nouvelle pile `[3, 4, 10]`. Si la pile est `[4, 10]` et que l'élément est `"*"`, alors elle devra retourner `[40]`. Mais avant cela, transformons notre fonction en style sans point, parce qu'elle est pleine de parenthèses qui m'effraient :

```
import Data.List

solveRPN :: (Num a) => String -> a
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction stack item = ...
```

Ah, nous voilà. Beaucoup mieux. Ainsi, la fonction de pli prend une pile et un élément et retourne une nouvelle pile. On va utiliser du filtrage par motif pour obtenir les deux éléments en haut de pile, et filtrer les opérateurs comme `"*"` et `"-"`.

```
solveRPN :: (Num a, Read a) => String -> a
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction (x:y:ys) "*" = (x * y):ys
            foldingFunction (x:y:ys) "+" = (x + y):ys
            foldingFunction (x:y:ys) "-" = (y - x):ys
            foldingFunction xs numberString = read numberString:xs
```

On a étendu cela sur quatre motifs. Les motifs seront essayés de haut en bas. D'abord, la fonction de pli regarde si l'élément courant est `"*"`. Si c'est le cas, alors elle prendra une liste comme `[3, 4, 9, 3]` et nommera ses deux premiers éléments `x` et `y` respectivement. Dans ce cas, `x` serait `3` et `y` serait `4`. `ys` serait `[9, 3]`. Elle retourne une liste comme `ys`, mais avec le produit de `x` et `y` en tête. Ainsi, on a dépilé les deux nombres en haut de pile, on les a multipliés et on a empilé le résultat. Si l'élément n'est pas `"*"`, le filtrage par motif continue avec le motif `"+"`, et ainsi de suite.

Si l'élément n'est aucun des opérateurs, alors on suppose que c'est une chaîne qui représente un nombre. Si c'est un nombre, on appelle `read` sur la chaîne pour obtenir un nombre, et on retourne la pile précédente avec ce nombre empilé.

Et c'est tout ! Remarquez aussi qu'on a ajouté une contrainte de classe supplémentaire `Read a` à la déclaration de type de la fonction, parce qu'on appelle `read` sur la chaîne de caractères pour obtenir le nombre. Ainsi, cette déclaration signifie que le résultat peut être de n'importe quel type membre des classes de types `Num` et `Read` (comme `Int`, `Float`, etc.).

Pour la liste d'éléments `["2", "3", "+"]`, notre fonction va commencer à plier par la gauche. La pile initiale sera `[]`. Elle appellera la fonction de pli avec `[]` en tant que pile (accumulateur) et `"2"` en tant qu'élément. Puisque cet élément n'est pas un opérateur, il sera lu avec `read` et ajouté au début de `[]`. La nouvelle pile est donc `[2]`, et la fonction de pli sera appelée avec `[2]` pour pile et `["3"]` pour élément, produisant une nouvelle pile `[3, 2]`. Ensuite, elle est appelée pour la troisième fois avec `[3, 2]` pour pile et `"+"` pour élément. Cela cause le dépilement des deux nombres, qui sont alors sommés, et leur résultat est empilé. La pile finale est `[5]`, qui est le nombre qu'on retourne.

Jouons avec notre fonction :

```
ghci> solveRPN "10 4 3 + 2 * -"
-4
ghci> solveRPN "2 3 +"
5
ghci> solveRPN "90 34 12 33 55 66 + * - +"
-3947
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037
ghci> solveRPN "90 3 -"
87
```

Cool, elle marche ! Une chose sympa avec cette fonction, c'est qu'elle peut être facilement modifiée pour supporter une variété d'autres opérateurs. Ils n'ont même pas besoin d'être binaires. Par exemple, on peut créer un opérateur `"log"` qui ne dépile qu'un nombre, et empile son logarithme. On peut aussi faire un opérateur ternaire qui dépile trois nombres et empile le résultat, comme `"sum"` qui dépile tous les nombres et empile leur somme.

Modifions notre fonction pour gérer quelques nouveaux opérateurs. Par simplicité, on va changer sa déclaration de type pour qu'elle retourne un type `Float`.

```
import Data.List

solveRPN :: String -> Float
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction (x:y:ys) "*" = (x * y):ys
            foldingFunction (x:y:ys) "+" = (x + y):ys
            foldingFunction (x:y:ys) "-" = (y - x):ys
            foldingFunction (x:y:ys) "/" = (y / x):ys
            foldingFunction (x:y:ys) "^" = (y ** x):ys
            foldingFunction (x:xs) "ln" = log x:xs
            foldingFunction xs "sum" = [sum xs]
            foldingFunction xs numberString = read numberString:xs
```

Wow, génial ! `/` est la division bien sûr, et `**` est l'exponentiation des nombres à virgule flottante. Pour l'opérateur logarithme, on filtre avec un motif à un seul élément parce qu'on n'a besoin que d'un élément pour calculer un logarithme naturel. Avec l'opérateur de somme, on retourne une pile qui n'a qu'un élément, égal à la somme de tout ce que contenait la pile jusqu'alors.

```
ghci> solveRPN "2.7 ln"
0.9932518
ghci> solveRPN "10 10 10 10 sum 4 /"
10.0
ghci> solveRPN "10 10 10 10 10 sum 4 /"
12.5
ghci> solveRPN "10 2 ^"
100.0
```

Remarquez qu'on peut inclure des nombres à virgule flottante dans nos expressions parce que `read` sait comment les lire.

```
ghci> solveRPN "43.2425 0.5 ^"
6.575903
```

Je pense que faire une fonction qui calcule des expressions arbitraires sur les nombres à virgule flottante en NPI, et qui peut être facilement extensible, en une dizaine de lignes, est plutôt génial.

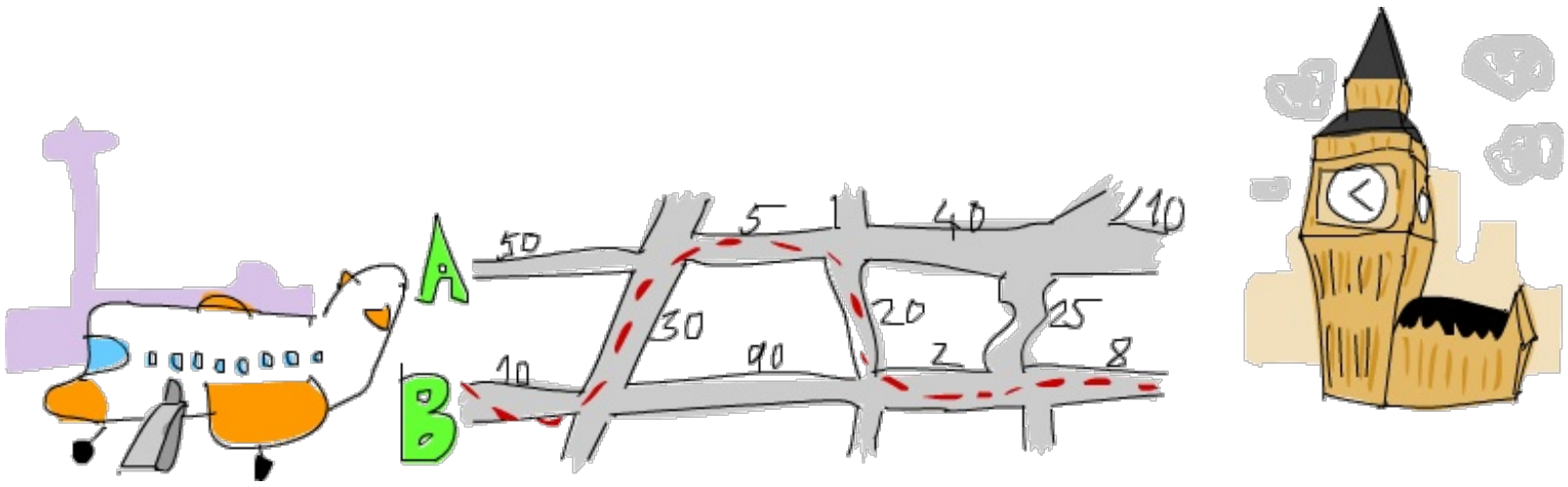
Une chose à noter à propos de cette fonction est qu'elle n'est pas très résistante aux erreurs. Si on lui donne une entrée qui n'a pas de sens, cela va juste tout planter. On fera une version résistante aux erreurs de cette fonction qui aura pour déclaration de type `solveRPN :: String -> Maybe Float` une fois qu'on aura découvert les monades (elles ne sont pas effrayantes, faites-moi confiance !). On pourrait en écrire une dès maintenant, mais ce serait un peu fastidieux parce qu'il faudrait vérifier les valeurs `Nothing` à chaque étape. Toutefois, si vous vous sentez d'humeur pour le défi, vous pouvez vous lancer ! Indice : vous pouvez utiliser `reads` pour voir si un `read` a réussi ou non.

D'Heathrow à Londres

Notre prochain problème est le suivant : votre avion vient d'atterrir en Angleterre, et vous louez une voiture. Vous avez un rendez-vous très bientôt, et devez aller de l'aéroport d'Heathrow jusqu'à Londres aussi vite que possible (mais sans vous mettre en danger !).

Il y a deux routes principales allant d'Heathrow à Londres, et un nombre de routes régionales qui croisent celles-ci. Il vous faut une quantité de temps fixe pour voyager d'une intersection à une autre. Vous devez trouver le chemin optimal pour arriver à Londres aussi vite que possible ! Vous partez du côté gauche et

pouvez soit changer de route principale, soit rouler vers Londres.



Comme vous le voyez sur l'image, le chemin le plus court d'Heathrow à Londres dans ce cas consiste à démarrer sur la route principale B, changer de route principale, avancer sur A, changer à nouveau, et continuer jusqu'à Londres sur la route B. En prenant ce chemin, il nous faut 75 minutes. Si on en avait choisi un autre, il nous faudrait plus longtemps que ça.

Notre travail consiste à créer un programme qui prend une entrée représentant un système routier, et affiche le chemin le plus court pour le traverser. Voici à quoi ressemblera l'entrée dans ce cas :

```
50
10
30
5
90
20
40
2
25
10
8
0
```

Pour découper mentalement le fichier d'entrée, lisez-le trois lignes par trois lignes, et coupez mentalement le système routier en sections. Chaque section se compose d'un morceau de route A, d'un morceau de route B, et d'une route croisant A et B. Pour conserver cette lecture trois par trois, on dit qu'il y a une dernière route transversale qui prend 0 minute à traverser. Parce qu'une fois arrivé à Londres, ce n'est plus important, on est arrivé.

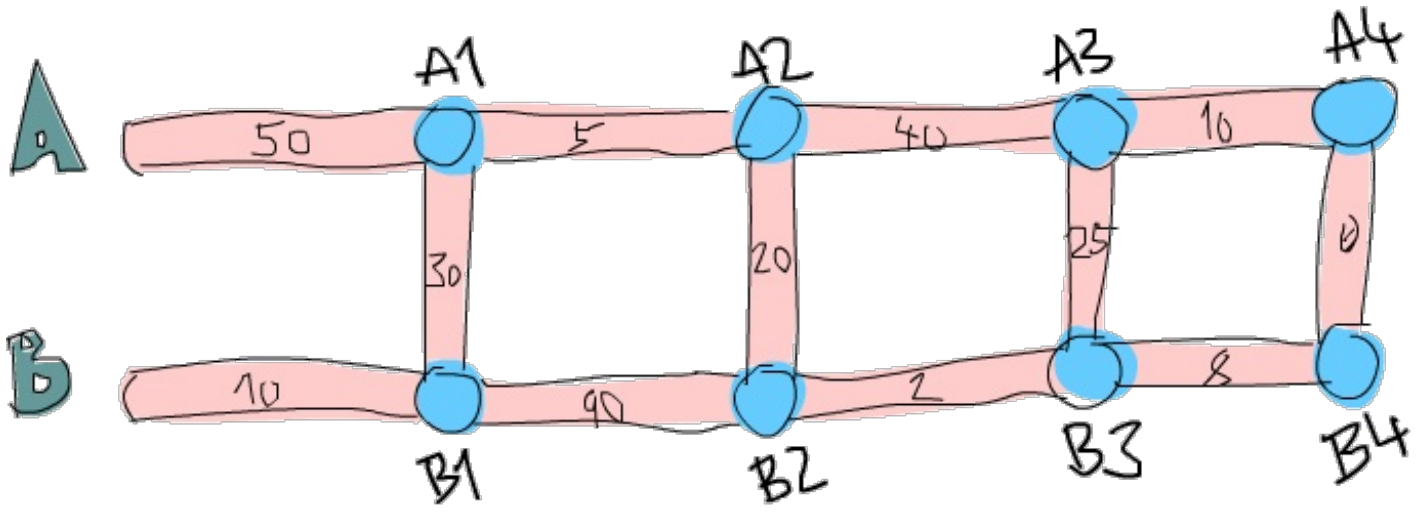
Tout comme on l'a fait en résolvant le problème de la calculatrice NPI, on va résoudre ce problème en trois étapes :

- Oublier Haskell un instant et penser à la résolution du problème à la main
- Penser à la représentation des données en Haskell
- Trouver comment opérer sur les données en Haskell pour aboutir à la solution

Dans la section sur la calculatrice NPI, on a d'abord remarqué qu'en calculant une expression à la main, on avait gardé une sorte de pile dans notre esprit et traversé l'expression un élément à la fois. On a décidé d'utiliser une liste de chaînes de caractères pour représenter l'expression. Finalement, on a utilisé un pli gauche pour traverser la liste de chaînes tout en maintenant la pile pour produire la solution.

Ok, donc, comment trouverions-nous le plus court chemin d'Heathrow à Londres à la main ? Eh bien, on peut prendre du recul, essayer de deviner ce plus court chemin, et avec un peu de chance on trouvera le bon résultat. Cette solution fonctionne pour de petites entrées, mais qu'en sera-t-il si notre route a 10 000 sections ? Ouf ! On ne saura pas non plus dire avec certitude que notre solution est optimale, on pourra simplement se dire qu'on en est plutôt sûr.

Ce n'est donc pas une bonne solution. Voici une image simplifiée de notre système routier :



Parfait, pouvez-vous trouver quel est le plus court chemin jusqu'à la première intersection de la route A (le premier point bleu sur la route A, noté A1) ? C'est plutôt

trivial. On regarde simplement s'il est plus court d'y aller directement depuis A, ou de passer par B puis traverser. Évidemment, il vaut mieux passer par B et traverser, puisque cela prend 40 minutes, alors qu'il faut 50 minutes depuis A. Qu'en est-il du plus court chemin vers B1 ? La même chose. On voit qu'il est beaucoup plus court de passer par B (10 minutes) que de passer par A et traverser, ce qui nous prendrait 80 minutes !

À présent, on connaît le chemin le plus court jusqu'à A1 (passer par B et traverser, on va dire `B, C` avec un coût de 40 minutes) et on connaît le plus court chemin jusqu'à B1 (directement par B, donc `B`, en 10 minutes). Est-ce que ce savoir nous aide pour connaître le chemin le plus court jusqu'à la prochaine intersection de chacune des routes principales ? Mon dieu, mais c'est bien sûr !

Voyons ce que serait le plus court chemin jusqu'à A2. Pour aller à A2, on peut soit aller directement d'A1 à A2, soit avancer depuis B1 et traverser (souvenez-vous, on ne peut qu'avancer ou traverser). Et puisqu'on connaît les coûts d'A1 et B1, on peut facilement trouver le meilleur chemin jusqu'à A2. Il faut 40 minutes pour aller à A1, puis 5 minutes d'A1 à A2, donc `B, C, A` coûte 45 minutes. Il ne coûte que 10 minutes pour aller à B2, mais il faut 110 minutes supplémentaires pour avancer jusqu'à B2 puis traverser ! Évidemment, le chemin le plus court jusqu'à A2 est `B, C, A`. De la même façon, le chemin le plus court jusqu'à B2 consiste à aller tout droit depuis A1 et traverser.

Vous vous demandez peut-être : mais qu'en est-il d'aller jusqu'à A2 en traversant à B1 puis en continuant tout droit ? Eh bien, on a déjà couvert la traversée de B1 à A1 quand on cherchait le meilleur moyen d'aller à A1, donc on n'a plus besoin de prendre cela en compte à l'étape suivante.

À présent qu'on connaît les meilleurs chemins pour aller à A2 et à B2, on peut répéter le processus indéfiniment jusqu'à atteindre l'arrivée. Une fois qu'on connaît les meilleurs chemins pour A4 et B4, le moins coûteux sera notre chemin optimal.

En gros, pour la deuxième section, on répète ce qu'on a fait pour la première section, mais en prenant en compte les meilleurs chemins précédents pour A comme pour B. On pourrait dire qu'on a aussi pris en compte les meilleurs chemins précédents à la première étape, en considérant que ces chemins vides avaient un coût de 0.

Voilà un résumé. Pour trouver le meilleur chemin d'Heathrow à Londres, on procède ainsi : d'abord, on cherche le meilleur chemin jusqu'à la prochaine intersection de la route A. Il n'y a que deux options : aller directement tout droit, ou commencer de la route opposée, avancer puis traverser. On se souvient du meiller chemin et du coût. On utilise la même méthode pour trouver le meilleur chemin jusqu'à la prochaine intersection de la route B. Ensuite, on se demande si le chemin pour aller à la prochaine intersection de la route A est plus court en partant de l'intersection précédente de A et en allant tout droit, ou en partant de l'intersection précédente de B, en avançant et traversant. On se souvient du chemin le plus court, de son coût, et on fait pareil pour l'intersection opposée. On continue pour chaque section jusqu'à atteindre l'arrivée, et le plus court des deux chemins jusqu'aux deux intersections de l'arrivée est notre chemin optimal !

Pour faire simple, on garde un chemin le plus court sur A et un chemin le plus court sur B, jusqu'à atteindre l'arrivée, où le plus court des deux est le chemin optimal. On sait à présent trouver le chemin le plus court à la main. Si vous disposiez d'assez de temps, de papier et de stylos, vous pourriez trouver le chemin le plus court dans un système routier arbitrairement grand.

Prochaine étape ! Comment représenter le système routier avec des types de données d'Haskell ? Une manière consiste à imaginer les points de départ et d'intersection comme des nœuds d'un graphe qui pointe vers d'autres intersections. Si on imagine que les points de départs pointent l'un vers l'autre via une route de longueur nulle, on voit que chaque intersection (ou nœud) pointe vers le nœud du côté opposé et vers le prochain nœud du même côté. À l'exception des derniers nœuds, qui ne pointent que l'un vers l'autre.

```
data Node = Node Road Road | EndNode Road
data Road = Road Int Node
```

Un nœud est soit un nœud normal, avec l'information sur la route qui mène au nœud correspondant de l'autre route et celle sur la route qui amène au nœud suivant, soit un nœud final, qui ne contient que l'information vers le nœud opposé. Une route contient pour information sa longueur et le nœud vers lequel elle pointe. Par exemple, la première partie de la route A serait `Road 50 a1` où `a1` serait un nœud `Node x y`, où `x` et `y` sont des routes qui pointent vers B1 et A2.

Un autre moyen serait d'utiliser `Maybe` pour les parties de la route qui pointent vers l'avant. Chaque nœud a une route vers le nœud opposé, mais seuls les nœuds non terminaux ont une route vers le prochain nœud.

```
data Node = Node Road (Maybe Road)
data Road = Road Int Node
```

C'est une manière correcte de représenter le système routier en Haskell et on pourrait certainement résoudre le problème avec, mais on pourrait peut-être trouver quelque chose de plus simple ? Si on revient sur notre solution à la main, on n'a jamais vérifié que les longueurs de trois parties de routes à la fois : la partie de la route A, sa partie opposée sur la route B, et la partie C qui relie l'arrivée des deux parties précédentes. Quand on cherchait le plus court chemin vers A1 et B1, on ne se souciait que des longueurs des trois premières parties, qui avaient pour longueur 50, 10 et 30. On appellera cela une section. Ainsi, le système routier qu'on utilise pour l'exemple peut facilement être représenté par quatre sections : `50, 10, 30`, `5, 90, 20`, `40, 2, 25`, and `10, 8, 0`.

Il est toujours bon de garder nos types de données aussi simples que possible, mais pas trop simples non plus !

```
data Section = Section { getA :: Int, getB :: Int, getC :: Int } deriving (Show)
type RoadSystem = [Section]
```

C'est plutôt parfait ! C'est aussi simple que possible, et je pense que ça suffira amplement pour implémenter notre solution. `Section` est un simple type de données algébriques qui contient trois entiers pour la longueur des trois parties de route de la section. On introduit également un synonyme de type, nommant `RoadSystem` une liste de sections.

On aurait aussi pu utiliser un triplet `(Int, Int, Int)` pour représenter une section de route. Utiliser des tuples plutôt que vos propres types de données algébriques est bien pour des choses petites et localisées, mais il est généralement mieux de définir des nouveaux types pour des choses comme ici. Cela donne plus d'information au système de types sur ce qu'est chaque chose. On peut utiliser `(Int, Int, Int)` pour représenter une section de route ou un vecteur dans l'espace 3D, et on peut opérer sur ces deux, mais ainsi on peut se confondre et les mélanger. Si on utilise les types de données `Section` et `Vector`, on ne peut pas accidentellement sommer un vecteur et une section de système routier.

Notre système routier d'Heathrow à Londres peut être représenté ainsi :

```
heathrowToLondon :: RoadSystem
heathrowToLondon = [Section 50 10 30, Section 5 90 20, Section 40 2 25, Section 10 8 0]
```

Tout ce dont on a besoin à présent, c'est d'implémenter la solution qu'on a trouvée précédemment en Haskell. Que devrait-être la déclaration de type de la fonction qui calcule le plus court chemin pour n'importe quel système routier ? Elle devrait prendre un système routier en paramètre, et retourner un chemin. On représentera un chemin sous forme de liste. Introduisons un type `Label`, qui sera juste une énumération `A`, `B` ou `C`. On fera également un synonyme de type : `Path`.

```
data Label = A | B | C deriving (Show)
type Path = [(Label, Int)]
```

Notre fonction, appelons-la `optimalPath`, devrait donc avoir pour déclaration de type `optimalPath :: RoadSystem -> Path`. Si elle est appelée avec le système routier `heathrowToLondon`, elle doit retourner le chemin suivant :

```
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8)]
```

On va devoir traverser la liste des sections de la gauche vers la droite, et garder de côté le chemin optimal sur A et celui sur B. On va accumuler le meilleur chemin pendant qu'on traverse la liste, de la gauche vers la droite. Est-ce que ça sonne familier ? Ding, ding, ding ! Et oui, c'est un PLI GAUCHE !

Quand on déroulait la solution à la main, il y avait une étape qu'on répétait à chaque fois. Ça impliquait de vérifier les chemins optimaux sur A et B jusqu'ici et la section courante pour produire les nouveaux chemins optimaux sur A et B. Par exemple, au départ, nos chemins optimaux sont `[]` et `[]` pour A et B respectivement. On examinait la section `Section 50 10 30` et on a conclu que le nouveau chemin optimal jusqu'à A1 était `[(B,10),(C,30)]` et que le chemin optimal jusqu'à B1 était `[(B,10)]`. Si vous regardez cette étape comme une fonction, elle prend une paire de chemins et une section, et produit une nouvelle paire de chemins. Le type est `(Path, Path) -> Section -> (Path, Path)`. Implémentons directement cette fonction, elle sera forcément utile.

Indice : elle sera utile parce que `(Path, Path) -> Section -> (Path, Path)` peut être utilisée comme la fonction binaire du pli gauche, qui doit avoir pour type `a -> b -> a`.

```
roadStep :: (Path, Path) -> Section -> (Path, Path)
roadStep (pathA, pathB) (Section a b c) =
    let priceA = sum $ map snd pathA
        priceB = sum $ map snd pathB
        forwardPriceToA = priceA + a
        crossPriceToA = priceB + b + c
        forwardPriceToB = priceB + b
        crossPriceToB = priceA + a + c
        newPathToA = if forwardPriceToA <= crossPriceToA
                     then (A,a):pathA
                     else (C,c):(B,b):pathB
        newPathToB = if forwardPriceToB <= crossPriceToB
                     then (B,b):pathB
                     else (C,c):(A,a):pathA
    in  (newPathToA, newPathToB)
```

Que se passe-t-il là ? D'abord, il faut calculer le temps optimal sur la route A en se basant sur le précédent temps optimal sur A, et de même pour B. On fait `sum $ map snd pathA`, donc si `pathA` est quelque chose comme `[(A,100),(C,20)]`, `priceA` devient `120`.

`forwardPriceToA` est le temps optimal pour aller à la prochaine intersection de A en venant de la précédente intersection de A. Il est égal au meilleur temps jusqu'au précédent A, plus la longueur de la partie A de la section courante. `crossPriceToA` est le temps optimal pour aller au prochain A en venant du précédent B et en traversant. Il est égal au meilleur temps jusqu'au précédent B, plus la longueur B de la section, plus la longueur C de la section. On détermine `forwardPriceToB` et `crossPriceToB` de manière analogue.

À présent qu'on connaît le meilleur chemin jusqu'à A et B, il ne nous reste plus qu'à trouver les nouveaux chemins jusqu'à A et B. S'il est moins cher d'aller à A en avançant simplement, on définit `newPathToA` comme `(A, a):pathA`. On prépose simplement le `Label` `A` et la longueur de la section `a` au chemin optimal jusqu'au point A précédent. En gros, on dit que le meilleur chemin jusqu'à la prochaine intersection sur A est le meilleur chemin jusqu'à la précédente intersection sur A, suivie par la section en avant sur A. Souvenez-vous qu'`A` n'est qu'une étiquette, alors que `a` a pour type `Int`. Pourquoi est-ce qu'on prépose plutôt que de faire `pathA ++ [(A, a)]` ? Eh bien, ajouter un élément en début de liste (aussi appelé *conser*) est bien plus rapide que de l'ajouter à la fin. Cela implique que notre chemin sera à l'envers une fois qu'on aura plié la liste avec cette fonction, mais il sera simple de le renverser plus tard. S'il est moins cher d'aller à la prochaine intersection sur A en avançant sur B puis en traversant, alors `newPathToA` est le chemin jusqu'à la précédente intersection sur B, suivie d'une section en avant sur B et d'une section traversante. On fait de même pour `newPathToB`, à l'exception que tout est dans l'autre sens.

Finalement, on retourne `newPathToA` et `newPathToB` sous forme de paire.

Testons cette fonction sur la première section d'`heathrowToLondon`. Puisque c'est la première section, les paramètres contenant les meilleurs chemins jusqu'à l'intersection précédente sur A et sur B sera une paire de listes vides.

```
ghci> roadStep ([], []) (head heathrowToLondon)
([(C,30),(B,10)],[(B,10)])
```

Souvenez-vous, les chemins sont renversés, lisez-les donc de la droite vers la gauche. Ici, on peut lire que le meilleur chemin jusqu'au prochain A consiste à avancer sur B puis à traverser, et le meilleur chemin jusqu'au prochain B consiste à avancer directement sur B.

Astuce d'optimisation : quand on fait `priceA = sum $ map snd pathA`, on calcule le prix à partir du chemin à chaque étape de l'algorithme. On pourrait éviter cela en implémentant `roadStep` comme une fonction ayant pour type `(Path, Path, Int, Int) -> Section -> (Path, Path, Int, Int)`, où les entiers réprésenteraient le prix des chemins A et B.

Maintenant qu'on a une fonction qui prend une paire de chemins et une section et produit un nouveau chemin optimal, on peut simplement plier sur une liste de sections. `roadStep` sera appelée avec `([], [])` et la première section, et retournera une paire de chemins optimaux pour cette section. Puis, elle sera appelée avec cette paire de chemins et la prochaine section, et ainsi de suite. Quand on a traversé toutes les sections, il nous reste une paire de chemins optimaux, et le plus court des deux est notre réponse. Avec ceci en tête, on peut implémenter `optimalPath`.

```
optimalPath :: RoadSystem -> Path
optimalPath roadSystem =
    let (bestAPath, bestBPath) = foldl roadStep ([],[]) roadSystem
    in  if sum (map snd bestAPath) <= sum (map snd bestBPath)
            then reverse bestAPath
            else reverse bestBPath
```

On plie depuis la gauche `roadSystem` (souvenez-vous, c'est une liste de sections) avec pour accumulateur initial une paire de chemins vides. Le résultat de ce pli est une paire de chemins, qu'on filtre par motif pour obtenir les chemins. Puis, on regarde lequel est le plus rapide, et on retourne celui-ci. Avant de le retourner, on le renverse, parce que les chemins étaient jusqu'alors renversées parce qu'on avait choisi de conser plutôt que de postposer.

Testons cela !

```
ghci> optimalPath heathrowToLondon
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8),(C,0)]
```

C'est le bon résultat ! Génial ! Il diffère légèrement de celui auquel on s'attendait, parce qu'il y a une étape `(C, 0)` à la fin, qui signifie qu'on traverse la route à son arrivée à Londres, mais comme cette traversée n'a aucun coût, le résultat reste valide.

On a la fonction qui trouve le chemin optimal, il ne nous reste plus qu'à lire une représentation littérale d'un système routier de l'entrée standard, le convertir en un type `RoadSystem`, lancer notre fonction `optimalPath` dessus, et afficher le chemin.

D'abord, créons une fonction qui prend une liste et la découpe en groupes de même taille. On va l'appeler `groupsOf`. Pour le paramètre `[1..10]`, `groupsOf 3` devrait retourner `[[1,2,3],[4,5,6],[7,8,9],[10]]`.

```
groupsOf :: Int -> [a] -> [[a]]
groupsOf 0 _ = undefined
groupsOf _ [] = []
groupsOf n xs = take n xs : groupsOf n (drop n xs)
```

Une fonction récursive standard. Pour un `xs` valant `[1..10]` et un `n` égal à `3`, ceci est égal à `[1,2,3] : groupsOf 3 [4,5,6,7,8,9,10]`. Quand la récursivité s'achève, on a notre liste en groupes de trois éléments. Et voici notre fonction `main`, qui lit l'entrée standard, crée un `RoadSystem` et affiche le chemin le plus court.

```
import Data.List

main = do
    contents <- getContents
    let threes = groupsOf 3 (map read $ lines contents)
        roadSystem = map (\[a,b,c] -> Section a b c) threes
        path = optimalPath roadSystem
        pathString = concat $ map (show . fst) path
        pathPrice = sum $ map snd path
    putStrLn $ "The best path to take is: " ++ pathString
    putStrLn $ "The price is: " ++ show pathPrice
```

D'abord, on récupère le contenu de l'entrée standard. Puis, on appelle `lines` sur ce contenu pour convertir quelque chose comme `"50\n10\n30\n…` en `["50","10","30"…` et ensuite, on mappe `read` là-dessus pour obtenir une liste de nombres. On appelle `groupsOf 3` sur cette liste pour la changer en une liste de listes de longueur 3. On mappe la lambda `(\[a,b,c] -> Section a b c)` sur cette liste de listes. Comme vous le voyez, la lambda prend une liste de longueur 3, et la transforme en une section. Donc `roadSystem` est à présent notre système routier et a un type correct, c'est-à-dire `RoadSystem` (ou `[Section]`). On appelle `optimalPath` avec ça et on obtient le chemin optimal et son coût dans une représentation textuelle agréable qu'on affiche.

Enregistrons le texte suivant :

```
50
10
30
5
90
20
40
2
25
10
8
0
```

dans un fichier `paths.txt` et donnons-le à notre programme.

```
$ cat paths.txt | runhaskell heathrow.hs
The best path to take is: BCACBBC
The price is: 75
```

Ça fonctionne à merveille ! Vous pouvez utiliser vos connaissances du module `Data.Random` pour générer un système routier plus long, que vous pouvez donner à la fonction qu'on a écrite. Si vous obtenez un dépassement de pile, essayez de remplacer `foldl` par `foldl'`, sa version stricte.

← Entrées et sorties Table des matières Foncteurs, foncteurs applicatifs et monoïdes →

Foncteurs, foncteurs applicatifs et monoïdes

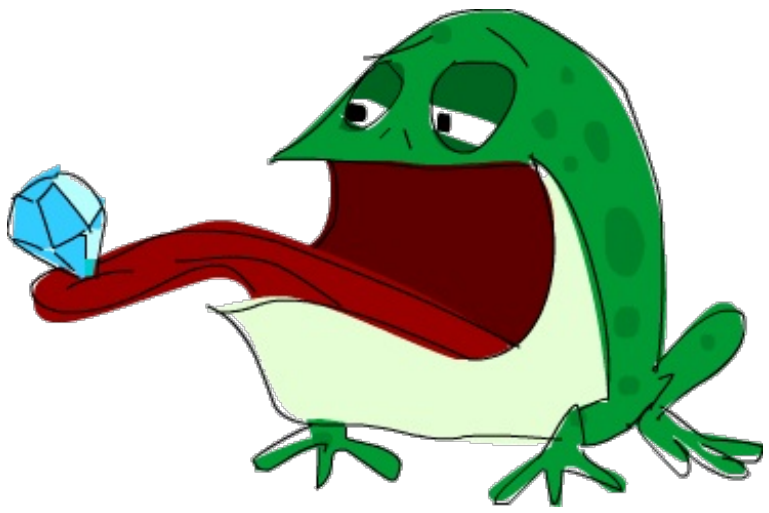
← Résoudre des problèmes fonctionnellement | Table des matières | Pour une poignée de monades →

La combinaison de la pureté, des fonctions d'ordre supérieur, des types de données algébriques paramétrés, et des classes de types que propose Haskell nous permet d'implémenter du polymorphisme à un niveau bien plus élevé que dans la plupart des autres langages. On n'a pas besoin de penser aux types comme appartenant à une hiérarchie de types. À la place, on pense à ce que les types peuvent faire, et on les connecte aux classes de types appropriées. Un `Int` peut se faire passer pour beaucoup de choses. Il peut se faire passer pour quelque chose dont on peut tester l'égalité, pour quelque chose qu'on peut ordonner, pour quelque chose qu'on peut énumérer, etc.

Les classes de types sont ouvertes, ce qui veut dire qu'on peut définir nos propres types de données, réfléchir à ce qu'ils peuvent faire, et les connecter aux classes de types qui définissent ses comportements. Grâce à cela, et au système de types d'Haskell qui nous permet de savoir beaucoup de choses sur une fonction rien qu'en regardant sa déclaration de type, on peut définir des classes de types qui définissent des comportements très généraux et abstraits. On a vu des classes de types définissant des opérateurs pour tester l'égalité ou comparer des choses. Ce sont des comportements assez abstraits et élégants, mais on ne les considère pas comme cela parce qu'on fait souvent la même chose dans nos vies réelles. On a récemment découvert les foncteurs, qui sont simplement des choses sur lesquelles on peut mapper. C'est un exemple de propriété utile mais plutôt abstraite que peuvent décrire les classes de types. Dans ce chapitre, on va regarder les foncteurs d'un peu plus près, ainsi que des versions plus fortes et utiles de foncteurs, appelés foncteurs applicatifs. On va aussi s'intéresser aux monoïdes, qui sont un peu comme des chaussettes.

Foncteurs revisités

On a déjà parlé des foncteurs dans leur propre petite section. Si vous ne l'avez pas encore lue, vous devriez probablement le faire à présent, ou plus tard, quand vous aurez plus de temps. Ou faire semblant de l'avoir lue.



Ceci étant, un petit rappel : les foncteurs sont des choses sur lesquelles on peut mapper, comme des listes, des `Maybe`, des arbres, et d'autres. En Haskell, ils sont définis par la classe de types `Functor`, qui n'a qu'une méthode de classe de type, `fmap`, ayant pour type `fmap :: (a -> b) -> f a -> f b`. Cela dit : donne-moi une fonction qui prend un `a` et retourne un `b`, et une boîte avec un (ou plusieurs) `a` à l'intérieur, et je te donnerai une boîte avec un (ou plusieurs) `b` à l'intérieur. Elle applique grosso-modo la fonction aux éléments dans la boîte.

Un conseil. Souvent, l'analogie de la boîte aide à se faire une intuition de la façon dont fonctionnent les foncteurs, et plus tard, on utilisera probablement la même analogie pour les foncteurs applicatifs et les monades. C'est une analogie correcte pour aider les débutants à comprendre les foncteurs, mais ne la prenez pas trop littéralement, parce que pour certains foncteurs, l'analogie est fortement tirée par les cheveux. Un terme plus correct pour définir ce qu'est un foncteur serait un contexte de calcul. Le contexte peut être que le calcul peut avoir renvoyé une valeur ou échoué (`Maybe` et `Either a`) ou que le calcul renvoie plusieurs valeurs (les listes), ce genre de choses.

Si on veut faire d'un constructeur de types une instance de `Functor`, il doit avoir pour sorte `* -> *`, ce qui signifie qu'il doit prendre exactement un type concret en paramètre de type. Par exemple, `Maybe` peut être une instance parce qu'il prend un paramètre de type pour produire un type concret, comme `Maybe Int` ou `Maybe String`. Si un constructeur de types prend deux paramètres, comme `Either`, il faut l'appliquer partiellement jusqu'à ce qu'il ne prenne plus qu'un paramètre de type. Ainsi, on ne peut pas écrire `instance Functor Either where`, mais on peut écrire `instance Functor (Either a) where`, et alors en imaginant que `fmap` ne fonctionne que pour les `Either a`, elle aurait pour déclaration de type `fmap :: (b -> c) -> Either a b -> Either a c`. Comme vous pouvez le voir, la partie `Either a` est fixée, parce que `Either a` ne prend qu'un paramètre de type, alors qu'`Either` en prend deux, et ainsi `fmap :: (b -> c) -> Either b -> Either c` ne voudrait rien dire.

On sait à présent comment plusieurs types (ou plutôt, des constructeurs de types) sont des instances de `Functor`, comme `[]`, `Maybe`, `Either a` et un type `Tree` qu'on a créé nous-mêmes. On a vu comment l'on pouvait mapper des fonctions sur ceux-ci pour notre plus grand bien. Dans cette section, on va découvrir deux autres instances de foncteurs, `IO` et `(->) r`.

Si une valeur a pour type, mettons, `IO String`, cela signifie que c'est une action I/O qui, lorsqu'elle est exécutée, ira dans le monde réel et nous récupèrera une chaîne de caractères, qu'elle rendra comme son résultat. On peut utiliser `<-` dans la syntaxe *do* pour lier ce résultat à un nom. On a mentionné que les actions

I/O sont comme des boîtes avec des petits pieds qui sortent chercher des valeurs dans le monde à l'extérieur pour nous. On peut inspecter ce qu'elles ont ramené, mais après inspection, on doit les envelopper à nouveau dans `IO`. En pensant à cette analogie de boîte avec des petits pieds, on peut voir comment `IO` agit comme un foncteur.

Voyons comment faire d'`IO` une instance de `Functor`. Quand on `fmap` une fonction sur une action I/O, on veut obtenir une action I/O en retour qui fait la même chose, mais applique notre fonction sur la valeur résultante.

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```

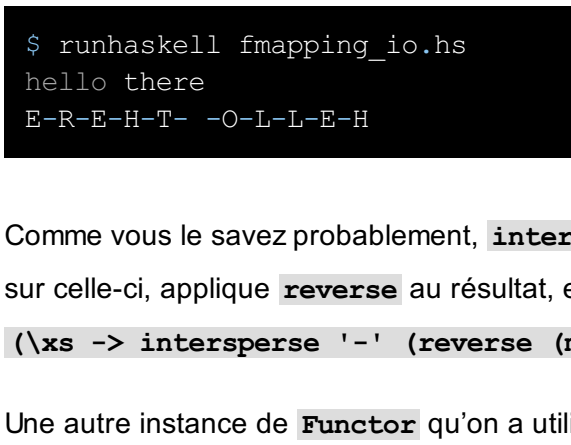
Le résultat du mappage de quelque chose sur une action I/O sera une action I/O, donc on utilise immédiatement la notation *do* pour coller deux actions en une. Dans l'implémentation de `fmap`, on crée une nouvelle action I/O qui commence par exécuter l'action I/O originale, et on appelle son résultat `result`. Puis, on fait `return (f result)`. `return` est, comme vous le savez, une fonction qui crée une action I/O qui ne fait rien, mais présente un résultat. L'action produite par un bloc *do* aura toujours pour résultat celui de sa dernière action. C'est pourquoi on utilise `return` pour créer une action I/O qui ne fait pas grand chose, mais présente `f result` comme le résultat de l'action I/O composée.

On peut jouer un peu avec pour se faire une intuition. C'est en fait assez simple. Regardez ce code :

```
main = do line <- getLine
          let line' = reverse line
          putStrLn $ "You said " ++ line' ++ " backwards!"
          putStrLn $ "Yes, you really said" ++ line' ++ " backwards!"
```

On demande une ligne à l'utilisateur, et on la lui rend, mais renversée. Voici comment réécrire ceci en utilisant `fmap` :

```
main = do line <- fmap reverse getLine
          putStrLn $ "You said " ++ line ++ " backwards!"
          putStrLn $ "Yes, you really said" ++ line ++ " backwards!"
```



Tout comme on peut `fmap reverse` sur `Just "blah"` pour obtenir `Just "halb"`, on peut `fmap reverse` sur `getLine`. `getLine` est une action I/O qui a pour type `IO String` et mapper `reverse` sur elle nous donne une action I/O qui va aller dans le monde réel et récupérer une ligne, puis appliquer `reverse` dessus. Tout comme on peut appliquer une fonction à quelque chose enfermé dans une boîte `Maybe`, on peut appliquer une fonction à quelque chose enfermé dans une boîte `IO`, seulement la boîte doit toujours aller dans le monde réel pour obtenir quelque chose. Ensuite, lorsqu'on la lie à quelque chose avec `<-`, le nom sera lié au résultat auquel `reverse` aura déjà été appliquée.

L'action I/O `fmap (++"!") getLine` se comporte comme `getLine`, mais ajoute toujours `"!"` à son résultat !

Si on regarde ce que le type de `fmap` serait si elle était limitée à `IO`, ce serait `fmap :: (a -> b) -> IO a -> IO b`. `fmap` prend une fonction et une action I/O et retourne une nouvelle action I/O comme l'ancienne, mais avec la fonction appliquée à son résultat.

Si jamais vous liez le résultat d'une action I/O à un nom, juste pour ensuite appliquer une fonction à ce nom et lui donner un nouveau nom, utilisez plutôt `fmap`, ce sera plus joli. Si vous voulez appliquez des transformations multiples à une donnée dans un foncteur, vous pouvez soit déclarer une fonction dans l'espace de nom global, soit utiliser une lambda expression, ou idéalement, utiliser la composition de fonctions :

```
import Data.Char
import Data.List

main = do line <- fmap (intersperse '-' . reverse . map toUpper) getLine
          putStrLn line
```

```
$ runhaskell fmapping_io.hs
hello there
E-R-E-H-T- -O-L-L-E-H
```

Comme vous le savez probablement, `intersperse '-' . reverse . map toUpper` est une fonction qui prend une chaîne de caractères, mappe `toUpper` sur celle-ci, applique `reverse` au résultat, et applique `intersperse '-'` à ce résultat. C'est comme écrire `(\xs -> intersperse '-' (reverse (map toUpper xs)))`, mais plus joli.

Une autre instance de `Functor` qu'on a utilisée tout du long sans se douter qu'elle était un foncteur est `(->) r`. Vous êtes sûrement un peu perdu à présent, qu'est-ce que ça veut dire `(->) r` ? Le type de fonctions `r -> a` peut être réécrit `(->) r a`, tout comme `2 + 3` peut être réécrit `(+) 2 3`. Quand on regarde

`(->) r a`, on peut voir `(->)` sous un nouveau jour, et s'apercevoir que c'est juste un constructeur de types qui prend deux paramètres de types, tout comme `Either`. Mais souvenez-vous, on a dit qu'un constructeur de types doit prendre exactement un paramètre pour être une instance de `Functor`. C'est pourquoi `(->)` ne peut pas être une instance de `Functor`, mais si on l'applique partiellement en `(->) r`, ça ne pose plus de problème. Si la syntaxe nous permettait d'appliquer partiellement les constructeurs de types avec des sections (comme l'on peut partiellement appliquer `+` en faisant `(2+)`, qui est équivalent à `(+) 2`), vous pourriez réécrire `(->) r` comme `(r ->)`. Comment est-ce que les fonctions sont-elles des foncteurs ? Eh bien, regardons l'implémentation, qui se trouve dans `Control.Monad.Instances`.

Généralement, on indique une fonction qui prend n'importe quoi et retourne n'importe quoi `a -> b`. `r -> a` est identique, on a juste choisi d'autres lettres pour les variables de type.

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

Si la syntaxe le permettait, on aurait pu écrire :

```
instance Functor (r ->) where
    fmap f g = (\x -> f (g x))
```

Mais elle ne le permet pas, donc on doit l'écrire de la première façon.

Tout d'abord, pensons au type de `fmap`. C'est `fmap :: (a -> b) -> f a -> f b`. À présent, remplaçons mentalement les `f`, qui ont pour rôle d'être notre foncteur, par des `(->) r`. On fait cela pour voir comment `fmap` doit se comporter pour cette instance. On obtient `fmap :: (a -> b) -> ((->) r a) -> ((->) r b)`. Maintenant, on peut réécrire `(->) r a` et `(->) r b` de manière infixe en `r -> a` et `r -> b`, comme on l'écrit habituellement pour les fonctions. On a donc `fmap :: (a -> b) -> (r -> a) -> (r -> b)`.

Hmmm OK. Mapper une fonction sur une fonction produit une fonction, tout comme mapper une fonction sur un `Maybe` produit un `Maybe` et mapper une fonction sur une liste produit une liste. Qu'est-ce que le type `fmap :: (a -> b) -> (r -> a) -> (r -> b)` nous indique-t-il ? Eh bien, on voit que la fonction prend une fonction de `a` vers `b` et une fonction de `r` vers `a`, et retourne une fonction de `r` vers `b`. Cela ne vous rappelle rien ? Oui ! La composition de fonctions ! On connecte la sortie de `r -> a` à l'entrée de `a -> b` pour obtenir une fonction `r -> b`, ce qui est exactement ce que fait la composition de fonctions. Si vous regardez comment l'instance est définie ci-dessus, vous verrez qu'on a juste composé les fonctions. Une autre façon d'écrire cette instance serait :

```
instance Functor ((->) r) where
    fmap = (.)
```

Cela rend évident le fait qu'utiliser `fmap` sur des fonctions sert juste à composer. Faites `:m + Control.Monad.Instances`, puisque c'est là que l'instance est définie, et essayez de jouer à mapper sur des fonctions.

```
ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a
ghci> fmap (*3) (+100) 1
303
ghci> (*3) `fmap` (+100) $ 1
303
ghci> (*3) . (+100) $ 1
303
ghci> fmap (show . (*3)) (*100) 1
"300"
```

On peut appeler `fmap` de façon infixe pour souligner la ressemblance avec `.`. Dans la deuxième ligne d'entrée, on mappe `(*3)` sur `(+100)`, ce qui retourne une fonction qui prend une entrée, appelle `(+100)` sur celle-ci, puis appelle `(*3)` sur ce résultat. On appelle cette fonction sur la valeur `1`.

Est-ce que l'analogie des boîtes fonctionne toujours ici ? Avec un peu d'imagination, oui. Quand on fait `fmap (+3)` sur `Just 3`, il est facile d'imaginer le `Maybe` boîte qui a un contenu sur lequel on applique la fonction `(+3)`. Mais qu'en est-il quand on fait `fmap (*3) (+100)` ? Eh bien, vous pouvez imaginer `(+100)` comme une boîte qui contient son résultat futur. Comme on imaginait une action I/O comme une boîte qui irait chercher son résultat dans le monde réel. Faire `fmap (*3)` sur `(+100)` crée une autre fonction qui se comporte comme `(+100)`, mais avant de produire son résultat, applique `(*3)` dessus. Ainsi, on voit que `fmap` se comporte comme `.` pour les fonctions.

Le fait que `fmap` soit la composition de fonctions quand elle est utilisée sur des fonctions n'est pas très utile pour l'instant, mais c'est tout du moins intéressant. Cela tord aussi un peu notre esprit et nous fait voir comment des choses agissant plutôt comme des calculs que comme des boîtes (tel `IO` et `(->) r`) peuvent elles aussi être des foncteurs. La fonction mappée sur un calcul agit comme ce calcul, mais modifie son résultat avec cette fonction.

Avant de regarder les règles que `fmap` doit respecter, regardons encore une fois son type. Celui-ci est `fmap :: (a -> b) -> f a -> f b`. Il manque la contrainte de classe `(Functor f) =>`, mais on l'oublie par concision, parce qu'on parle de foncteurs donc on sait ce que signifie `f`. Quand on a découvert les fonctions curryfiées, on a dit que toutes les fonctions Haskell prennent un unique paramètre. Une fonction `a -> b -> c` ne prend en réalité qu'un paramètre `a` et retourne une fonction `b -> c`, qui prend un paramètre et retourne un `c`. C'est pourquoi, si l'on appelle une fonction avec trop peu de paramètres (c'est-à-dire qu'on l'applique partiellement), on obtient en retour une fonction qui prend autant de paramètres qu'il en manquait (on repense à nouveau à nos fonctions comme prenant plusieurs paramètres). Ainsi, `a -> b -> c` peut être écrit `a -> (b -> c)` pour faire apparaître la curryfication.

Dans la même veine, si l'on écrit `fmap :: (a -> b) -> (f a -> f b)`, on peut imaginer `fmap` non pas comme une fonction qui prend une fonction et un foncteur pour retourner un foncteur, mais plutôt comme une fonction qui prend une fonction, et retourne une nouvelle fonction, similaire à l'ancienne, mais qui prend et retourne des foncteurs. Elle prend une fonction `a -> b`, et retourne une fonction `f a -> f b`. On dit qu'on *lifte* la fonction. Jouons avec cette idée en utilisant la commande `:t` de GHCi :

```
ghci> :t fmap (*2)
fmap (*2) :: (Num a, Functor f) => f a -> f a
ghci> :t fmap (replicate 3)
fmap (replicate 3) :: (Functor f) => f a -> f [a]
```

L'expression `fmap (*2)` est une fonction qui prend un foncteur `f` sur des nombres, et retourne un foncteur sur des nombres. Ce foncteur peut être une liste, un `Maybe`, un `Either String`, peu importe. L'expression `fmap (replicate 3)` prend un foncteur de n'importe quel type et retourne un foncteur sur des listes d'éléments de ce type.

Quand on dit *foncteur sur des nombres*, vous pouvez imaginer un *foncteur qui contient des nombres*. La première version est un peu plus sophistiquée et techniquement correcte, mais la seconde est plus simple à saisir.

Ceci est encore plus apparent si l'on applique partiellement, mettons, `fmap (++"!")` et qu'on lie cela à un nom dans GHCi.

Vous pouvez imaginer `fmap` soit comme une fonction qui prend une fonction et un foncteur, et mappe cette fonction sur le foncteur, ou bien comme une fonction qui prend une fonction et la lifte en une fonction sur des foncteurs. Les deux visions sont correctes et équivalentes en Haskell.

Le type `fmap (replicate 3) :: (Functor f) => f a -> f [a]` signifie que la fonction marchera sur n'importe quel foncteur. Ce qu'elle fera exactement dépendra du foncteur en question. Si on utilise `fmap (replicate 3)` sur une liste, l'implémentation de `fmap` pour les listes sera choisie, c'est-à-dire `map`. Si on l'utilise sur `Maybe a`, cela appliquera `replicate 3` à la valeur dans le `Just`, alors qu'un `Nothing` restera un `Nothing`.

```
ghci> fmap (replicate 3) [1,2,3,4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]
ghci> fmap (replicate 3) (Just 4)
Just [4,4,4]
ghci> fmap (replicate 3) (Right "blah")
Right ["blah","blah","blah"]
ghci> fmap (replicate 3) Nothing
Nothing
ghci> fmap (replicate 3) (Left "foo")
Left "foo"
```

Maintenant, nous allons voir les **lois des foncteurs**. Afin que quelque chose soit un foncteur, il doit satisfaire quelques lois. On attend de tous les foncteurs qu'ils présentent certaines propriétés et comportements fonctoriels. Ils doivent se comporter de façon fiable comme des choses sur lesquelles on peut mapper. Appeler `fmap` sur un foncteur devrait seulement mapper une fonction sur le foncteur, rien de plus. Ce comportement est décrit dans les lois des foncteurs. Il y en a deux, et toute instance de `Functor` doit les respecter. Elles ne sont cependant pas vérifiées automatiquement par Haskell, il faut donc les tester soi-même.

La première loi des foncteurs dit que si l'on mappe la fonction `id` sur un foncteur, le foncteur retourné doit être identique au foncteur original. Si on écrit cela plus formellement, cela signifie que `fmap id = id`. En gros, cela signifie que si l'on fait `fmap id` sur un foncteur, cela doit être pareil que de faire simplement `id` sur ce foncteur. Souvenez-vous, `id` est la fonction identité, qui retourne son paramètre à l'identique. Elle peut également être écrite `\x -> x`. Si l'on voit le foncteur comme quelque chose sur laquelle on peut mapper, alors la loi `fmap id` peut sembler triviale ou évidente.

Voyons si cette loi tient pour quelques foncteurs.

```
ghci> fmap id (Just 3)
Just 3
ghci> id (Just 3)
Just 3
ghci> fmap id [1..5]
[1,2,3,4,5]
ghci> id [1..5]
[1,2,3,4,5]
ghci> fmap id []
[]
ghci> fmap id Nothing
Nothing
```

Si on regarde l'implémentation de `fmap`, par exemple pour `Maybe`, on peut se rendre compte que la première loi des foncteurs est respectée.

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

Imaginons qu'`id` soit à la place de `f` dans l'implémentation. On voit que si l'on `fmap id` sur `Just x`, le résultat sera `Just (id x)`, et puisqu'`id` retourne son paramètre à l'identique, on peut en déduire que `Just (id x)` est égal à `Just x`. Ainsi, mapper `id` sur une valeur `Maybe` construite avec `Just` retourne la même valeur.

Voir que mapper `id` sur une valeur `Nothing` retourne la même valeur est trivial. De ces deux équations de l'implémentation de `fmap`, on déduit que `fmap id = id` est vrai.

La seconde loi dit que composer deux fonctions, et mapper le résultat sur un foncteur doit être identique à mapper d'abord une des fonctions sur le foncteur, puis mapper l'autre sur le résultat. Formellement, on veut `fmap (f . g) = fmap f . fmap g`. Ou, d'une autre façon, pour tout foncteur F, on souhaite : `fmap (f . g) F = fmap f (fmap g F)`.

Si l'on peut montrer qu'un type obéit à ces deux lois des foncteurs, alors on peut avoir l'assurance qu'il aura les mêmes propriétés fondamentales vis-à-vis du mappage. On sait que lorsque l'on fait `fmap` sur ce type, il ne se passera rien d'autre qu'un mappage, et il se comportera comme une chose sur laquelle on mappe, i.e. un foncteur. On peut voir si un type respecte la seconde loi en regardant l'implémentation de `fmap` pour ce type, et en utilisant la même méthode qu'on a utilisée pour voir si `Maybe` obéissait à la première loi.

Si vous le voulez, on peut vérifier que la seconde loi des foncteurs est vérifiée par `Maybe`. Si on fait `fmap (f . g)` sur `Nothing`, on obtient `Nothing`, parce que quelle que soit la fonction mappée sur `Nothing`, on obtient `Nothing`. De même, faire `fmap f (fmap g Nothing)` retourne `Nothing`, pour la même raison. OK, voir que la seconde loi tient pour une valeur `Nothing` de `Maybe` était plutôt facile, presque trivial.

Et si c'est une valeur `Just something` ? Eh bien, si l'on fait `fmap (f . g) (Just x)`, on voit de l'implémentation que c'est `Just ((f . g) x)`, qui est, évidemment, `Just (f (g x))`. Si l'on fait `fmap f (fmap g (Just x))`, on voit que `fmap g (Just x)` est `Just (g x)`. Donc, `fmap f (fmap g (Just x))` est égal à `fmap f (Just (g x))`, et de l'implémentation, on voit que cela est égal à `Just (f (g x))`.

Si vous êtes un peu perdu dans cette preuve, ne vous inquiétez pas. Soyez sûr de bien comprendre comme fonctionne la composition de fonctions. Très souvent, on peut voir intuitivement que ces lois sont respectées parce que le type se comporte comme un conteneur ou comme une fonction. Vous pouvez aussi essayer sur un tas de valeurs et vous convaincre que le type suit bien les lois.

Intéressons-nous au cas pathologique d'un constructeur de types instance de `Functor` mais qui n'est pas vraiment un foncteur, parce qu'il ne satisfait pas les lois. Mettons qu'on ait un type :

```
data CMaybe a = CNothing | CJust Int a deriving (Show)
```

Le C ici est pour *compteur*. C'est un type de données qui ressemble beaucoup à `Maybe a`, mais la partie `Just` contient deux champs plutôt qu'un. Le premier champ du constructeur de valeurs `CJust` aura toujours pour type `Int`, et ce sera une sorte de compteur, et le second champ sera de type `a`, qui vient du paramètre de type, et son type dépendra bien sûr du type concret qu'on choisit pour `CMaybe a`. Jouons avec notre type pour se faire une intuition.

```
ghci> CNothing
CNothing
```

```
ghci> CJust 0 "haha"
CJust 0 "haha"
ghci> :t CNothing
CNothing :: CMaybe a
ghci> :t CJust 0 "haha"
CJust 0 "haha" :: CMaybe [Char]
ghci> CJust 100 [1,2,3]
CJust 100 [1,2,3]
```

Si on utilise le constructeur `CNothing`, il n'y a pas de champs, alors que le constructeur `CJust` a un champ entier et un champ de n'importe quel type. Créons une instance de `Functor` pour laquelle, chaque fois qu'on utilise `fmap`, la fonction est appliquée au second champ, alors que le premier est incrémenté de 1.

```
instance Functor CMaybe where
    fmap f CNothing = CNothing
    fmap f (CJust counter x) = CJust (counter+1) (f x)
```

C'est un peu comme l'implémentation de `Maybe`, à l'exception que lorsqu'on fait `fmap` sur une valeur qui n'est pas une boîte vide (donc sur une valeur `CJust`), en plus d'appliquer la fonction au contenu, on augmente le compteur de 1. Tout va bien pour l'instant, on peut même jouer un peu avec :

```
ghci> fmap (++"ha") (CJust 0 "ho")
CJust 1 "hoha"
ghci> fmap (++"he") (fmap (++"ha") (CJust 0 "ho"))
CJust 2 "hohahe"
ghci> fmap (++"blah") CNothing
CNothing
```

Est-ce que cela vérifie les lois des foncteurs ? Pour démontrer que ce n'est pas le cas, il nous suffit de trouver un contre-exemple.

```
ghci> fmap id (CJust 0 "haha")
CJust 1 "haha"
ghci> id (CJust 0 "haha")
CJust 0 "haha"
```

Ah ! La première loi dit que mapper `id` sur un foncteur équivaut à appeler `id` sur ce même foncteur, mais dans cet exemple, ce n'est pas vrai pour notre foncteur `CMaybe`. Bien qu'il soit membre de la classe `Functor`, il n'obéit pas aux lois des foncteurs, et n'est donc pas un foncteur. Si quelqu'un utilisait notre type `CMaybe` comme un foncteur, il s'attendrait à ce qu'il obéisse aux lois des foncteurs, comme tout bon foncteur. Mais `CMaybe` échoue à être un foncteur bien qu'il puisse faire semblant d'en être un, et l'utiliser en tant que tel pourrait amener à un code erroné. Lorsqu'on utilise un foncteur, cela ne devrait pas importer que l'on compose des fonctions avant de les mapper ou que l'on mappe les fonctions une à une à la suite. Mais pour `CMaybe`, cela compte, parce qu'il compte combien de fois on a mappé sur lui. Pas cool ! Si l'on voulait que `CMaybe` obéisse aux lois des foncteurs, il faudrait que le champ `Int` ne soit pas modifié lors d'un `fmap`.

Au départ, les lois des foncteurs peuvent sembler un peu déroutantes et peu nécessaires, mais on finit par s'apercevoir que si un type obéit à ces lois, alors on peut présumer son comportement. Si un type obéit aux lois des foncteurs, on sait qu'appeler `fmap` sur une valeur va seulement mapper la fonction, rien de plus. Cela amène à un code plus abstrait et extensible, parce qu'on peut utiliser ces lois pour raisonner sur le comportement de n'importe quel foncteur et créer des fonctions qui opèrent de façon fiable sur n'importe quel foncteur.

Toutes les instances de `Functor` de la bibliothèque standard obéissent à ces lois, mais vous pouvez vérifier si vous ne me croyez pas. Et la prochaine fois que vous créez une instance de `Functor`, prenez une minute pour vous assurer qu'elle obéit aux lois des foncteurs. Une fois que vous avez utilisé assez de foncteurs, vous développez une intuition pour ces propriétés et comportements qu'ils ont en commun et il n'est plus dur de se rendre compte intuitivement qu'un type obéit ou non aux lois des foncteurs. Mais même sans intuition, vous pouvez toujours regarder l'implémentation ligne par ligne et voir si les lois sont respectées, ou trouver un contre-exemple.

On peut aussi regarder les foncteurs comme des choses qui retournent des valeurs dans un contexte. Par exemple, `Just 3` retourne la valeur `3` dans le contexte où il peut ne pas y avoir de valeur retournée. `[1, 2, 3]` retourne trois valeurs - `1`, `2` et `3`, le contexte étant qu'il peut y avoir aucune ou plusieurs valeurs. La fonction `(+3)` retourne une valeur, qui dépend du paramètre qu'on lui donne.

Si vous imaginez les foncteurs comme des choses qui retournent des valeurs, vous pouvez imaginer mapper sur des foncteurs comme attacher des transformations à la sortie du foncteur pour changer les valeurs qu'il retourne. Lorsqu'on fait `fmap (+3) [1, 2, 3]`, on attache la transformation `(+3)` à la sortie de `[1, 2, 3]`, donc quand on observe un des nombres que la liste retourne, `(+3)` lui est appliqué. Un autre exemple est celui du mappage sur des fonctions. Quand on fait `fmap (+3) (*3)`, on attache la transformation `(+3)` à ce qui sortira de `(*3)`. On voit ainsi mieux pourquoi utiliser `fmap` sur des fonctions consiste juste à composer les fonctions (`fmap (+3) (*3)` est égal à `(+3) . (*3)`, qui est égal à `\x -> ((x*3)+3)`), parce qu'on prend une fonction comme `(*3)` et qu'on attache la transformation `(+3)` à sa sortie. Le résultat est toujours une fonction, seulement quand on lui donne une valeur, elle sera multipliée par trois, puis passera par la transformation attachée où on lui ajoutera trois. C'est ce qui se passe avec la composition.

Foncteurs applicatifs

Dans cette section, nous allons nous intéresser aux foncteurs applicatifs, qui sont des foncteurs gonflés aux hormones, représentés en Haskell par la classe de types `Applicative`, située dans le module `Control.Applicative`.

Comme vous le savez, les fonctions en Haskell sont curryfiées par défaut, ce qui signifie qu'une fonction qui semble prendre plusieurs paramètres en prend en fait un seul et retourne une fonction qui prend le prochain paramètre, et ainsi de suite. Si une fonction a pour type `a -> b -> c`, on dit généralement qu'elle prend deux paramètres et retourne un `c`, mais en réalité, elle prend un `a` et retourne une fonction `b -> c`. C'est pourquoi on peut appeler une fonction en faisant `f x y` ou bien `(f x) y`. Ce mécanisme nous permet d'appliquer partiellement des fonctions en les appelant avec trop peu de paramètres, ce qui résulte en des fonctions que l'on peut passer à d'autres fonctions.

Jusqu'ici, lorsqu'on mappait des fonctions sur des foncteurs, on mappait généralement des fonctions qui ne prenaient qu'un paramètre. Mais que se passe-t-il lorsqu'on souhaite mapper `*`, qui prend deux paramètres, sur un foncteur ? Regardons quelques exemples concrets. Si l'on a `Just 3` et que l'on fait `fmap (*) (Just 3)`, qu'obtient-on ? En regardant l'implémentation de l'instance de `Functor` de `Maybe`, on sait que si c'est une valeur `Just something`, elle applique la fonction sur le `something` à l'intérieur du `Just`. Ainsi, faire `fmap (*) (Just 3)` retourne `Just ((*) 3)`, qui peut être écrit `Just (* 3)` en utilisant une section. Intéressant ! On obtient une fonction enveloppée dans un `Just` !

```
ghci> :t fmap (++) (Just "hey")
fmap (++) (Just "hey") :: Maybe ([Char] -> [Char])
ghci> :t fmap compare (Just 'a')
fmap compare (Just 'a') :: Maybe (Char -> Ordering)
ghci> :t fmap compare "A LIST OF CHARS"
fmap compare "A LIST OF CHARS" :: [Char -> Ordering]
ghci> :t fmap (\x y z -> x + y / z) [3,4,5,6]
fmap (\x y z -> x + y / z) [3,4,5,6] :: (Fractional a) => [a -> a -> a]
```

Si l'on mappe `compare`, qui a pour type `(Ord a) => a -> a -> Ordering` sur une liste de caractères, on obtient une liste de fonctions ayant pour type `Char -> Ordering`, parce que la fonction `compare` est partiellement appliquée sur les caractères de la liste. Ce n'est pas une liste de fonctions `(Ord a) => a -> Ordering`, parce que le premier `a` appliqué était un `Char` et donc le deuxième `a` doit également être un `Char`.

On voit qu'en mappant des fonctions à plusieurs paramètres sur des foncteurs, on obtient des foncteurs qui contiennent des fonctions. Que peut-on faire de ceux-ci ? Pour commencer, on peut mapper sur ceux-ci des fonctions qui prennent en paramètre une fonction, parce que ce qui est dans le foncteur sera donné à la fonction mappée comme paramètre.

```
ghci> let a = fmap (*) [1,2,3,4]
ghci> :t a
a :: [Integer -> Integer]
ghci> fmap (\f -> f 9) a
[9,18,27,36]
```

Mais si l'on a une valeur fonctorielle `Just (3 *)` et une autre valeur fonctorielle `Just 3`, et qu'on souhaite sortir la fonction de `Just (3 *)` pour la mapper sur `Just 5` ? Avec des foncteurs normaux, on est bloqué, parce qu'ils ne supportent que la mappage de fonctions normales sur des foncteurs. Même lorsqu'on a mappé `\f -> f 9` sur un foncteur qui contenait une fonction, on ne mappait qu'une fonction normale sur celui-ci. Mais `fmap` ne nous permet pas de mapper une fonction qui est dans un foncteur sur un autre foncteur. On pourrait filtrer par motif sur le constructeur `Just` pour récupérer la fonction, puis la mapper sur `Just 5`, mais on voudrait une solution plus générale et abstraite à ce problème, qui fonctionnerait pour tous les foncteurs.

Je vous présente la classe de types `Applicative`. Elle réside dans le module `Control.Applicative` et définit deux méthodes, `pure` et `<*>`. Elle ne fournit pas d'implémentation par défaut pour celles-ci, il faut donc les définir toutes deux nous-mêmes si l'on veut faire de quelque chose un foncteur applicatif. La classe est définie ainsi :

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Ces trois petites lignes de définition nous disent beaucoup de choses ! Commençons par la première ligne. Elle débute la définition de la classe `Applicative` et introduit une contrainte de classe. Elle dit que si l'on veut faire d'un constructeur de types une instance d'`Applicative`, il doit d'abord être membre de `Functor`. C'est pourquoi, si l'on sait qu'un constructeur de type est membre d'`Applicative`, alors c'est aussi un `Functor`, et on peut donc utiliser `fmap` sur lui.

La première méthode qu'elle définie est appelée `pure`. Sa déclaration de type est `pure :: a -> f a`. `f` joue le rôle de notre instance de foncteur applicatif. Puisqu'Haskell a un très bon système de types, et puisqu'une fonction ne peut que prendre des paramètres et retourner une valeur, on peut dire beaucoup de choses avec une déclaration de type, et ceci n'est pas une exception. `pure` prend une valeur de n'importe quel type, et retourne un foncteur applicatif encapsulant cette valeur en son intérieur. Quand on dit *en son intérieur* , on se réfère à nouveau à l'analogie de la boîte, bien qu'on ait vu qu'elle ne soit pas toujours la plus à même d'expliquer ce qu'il se trame. La déclaration `a -> f a` est tout de même plutôt descriptive. On prend une valeur et on l'enveloppe dans

un foncteur applicatif.

Une autre façon de penser à `pure` consiste à dire qu'elle prend une valeur et la met dans un contexte par défaut (ou pur) - un contexte minimal qui retourne cette valeur.

La fonction `<*>` est très intéressante. Sa déclaration de type est `f (a -> b) -> f a -> f b`. Cela ne vous rappelle rien ? Bien sûr, `fmap :: (a -> b) -> f a -> f b`. C'est un peu un `fmap` amélioré. Alors que `fmap` prend une fonction et un foncteur, et applique cette fonction à l'intérieur du foncteur, `<*>` prend un foncteur contenant une fonction et un autre foncteur, et d'une certaine façon extrait la fonction du premier foncteur pour la mapper sur l'autre foncteur. Quand je dis *extrait*, je veux en fait presque dire *exécute* puis extrait, peut-être même *séquence*. On verra pourquoi bientôt.

Regardons l'implémentation de l'instance d'`Applicative` de `Maybe`.

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

Encore une fois, de la définition de la classe, on voit que le `f` qui joue le rôle du foncteur applicatif doit prendre un type concret en paramètre, donc on écrit `instance Applicative Maybe where` plutôt que `instance Applicative (Maybe a) where`.

Tout d'abord, `pure`. On a dit précédemment qu'elle était supposée prendre quelque chose et l'encapsuler dans un foncteur applicatif. On a écrit `pure = Just`, parce que les constructeurs de valeurs comme `Just` sont des fonctions normales. On aurait pu écrire `pure x = Just x`.

Ensuite, on a la définition de `<*>`. On ne peut pas extraire de fonction d'un `Nothing`, parce qu'il ne contient rien. Donc si l'on essaie d'extraire une fonction d'un `Nothing`, le résultat est `Nothing`. Si vous regardez la définition de classe d'`Applicative`, vous verrez qu'il y a une classe de contrainte `Functor`, qui signifie que les deux paramètres de `<*>` sont des foncteurs. Si le premier paramètre n'est pas un `Nothing`, mais un `Just` contenant une fonction, on veut mapper cette fonction sur le second paramètre. Ceci prend également en compte le cas où le second paramètre est `Nothing`, puisque faire `fmap` sur un `Nothing` retourne `Nothing`.

Donc, pour `Maybe`, `<*>` extrait la fonction de la valeur de gauche si c'est un `Just` et la mappe sur la valeur de droite. Si n'importe lequel des paramètres est `Nothing`, le résultat est `Nothing`.

Ok, cool, super. Testons cela.

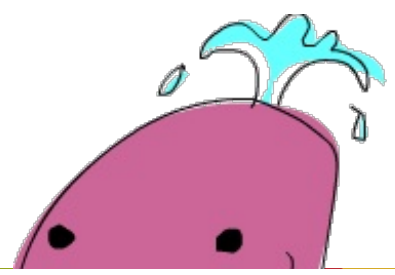
```
ghci> Just (+3) <*> Just 9
Just 12
ghci> pure (+3) <*> Just 10
Just 13
ghci> pure (+3) <*> Just 9
Just 12
ghci> Just (++"hahah") <*> Nothing
Nothing
ghci> Nothing <*> Just "woot"
Nothing
```

On voit que faire `pure (+3)` où `Just (+3)` est équivalent dans ce cas. Utilisez `pure` quand vous utilisez des valeurs `Maybe` dans un contexte applicatif (i.e. quand vous les utilisez avec `<*>`), sinon utilisez `Just`. Les quatre premières lignes entrées montrent comment la fonction extraite est mappée, mais dans ces exemples, elle aurait tout aussi bien pu être mappée immédiatement sur les foncteurs. La dernière ligne est intéressante parce qu'on essaie d'extraire une fonction d'un `Nothing` et de le mapper sur quelque chose, ce qui résulte en un `Nothing` évidemment.

Avec des foncteurs normaux, on peut juste mapper une fonction sur un foncteur, et on ne peut plus récupérer ce résultat hors du foncteur de manière générale, même lorsque le résultat est une fonction appliquée partiellement. Les foncteurs applicatifs, quant à eux, permettent d'opérer sur plusieurs foncteurs avec la même fonction. Regardez ce bout de code :

```
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing
```

Que se passe-t-il là ? Regardons, étape par étape. `<*>` est associatif à gauche, ce qui signifie que `pure (+) <*> Just 3 <*> Just 5` est équivalent à `(pure (+) <*> Just 3) <*> Just 5`. Tout d'abord, la fonction `+` est mise dans un foncteur, qui est dans ce cas une valeur `Maybe`. Donc, au départ, on a `pure (+)` qui est égal à `Just (+)`. Ensuite, `Just (+) <*> Just 3` a lieu. Le résultat est `Just (3+)`. Ceci à cause de l'application partielle. Appliquer la fonction `+` seulement sur `3` résulte en une fonction qui prend un paramètre, et lui ajoute 3. Finalement,



`Just (3+) <*> Just 5` est effectuée, ce qui résulte en `Just 8`.

N'est-ce pas génial !? Les foncteurs applicatifs et le style applicatif d'écrire `pure f <*> x <*> y <*> …` nous permettent de prendre une fonction qui attend des paramètres qui ne sont pas nécessairement enveloppés dans des foncteurs, et d'utiliser cette fonction pour opérer sur des valeurs qui sont dans des contextes fonctoriels. La fonction peut prendre autant de paramètres qu'on le souhaite, parce qu'elle est partiellement appliquée étape par étape, à chaque occurrence de `<*>`.

Cela devient encore plus pratique et apparent si l'on considère le fait que `pure f <*> x` est égal à `fmap f x`. C'est une des lois des foncteurs applicatifs. Nous les regarderons plus en détail plus tard, pour l'instant, on peut voir intuitivement que c'est le cas. Pensez-y, ça tombe sous le sens. Comme on l'a dit plus tôt, `pure` place une valeur dans un contexte par défaut. Si l'on place une fonction dans un contexte par défaut, et qu'on l'extrait de ce contexte pour l'appliquer à une valeur dans un autre foncteur applicatif, on a fait la même chose que de juste mapper cette fonction sur le second foncteur applicatif. Plutôt que d'écrire `pure f <*> x <*> y <*> …`, on peut écrire `fmap f x <*> y <*> …`. C'est pourquoi `Control.Applicative` exporte une fonction `<$>` qui est simplement `fmap` en tant qu'opérateur infixe. Voici sa définition :

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

Yo ! Petit rappel : les variables de types sont indépendantes des noms des paramètres ou des noms des valeurs en général. Le `f` de la déclaration de la fonction ici est une variable de type avec une contrainte de classe disant que tout type remplaçant `f` doit être membre de la classe `Functor`. Le `f` dans le corps de la fonction dénote une fonction qu'on mappe sur `x`. Le fait que `f` soit utilisé pour représenter ces deux choses ne signifie pas qu'elles représentent la même chose.

En utilisant `<$>`, le style applicatif devient brillant, parce qu'à présent, si l'on veut appliquer une fonction `f` sur trois foncteurs applicatifs, on peut écrire `f <$> x <*> y <*> z`. Si les paramètres n'étaient pas des foncteurs applicatifs mais des valeurs normales, on aurait écrit `f x y z`.

Regardons cela de plus près. On a une valeur `Just "johntra"` et une valeur `Just "volta"`, et on veut joindre les deux en une `String` dans un foncteur `Maybe`. On fait cela :

```
ghci> (++) <$> Just "johntra" <*> Just "volta"
Just "johntravolta"
```

Avant qu'on se penche là-dessus, comparez la ligne ci-dessus avec celle-ci :

```
ghci> (++) "johntra" "volta"
"johntravolta"
```

Génial ! Pour utiliser une fonction sur des foncteurs applicatifs, parsemez quelques `<$>` et `<*>` et la fonction opérera sur des foncteurs applicatifs et retournera un foncteur applicatif.

Quand on fait `(++) <$> Just "johntra" <*> Just "volta"`, `(++)`, qui a pour type `(++) :: [a] -> [a] -> [a]` est tout d'abord mappée sur `Just "johntra"`, résultant en une valeur comme `Just ("johntra"++)` ayant pour type `Maybe ([Char] -> [Char])`. Remarquez comme le premier paramètre de `(++)` a été avalé et les `a` sont devenus des `Char`. À présent, `Just ("johntra"++) <*> Just "volta"` est exécuté, ce qui sort la fonction du `Just` et la mappe sur `Just "volta"`, retournant `Just "johntravolta"`. Si l'une de ces deux valeurs était un `Nothing`, le résultat serait `Nothing`.

Jusqu'ici, on a seulement regardé `Maybe` dans nos exemples, et vous vous dites peut-être que les foncteurs applicatifs sont juste pour les `Maybe`. Il y a beaucoup d'autres instances d'`Applicative`, alors découvrons en plus !

Les listes (ou plutôt, le constructeur de types listes, `[]`) sont des foncteurs applicatifs. Quelle surprise ! Voici l'instance d'`Applicative` de `[]` :

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Plus tôt, on a dit que `pure` prenait des valeurs, et les plaçait dans un contexte par défaut. En d'autres mots, dans un contexte minimal qui retourne cette valeur. Le contexte minimal pour les listes serait une liste vide, `[]`, mais la liste vide représente l'absence de valeurs, donc elle ne peut pas contenir l'élément qu'on passe à `pure`. C'est pourquoi `pure` prend un élément, et le place dans une liste singleton. De manière similaire, le contexte minimal du foncteur applicatif `Maybe` aurait été `Nothing`, mais comme il représentait l'absence de valeurs, `pure` était implémenté avec `Just`.

```
ghci> pure "Hey" :: [String]
["Hey"]
ghci> pure "Hey" :: Maybe String
```

```
Just "Hey"
```

Qu'en est-il de `<*>` ? Si on imagine le type de `<*>` si elle était limitée aux listes, ce serait `(<*>) :: [a -> b] -> [a] -> [b]`. On l'implémente à l'aide d'une liste en compréhension. `<*>` doit d'une certaine façon extraire la fonction de son paramètre de gauche, et la mapper sur son paramètre de droite. Mais ici, la liste de gauche peut contenir zéro, une ou plusieurs fonctions. La liste de droite peut elle aussi contenir plusieurs valeurs. C'est pourquoi on utilise une liste en compréhension pour piocher dans les deux listes. On applique toutes les fonctions possibles de la liste de gauche sur toutes les valeurs possibles de la liste de droite. La liste résultante contient toutes les combinaisons possibles d'application d'une fonction de la liste de gauche sur une valeur de la liste de droite.

```
ghci> [(*0),(+100),(^2)] <*> [1,2,3]
[0,0,0,101,102,103,1,4,9]
```

La liste de gauche contient trois fonctions, et la liste de droite contient trois valeurs, donc la liste résultante contient neuf éléments. Chaque fonction de la liste de gauche est appliquée à chaque valeur de la liste de droite. Si l'on a une liste de fonctions qui prennent deux paramètres, on peut les appliquer entre deux listes.

```
ghci> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

Puisque `<*>` est associative à gauche, `[(+), (*)] <*> [1, 2]` est exécuté en premier, résultant en une liste équivalente à `[(1+), (2+), (1*), (2*)]`, parce que chaque fonction à gauche est appliquée sur chaque valeur de droite. Puis, `[(1+),(2+),(1*),(2*)] <*> [3,4]` est exécuté, produisant le résultat final.

Utiliser le style applicatif avec les listes est fun ! Regardez :

```
ghci> (++) <$> ["ha","heh","hmm"] <*> ["?","!","."]
["ha?","ha!","ha.","heh?","heh!","heh.","hmm?","hmm!","hmm."]
```

À nouveau, remarquez qu'on a utilisé une fonction normale qui prend deux chaînes de caractères entre deux foncteurs applicatifs à l'aide des opérateurs applicatifs appropriés.

Vous pouvez imaginer les listes comme des calculs non déterministes. Une valeur comme `100` ou `"what"` peut être vu comme un calcul déterministe qui ne renvoie qu'un seul résultat, alors qu'une liste `[1, 2, 3]` peut être vue comme un calcul qui ne peut pas se décider sur le résultat qu'il doit avoir, et nous présente donc tous les résultats possibles. Donc, quand vous faites quelque chose comme `(+) <$> [1, 2, 3] <*> [4, 5, 6]`, vous pouvez imaginer ça comme la somme de deux calculs non déterministes avec `+`, qui produit ainsi un nouveau calcul non déterministe encore moins certain de son résultat.

Le style applicatif sur les listes est souvent un bon remplaçant des listes en compréhension. Dans le deuxième chapitre, on souhaitait connaître tous les produits possibles de `[2, 5, 10]` et `[8, 10, 11]`, donc on a fait :

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

On pioche simplement dans deux listes et on applique la fonction à toutes les combinaisons d'éléments. Cela peut être fait dans le style applicatif :

```
ghci> (*) <$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
```

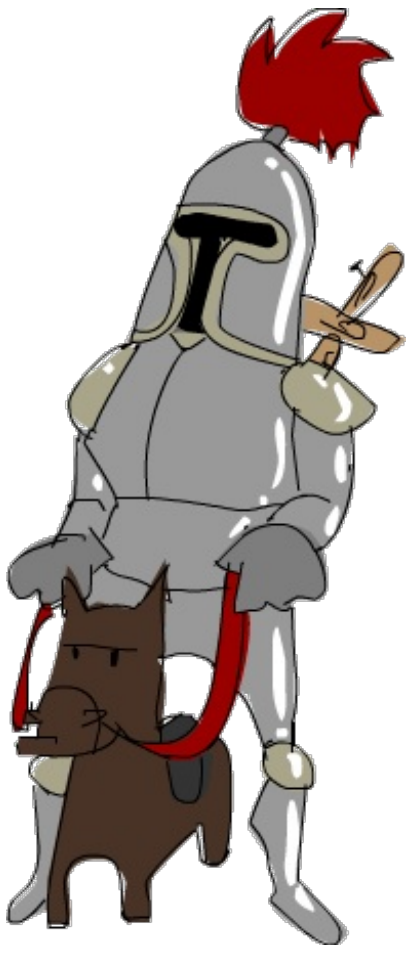
Cela me paraît plus clair, parce qu'il est plus simple de voir qu'on appelle seulement `*` entre deux calculs non déterministes. Si l'on voulait tous les produits possibles de deux listes supérieurs à 50, on ferait :

```
ghci> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

Il est facile de voir comment `pure f <*> xs` est égal à `fmap f xs` sur les listes. `pure f` est juste `[f]`, et `[f] <*> xs` appliquera chaque fonction de la liste de gauche à chaque valeur de la liste de droite, mais puisqu'il n'y a qu'une fonction à gauche, c'est comme mapper.

Une autre instance d'`Applicative` qu'on a déjà rencontrée est `IO`. Voici comment son instance est implémentée :

```
instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```



Puisque `pure` ne fait que mettre des valeurs dans un contexte minimal qui puisse toujours renvoyer ce résultat, il est sensé que `pure` soit juste `return`, parce que c'est exactement ce que fait `return` : elle crée une action I/O qui ne fait rien, mais retourne la valeur passée en résultat, sans rien écrire sur le terminal ni écrire dans un fichier.

Si `<*>` était spécialisée pour `IO`, elle aurait pour type `(<*>) :: IO (a -> b) -> IO a -> IO b`. Elle prendrait une action I/O qui retourne une fonction en résultat, et une autre action I/O, et retournerait une action I/O à partir de ces deux, qui, lorsqu'elle serait exécutée, effectuerait d'abord la première action pour obtenir la fonction, puis effectuerait la seconde pour obtenir une valeur, et retournerait le résultat de la fonction appliquée à la valeur. On utilise la syntaxe *do* pour l'implémenter ici. Souvenez-vous, la syntaxe *do* prend plusieurs actions I/O et les colle les unes aux autres, ce qui est exactement ce que l'on veut ici.

Avec `Maybe` et `[]`, on pouvait imaginer que `<*>` extrayait simplement une fonction de son paramètre de gauche et l'appliquait d'une certaine façon sur son paramètre de droite. Avec `IO`, l'extraction est toujours de la partie, mais on a également une notion d'*ordonnancement*, parce qu'on prend deux actions I/O et qu'on les ordonne, en les collant l'une à l'autre. On doit extraire la fonction de la première action I/O, mais pour pouvoir l'extraire, il faut exécuter l'action.

Considérez ceci :

```
myAction :: IO String
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b
```

C'est une action I/O qui demande à l'utilisateur d'entrer deux lignes, et retourne en résultat ces deux lignes concaténées. Ceci est obtenu en collant deux actions I/O `getLine` ensemble avec un `return`, afin que le résultat soit `a ++ b`. Une autre façon d'écrire cela en style applicatif serait :

```
myAction :: IO String
myAction = (++) <$> getLine <*> getLine
```

Ce qu'on faisait précédemment, c'était créer une action I/O qui appliquait une fonction entre les résultats de deux actions I/O, et ici c'est pareil. Souvenez-vous, `getLine` est une action I/O qui a pour type `getLine :: IO String`. Quand on utilise `<*>` entre deux foncteurs applicatifs, le résultat est un foncteur applicatif, donc tout va bien.

Le type de l'expression `(++) <$> getLine <*> getLine` est `IO String`, ce qui signifie que cette expression est une action I/O comme une autre, qui contient également une valeur résultante, comme toutes les actions I/O. C'est pourquoi on peut faire :

```
main = do
    a <- (++) <$> getLine <*> getLine
    putStrLn $ "The two lines concatenated turn out to be: " ++ a
```

Si jamais vous vous retrouvez en train de lier des actions I/O à des noms, puis à faire `return` sur l'application d'une fonction à ces noms, considérez utiliser le style applicatif, qui sera probablement plus concis et simple.

Une autre instance d'`Applicative` est `(->) r`, autrement dit les fonctions. Elles sont rarement utilisées en style applicatif, à moins que vous ne fassiez du golf avec votre code, mais elles sont tout de même d'intéressants foncteurs applicatifs, regardons donc comment leur instance est implémentée.

Si vous ne comprenez pas ce que `(->) r` signifie, lisez la section précédente où l'on expliquait que `(->) r` était un foncteur.

```
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

Lorsqu'on encapsule une valeur dans un foncteur applicatif avec `pure`, le résultat retourné doit être cette valeur. Il nous faut un contexte minimal retournant cette valeur. C'est pourquoi, dans l'instance des fonctions, `pure` prend une valeur, et crée une fonction qui ignore le paramètre qu'elle reçoit pour retourner plutôt cette valeur là. Le type de `pure` si elle était spécialisée pour l'instance `(->) r` serait `pure :: a -> (r -> a)`.

```
ghci> (pure 3) "blah"
3
```

Grâce à la curryfication, l'application des fonctions est associative à gauche, on peut donc omettre les parenthèses.

```
ghci> pure 3 "blah"
3
```

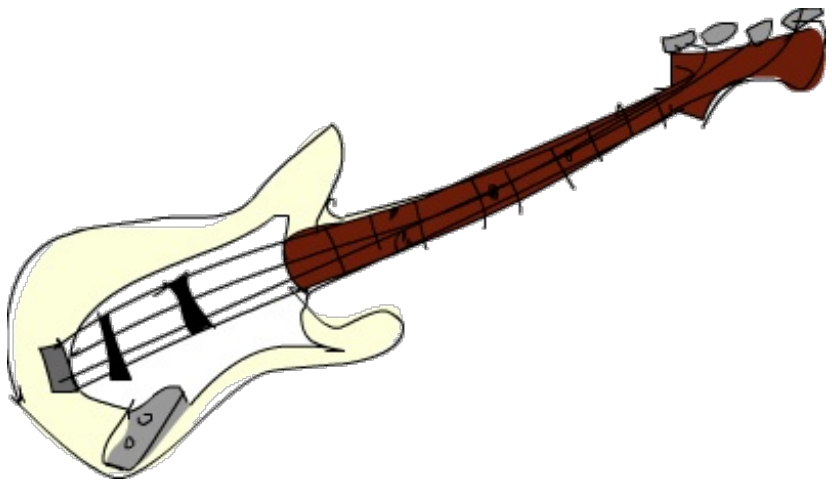
L'implémentation pour cette instance de `<*>` est un peu énigmatique, il vaut donc mieux regarder comment l'on utilise les fonctions en foncteurs applicatifs en style applicatif.

```
ghci> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
ghci> (+) <$> (+3) <*> (*100) $ 5
508
```

Appeler `<*>` avec deux foncteurs applicatifs retourne un foncteur applicatif, donc utilisé sur deux fonctions elle renvoie une fonction. Que se passe-t-il donc ici ? Quand on fait `(+) <$> (+3) <*> (*100)`, on crée une fonction qui utilise `+` sur les résultats de `(+3)` et de `(*100)`, et retourne cela. Pour montrer un exemple réel, quand on fait `(+) <$> (+3) <*> (*100) $ 5`, `(+3)` et `(*100)` sont appliquées sur `5`, retournant `8` et `500`. Puis `+` est appelée sur `8` et `500`, retournant `508`.

```
ghci> (\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
[8.0,10.0,2.5]
```

De même ici. On crée une fonction qui appelle la fonction `\x y z -> [x, y, z]` sur les résultats de `(+3)`, `(*2)` et `(/2)`. Le `5` est donné à chacune de ces trois fonctions, puis `\x y z -> [x, y, z]` est appelée avec ces trois résultats.



Vous pouvez imaginer les fonctions comme des boîtes qui contiennent leur résultat à venir, donc faire `k <$> f <*> g` crée une fonction qui appellera `k` sur les résultats à venir de `f` et `g`. Quand on fait `(+) <$> Just 3 <*> Just 5`, on utilise `+` sur des valeurs qui peuvent être là ou non, ce qui retourne donc une valeur qui peut être là ou non. Quand on fait `(+) <$> (+10) <*> (+5)`, on utilise `+` sur une valeur future de `(+10)` et `(+5)`, et le résultat sera aussi une valeur future qui ne sera produit que lorsqu'on appellera la fonction avec un paramètre.

On utilise peu les fonctions comme des foncteurs applicatifs, mais c'est tout de même intéressant. Ce n'est pas très important pour vous de comprendre comment l'instance de `(->) r` d'`Applicative` fonctionne, donc ne désespérez pas si vous ne le saisissez pas dès maintenant. Essayez de jouer avec le style applicatif pour améliorer votre intuition des fonctions en tant que foncteurs applicatifs.

Une instance d'`Applicative` que l'on n'a jamais rencontrée auparavant est `ZipList`, et elle réside dans `Control.Applicative`.

Il s'avère qu'il y a plusieurs façons pour une liste d'être un foncteur applicatif. L'une est celle qu'on a déjà vue, qui dit qu'appeler `<*>` sur une liste et une liste de valeurs retourne une liste de toutes les combinaisons possibles d'application d'une fonction de la liste de gauche sur une valeur de la liste de droite. Si l'on fait `[(+3),(*2)] <*> [1,2]`, `(+3)` est appliquée à la fois sur `1` et sur `2`, et `(*2)` est également appliquée avec `1` et `2`, ce qui nous donne une liste à quatre éléments, `[4, 5, 2, 4]`.

Cependant, `[(+3),(*2)] <*> [1,2]` pourrait aussi fonctionner de manière à ce que la première fonction de la liste de gauche soit appliquée avec la première valeur de la liste de droite, la deuxième fonction avec la deuxième valeur, et ainsi de suite. Cela retournerait une liste à deux valeurs, `[4, 4]`. Vous pouvez l'imaginer comme `[1 + 3, 2 * 2]`.

Puisqu'un type ne peut pas avoir deux instances de la même classe de types, le type `ZipList a` est introduit, et il a pour seul constructeur `ZipList` qui ne prend qu'un seul champ, de type liste. Voici son instance :

```
instance Applicative ZipList where
        pure x = ZipList (repeat x)
        ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

`<*>` fait juste ce qu'on a dit. Elle applique la première fonction avec la première valeur, la deuxième fonction avec la deuxième valeur, etc. Cela est réalisé avec `zipWith (\f x -> f x) fs xs`. La liste résultante sera aussi longue que la plus courte des deux listes, parce que c'est ce que fait `zipWith`.

`pure` est également intéressante ici. Elle prend une valeur, et la met dans une liste qui contient cette valeur un nombre infini de fois. `pure "haha"` retourne `ZipList ["haha", "haha", "haha", …`. C'est peut-être un peu déroutant, puisqu'on a dit que `pure` devait mettre une valeur dans un contexte minimal qui retournait cette valeur. Et vous vous dites sûrement qu'une liste infinie est difficilement minimale. Mais cela a du sens pour les listes zippées, parce qu'elle doit produire cette valeur à chaque position. Cela permet aussi de satisfaire la loi disant que `pure f <*> xs` doit être égal à `fmap f xs`. Si `pure 3` ne retournait que `ZipList [3]`, `pure (*2) <*> ZipList [1, 5, 10]` retournerait `ZipList [2]`, parce que la liste retournée par `zipWith` est aussi longue que la plus courte des deux. Alors que si l'on zippe une liste finie à une liste infinie, la longueur de la liste résultante sera celle de la liste finie.

Que font donc les listes zippées dans le style applicatif ? Voyons cela. Oh, le type `ZipList a` n'a pas d'instance de `Show`, il faut donc utiliser `getZipList` pour récupérer une liste normale à partir d'une liste zippée.

```
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100,100]
[101,102,103]
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100..]
[101,102,103]
ghci> getZipList $ max <$> ZipList [1,2,3,4,5,3] <*> ZipList [5,3,1,2]
[5,3,3,4]
ghci> getZipList $ (,,) <$> ZipList "dog" <*> ZipList "cat" <*> ZipList "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

La fonction `(,,)` est identique à `\x y z -> (x, y, z)`. Également, la fonction `(,)` est identique à `\x y -> (x, y)`.

En plus de `zipWith`, la bibliothèque standard a des fonctions comme `zipWith3`, `zipWith4`, jusqu'à 7. `zipWith` prend une fonction à deux paramètres et zippe deux listes avec cette fonction. `zipWith3` prend une fonction à trois paramètres et zippe trois listes à l'aide de celle-ci, et ainsi de suite. En utilisant des listes zippées avec un style applicatif, on n'a pas besoin d'avoir une fonction zip différente pour chaque nombre de listes à zipper. On utilise simplement le style applicatif pour zipper ensemble un nombre arbitraire de listes avec une fonction, c'est plutôt cool.

`Control.Applicative` définit une fonction nommée `liftA2`, qui a pour type `liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c`. Elle est définie ainsi :

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

Rien de spécial, elle applique juste une fonction entre deux foncteurs applicatifs, encapsulant le style applicatif auquel on s'était habitué. On observe cette fonction pour la raison qu'elle démontre clairement en quoi les foncteurs applicatifs sont plus puissants que les foncteurs ordinaires. Avec des foncteurs ordinaires, on peut seulement mapper des fonctions sur un foncteur. Avec des foncteurs applicatifs, on peut appliquer une fonction entre plusieurs foncteurs. Il est aussi intéressant d'imaginer le type de cette fonction comme `(a -> b -> c) -> (f a -> f b -> f c)`. Quand on regarde de cette façon, on voit que `liftA2` prend une fonction binaire normale, et la promeut en une fonction binaire sur deux foncteurs.

Voici un concept intéressant : on peut prendre deux foncteurs applicatifs, et les combiner en un foncteur applicatif qui contient en lui les résultats de ces deux foncteurs applicatifs, sous forme d'une liste. Par exemple, on a `Just 3` et `Just 4`. Imaginons que ce deuxième a une liste singleton en lui, puisque c'est très simple à réaliser :

```
ghci> fmap (\x -> [x]) (Just 4)
Just [4]
```

Ok, donc disons qu'on a `Just 3` et `Just [4]`. Comment obtenir `Just [3, 4]` ? Facile.

```
ghci> liftA2 (:) (Just 3) (Just [4])
Just [3,4]
ghci> (:) <$> Just 3 <*> Just [4]
Just [3,4]
```

Souvenez-vous, `:` est la fonction qui prend un élément, une liste, et retourne une nouvelle liste qui a cet élément en tête. À présent qu'on a `Just [3, 4]`, pourrait-on combiner ceci avec `Just 2` pour produire `Just [2, 3, 4]` ? Bien sûr. Il semble qu'on puisse combiner n'importe quel nombre de foncteurs applicatifs en un foncteur applicatif qui contient une liste de tous les résultats. Essayons d'implémenter une fonction qui prend une liste de foncteurs applicatifs et retourne un foncteur applicatif qui contienne une liste en résultat. On l'appellera `sequenceA`.

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

Ah, de la récursivité ! Tout d'abord, regardons le type. Elle transforme une liste de foncteurs applicatifs en un foncteur applicatif avec une liste. Ainsi, on peut voir apparaître notre cas de base. Si on veut transformer une liste vide en un foncteur applicatif retournant une liste, eh bien il suffit de mettre la liste vide dans un contexte minimal applicatif. Vient ensuite la récursion. Si on a une liste avec une tête et une queue (souvenez-vous, `x` est un foncteur applicatif, `xs` une liste de ceux-ci), on appelle `sequenceA` sur la queue pour obtenir un foncteur applicatif qui contient une liste. On n'a plus qu'à préposer la valeur contenue dans le foncteur `x` à cette liste, et voilà !

Ainsi, si l'on fait `sequenceA [Just 1, Just 2]`, c'est comme `(:) <$> Just 1 <*> sequenceA [Just 2]`. C'est égal à `(:) <$> Just 1 <*> ((:) <$> Just 2 <*> sequenceA [])`. Ah ! On sait que `sequenceA []` est simplement `Just []`, donc cette expression se réduit en `(:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just [])`, puis en `(:) <$> Just 1 <*> Just [2]` et finalement en `Just [1,2]` !

Un autre moyen d'implémenter `sequenceA` est à l'aide d'un pli. Souvenez-vous, presque toute fonction parcourant une liste d'éléments en accumulant un résultat peut être implémentée comme un pli.

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

On approche la liste par la droite avec un accumulateur initial égal à `pure []`. On fait `listA2 (:)` entre l'accumulateur et le dernier élément de la liste, ce qui retourne un foncteur applicatif qui contient une liste singleton. Puis l'on fait `liftA2 (:)` sur le nouveau dernier élément et l'accumulateur, et ainsi de suite, jusqu'à ce qu'il ne reste plus que l'accumulateur, contenant la liste des résultats des foncteurs applicatifs.

Testons notre fonction sur quelques foncteurs applicatifs.

```
ghci> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
ghci> sequenceA [Just 3, Nothing, Just 1]
Nothing
ghci> sequenceA [(+3),(+2),(+1)] 3
[6,5,4]
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2,3],[4,5,6],[3,4,4],[]]
[]
```

Ah ! Plutôt cool. Quand on l'utilise sur des valeurs `Maybe`, `sequenceA` crée une valeur `Maybe` avec tous les résultats dans une liste. Si l'une des valeurs est `Nothing`, alors le résultat est `Nothing`. C'est utile lorsque vous avez une liste de valeurs `Maybe`, et que vous ne vous intéressez à ses résultats que s'ils sont tous différents de `Nothing`.

Quand on l'utilise avec des fonctions, `sequenceA` prend une liste de fonctions et retourne une fonction qui retourne une liste. Dans notre exemple, on a créé une fonction qui prend un nombre en paramètre et appliquait chaque fonction d'une liste sur ce nombre, retournant la liste des résultats. `sequenceA [(+3), (+2), (+1)] 3` appellera `(+3)` avec `3`, `(+2)` avec `3` et `(+1)` avec `3`, et retourne tous ces résultats sous forme de liste.

Faire `(+) <$> (+3) <*> (*2)` créera une fonction qui prend un paramètre, le donne à la fois à `(+3)` et à `(*2)`, et appelle `+` avec ces deux résultats. Dans la même veine, il est normal que `sequenceA [(+3), (*2)]` crée une fonction qui prend un paramètre, et le donne à toutes les fonctions de la liste. Plutôt que d'appeler `+` sur le résultat de ces fonctions, une combinaison de `:` et de `pure []` est utilisée pour rassembler ces résultats en une liste, qui est le résultat de cette fonction.

Utiliser `sequenceA` est cool quand on a une liste de fonctions et qu'on veut leur donner la même entrée et voir une liste des résultats. Par exemple, si l'on a un nombre et qu'on se demande s'il satisfait tous les prédicats d'une liste. Un moyen de faire serait le suivant :

```
ghci> map (\f -> f 7) [(>4),(<10),odd]
[True,True,True]
ghci> and $ map (\f -> f 7) [(>4),(<10),odd]
True
```

Souvenez-vous, `and` prend une liste de booléens et ne retourne `True` que s'ils sont tous `True`. Un autre moyen de faire ceci, avec `sequenceA` :

```
ghci> sequenceA [(>4),(<10),odd] 7
[True,True,True]
ghci> and $ sequenceA [(>4),(<10),odd] 7
True
```

`sequenceA [(>4),(<10),odd]` crée une fonction qui prendra un nombre et le donnera à tous les prédicats de la liste `[(>4),(<10),odd]`, et retournera une liste de booléens. Elle transforme une liste ayant pour type `(Num a) => [a -> Bool]` en une fonction ayant pour type `(Num a) => a -> [Bool]`. Plutôt joli, hein ?

Parce que les listes sont homogènes, toutes les fonctions de la liste doivent avoir le même type, évidemment. Vous ne pouvez pas avoir une liste comme `[ord, (+3)]`, parce qu'`ord` prend un caractère et retourne un nombre, alors que `(+3)` prend un nombre et retourne un nombre.

Quand on l'utilise avec `[]`, `sequenceA` prend une liste de listes et retourne une liste de listes. Hmm, intéressant. Elle crée en fait des listes contenant toutes les combinaisons possibles des éléments. Par exemple, voici ceci réalisé avec `sequenceA` puis avec une liste en compréhension :

```
ghci> sequenceA [[1,2,3],[4,5,6]]
```

```
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> [[x,y] | x <- [1,2,3], y <- [4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2],[3,4]]
[[1,3],[1,4],[2,3],[2,4]]
ghci> [[x,y] | x <- [1,2], y <- [3,4]]
[[1,3],[1,4],[2,3],[2,4]]
ghci> sequenceA [[1,2],[3,4],[5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
ghci> [[x,y,z] | x <- [1,2], y <- [3,4], z <- [5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
```

C'est peut-être un peu dur à saisir, mais en jouant un peu avec, vous verrez comment ça marche. Mettons qu'on fasse **`sequenceA [[1,2],[3,4]]`**. Pour voir ce qu'il se passe, utilisons la définition **`sequenceA (x:xs) = (:) <$> x <*> sequenceA xs`** de **`sequenceA`** et son cas de base **`sequenceA [] = pure []`**. Vous n'êtes pas obligé de suivre cette évaluation, mais cela peut vous aider si vous avez du mal à imaginer comment **`sequenceA`** fonctionne sur des listes de listes, car ça peut être un peu gratiné.

- On commence avec **`sequenceA [[1,2],[3,4]]`**
- Ceci est évalué en **`(:) <$> [1,2] <*> sequenceA [[3,4]]`**
- En développant le **`sequenceA`**, on obtient **`(:) <$> [1,2] <*> ((:) <$> [3,4] <*> sequenceA [])`**
- On a atteint le cas de base, cela donne donc **`(:) <$> [1,2] <*> ((:) <$> [3,4] <*> [[]])`**
- On évalue maintenant la partie **`(:) <$> [3,4] <*> [[]]`**, qui va utiliser **`:`** sur toutes les valeurs possibles de la liste de gauche (**`3`** et **`4`** sont possibles) et avec chaque valeur possible de la liste de droite (seule **`[]`** est possible), ce qui donne **`[3:[], 4:[]]`**, autrement dit **`[[3],[4]]`**. On a donc **`(:) <$> [1,2] <*> [[3],[4]]`**
- Maintenant, **`:`** est utilisée sur toutes les valeurs possibles de la liste de gauche (**`1`** et **`2`**) avec chaque valeur possible de la liste de droite (**`[3]`** et **`[4]`**), ce qui donne **`[1:[3], 1:[4], 2:[3], 2:[4]]`**, qui est juste **`[[1,3],[1,4],[2,3],[2,4]`**

Faire **`(+) <$> [1,2] <*> [4,5,6]`** retourne un calcul non déterministe **`x + y`** où **`x`** prend toute valeur de **`[1, 2]`** et **`y`** prend toute valeur de **`[4, 5, 6]`**. On représente ceci comme une liste contenant tous les résultats possibles. De façon similaire, quand on fait **`sequence [[1,2],[3,4],[5,6],[7,8]]`**, le résultat est un calcul non déterministe **`[x, y, z, w]`**, où **`x`** prend toute valeur de **`[1, 2]`**, **`y`** prend toute valeur de **`[3, 4]`** et ainsi de suite. Pour représenter le résultat de ce calcul non déterministe, on utilise une liste, dans laquelle chaque élément est une des listes possibles. C'est pourquoi le résultat est une liste de listes.

Quand on l'utilise avec des actions I/O, **`sequenceA`** est équivalent à **`sequence`** ! Elle prend une liste d'actions I/O et retourne une action I/O qui exécutera chacune de ces actions et aura pour résultat une liste des résultats de ces actions I/O. Ceci parce que, pour changer une valeur **`[IO a]`** en une valeur **`IO [a]`**, c'est-à-dire créer une action I/O qui retourne une liste de tous les résultats, il faut ordonnancer les action I/O afin qu'elles soient effectuées l'une après l'autre lorsque l'évaluation sera forcée. On ne peut en effet pas obtenir le résultat d'une action I/O sans l'exécuter.

```
ghci> sequenceA [getLine, getLine, getLine]
heyh
ho
woo
["heyh","ho","woo"]
```

Comme les foncteurs ordinaires, les foncteurs applicatifs viennent avec quelques lois. La plus importante est celle qu'on a déjà mentionnée, qui dit que `pure f <*> x = fmap f x`. En exercice, vous pouvez prouver cette loi pour quelques uns des foncteurs applicatifs vus dans ce chapitre. Les autres lois des foncteurs applicatifs sont :

- `pure id <*> v = v`
- `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- `pure f <*> pure x = pure (f x)`
- `u <*> pure y = pure ($ y) <*> u`

On ne va pas les regarder en détail pour l'instant, parce que cela prendrait trop de pages et serait probablement ennuyeux, mais si vous vous sentez d'attaque, regardez-les et voyez si elles tiennent pour certaines instances.

En conclusion, les foncteurs applicatifs ne sont pas seulement intéressants, mais également utiles, parce qu'ils nous permettent de combiner différents calculs, comme les entrées-sorties, les calculs non déterministes, les calculs pouvant échouer, etc. en utilisant le style applicatif. Simplement en utilisant **`<$>`** et **`<*>`**, on

peut utiliser des fonctions ordinaires pour opérer uniformément sur n'importe quel nombre de foncteurs applicatifs, et profiter de la sémantique de chacun d'entre eux.

Le mot-clé newtype

Jusqu'ici, nous avons appris comment créer nos propres types de données algébriques en utilisant le mot-clé **data**. On a aussi appris comment donner des synonymes à des types avec le mot-clé **type**. Dans cette section, on va regarder comment créer de nouveaux types à partir de types existants, et pourquoi on voudrait pouvoir faire cela.

Dans la section précédente, on a vu qu'il y a plusieurs manières de définir un foncteur applicatif pour le type des listes. L'une d'elles fait prendre à `<*>` chaque fonction d'une liste passée en paramètre de gauche et la lui fait appliquer à chaque valeur d'une liste passée en paramètre de droite, retournant toutes les combinaisons possibles d'application de fonctions de la liste de gauche sur des valeurs de la liste de droite.

```
ghci> [(+1),(*100),(*5)] <*> [1,2,3]
[2,3,4,100,200,300,5,10,15]
```

La deuxième manière consiste à prendre la première fonction de la liste à gauche de `<*>`, et de l'appliquer à la première valeur de la liste de droite, puis prendre la deuxième fonction à gauche et l'appliquer à la deuxième valeur à droite, et ainsi de suite. Finalement, c'est comme zipper les deux listes ensemble. Mais les listes sont déjà des instances d'`Applicative`, alors comment les faire aussi instances d'`Applicative` de l'autre manière ? Si vous vous souvenez, on a dit que la type `ZipList a` était introduit pour cette raison, avec un seul constructeur de valeurs, `ZipList`, contenant un seul champ. On place la liste qu'on enveloppe dans ce champ. Ainsi, `ZipList` est fait instance d'`Applicative`, de manière à ce que si l'on souhaite utiliser des listes comme foncteurs applicatifs de la manière zip, on ait juste à envelopper la liste dans le constructeur `ZipList`, puis, une fois les calculs terminés, là sortir avec `getZipList` :

```
ghci> getZipList $ ZipList [(+1),(*100),(*5)] <*> ZipList [1,2,3]
[2,200,15]
```

Donc, que fait le nouveau mot-clé *newtype* ? Eh bien, réfléchissez à la façon dont vous écririez la déclaration *data* du type `ZipList a`. Une façon de l'écrire serait :

```
data ZipList a = ZipList [a]
```

Un type qui n'a qu'un constructeur de valeurs, et ce constructeur de valeurs n'a qu'un champ, qui est une liste de choses. On aurait aussi pu vouloir utiliser la syntaxe des enregistrements pour obtenir automatiquement une fonction qui extrait une liste d'une `ZipList` :

```
data ZipList a = ZipList { getZipList :: [a] }
```

Cela a l'air correct, et fonctionnerait en fait plutôt bien. Puisqu'on avait deux manières de faire d'un type existant une instance d'une classe de types, on a utilisé le mot clé *data* pour juste envelopper ce type dans un autre type, et fait de ce second type une instance de la seconde manière.

Le mot-clé *newtype* en Haskell est fait exactement pour ces cas où l'on veut juste prendre un type et l'encapsuler dans quelque chose pour le présenter comme un nouveau type. Dans la bibliothèque, `ZipList a` est définie comme :

```
newtype ZipList a = ZipList { getZipList :: [a] }
```

Plutôt que d'utiliser le mot-clé *data*, on utilise le mot-clé *newtype*. Pourquoi cela ? Eh bien, premièrement, *newtype* est plus rapide. Quand vous utilisez le mot-clé *data* pour envelopper un type, il y aura un coût à l'exécution pour faire l'encapsulage et le décapsulage. Alors qu'avec le mot-cle *newtype*, Haskell sait que vous utilisez cela juste pour encapsuler un type existant dans un nouveau type (d'où le nom du mot-clé), parce que vous voulez la même chose en interne, mais avec un nom (type) différent. Avec cela en tête, Haskell peut se débarasser de l'emballage et du déballage une fois qu'il a tenu compte du type de chaque valeur pour savoir quelle instance utiliser.

Pourquoi ne pas utiliser *newtype* tout le temps plutôt que *data* dans ce cas ? Eh bien, quand vous créez un nouveau type à partir d'un type existant en utilisant le mot-clé *newtype*, vous ne pouvez avoir qu'un seul constructeur de valeurs, et ce constructeur de valeurs ne peut avoir qu'un seul champ. Alors qu'avec *data*, vous pouvez créer des types de données qui ont plusieurs constructeurs de valeurs, chacun pouvant avoir zéro ou plusieurs champs :

```
data Profession = Fighter | Archer | Accountant

data Race = Human | Elf | Orc | Goblin

data PlayerCharacter = PlayerCharacter Race Profession
```

Quand on utilise *newtype*, on est restreint à un seul constructeur avec un seul champ.

On peut également utiliser le mot-clé *deriving* avec *newtype*, comme on le fait avec *data*. On peut dériver les instances d'`Eq`, `Ord`, `Enum`, `Bounded`, `Show` et `Read`. Si l'on souhaite dériver une instance d'une classe de types, le type qu'on enveloppe doit lui-même être membre de cette classe. C'est logique, parce que *newtype* ne fait qu'envelopper ce type. On peut donc afficher et tester l'égalité de valeurs de notre nouveau type en faisant :

```
newtype CharList = CharList { getCharList :: [Char] } deriving (Eq, Show)
```

Essayons :

```
ghci> CharList "this will be shown!"
CharList {getCharList = "this will be shown!"}
ghci> CharList "benny" == CharList "benny"
True
ghci> CharList "benny" == CharList "oisters"
False
```

Pour ce *newtype*, le constructeur de valeurs a pour type :

```
CharList :: [Char] -> CharList
```

Il prend une valeur `[Char]`, comme `"my sharona"`, et retourne une valeur `CharList`. Dans les exemples ci-dessus où l'on utilisait le constructeur de valeurs `CharList`, on voit que c'est le cas. Réciproquement, la fonction `getCharList`, qui a été générée lorsqu'on a utilisé la syntaxe des enregistrements dans notre *newtype*, a pour type :

```
getCharList :: CharList -> [Char]
```

Elle prend une valeur `CharList` et la convertit en `[Char]`. Vous pouvez imaginer ceci comme emballer et déballer, mais vous pouvez aussi l'imaginer comme une conversion d'un type vers l'autre.

Utiliser newtype pour créer des instances de classes de types

Souvent, on veut faire de nos types des instances de certaines classes de types, mais les paramètres de types ne correspondent pas à ce que l'on souhaite. Il est facile de faire une instance de `Functor` pour `Maybe`, parce que la classe de types `Functor` est définie comme :

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Donc on peut démarrer en faisant :

```
instance Functor Maybe where
```

Et implémenter `fmap`. Tous les paramètres de type s'alignent parce que `Maybe` vient prendre la place de `f` dans la définition de la classe de types `Functor` et donc, si l'on considère `fmap` comme vu des `Maybe`, elle se comporte ainsi :

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

N'est-ce pas chouette ? Maintenant, si l'on voulait faire des tuples une instance de `Functor` de manière à ce que si l'on applique `fmap` sur un tuple, elle est appliquée à la première composante du tuple ? Ainsi, faire `fmap (+3) (1, 1)` donnerait `(4, 1)`. Il s'avère qu'écrire cette instance est plutôt difficile. Avec `Maybe`, on disait juste `instance Functor Maybe where` parce que seuls les constructeurs de types prenant exactement un paramètre peuvent être des instances de `Functor`. Mais cela semble impossible pour quelque chose comme `(a, b)` de manière à ce que ce soit `a` qui change quand on utilise `fmap`. Pour contourner ce problème, on peut envelopper notre tuple dans un *newtype* de manière à ce que le deuxième paramètre de type représente la première composante du tuple :

```
newtype Pair b a = Pair { getPair :: (a,b) }
```

À présent, on peut en faire une instance de `Functor` qui mappe la fonction sur la première composante :

```
instance Functor (Pair c) where
    fmap f (Pair (x,y)) = Pair (f x, y)
```

Comme vous le voyez, on peut filtrer par motif les types définis par *newtype*. On filtre par motif pour obtenir le tuple sous-jacent, puis on applique `f` à la première composante de ce tuple, et on utilise le constructeur de valeurs `Pair` pour convertir à nouveau le tuple en `Pair b a`. Le type de `fmap` si elle ne marchait que sur nos paires serait :

```
fmap :: (a -> b) -> Pair c a -> Pair c b
```

À nouveau, on a dit `instance Functor (Pair c) where`, et ainsi, `Pair c` prend la place de `f` dans la définition de classe de `Functor` :

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Maintenant, si l'on convertit un tuple en une `Pair b a`, on peut utiliser `fmap` sur celle-ci et la fonction sera mappée sur la première composante :

```
ghci> getPair $ fmap (*100) (Pair (2,3))
(200,3)
ghci> getPair $ fmap reverse (Pair ("london calling", 3))
("gnillac nodnol",3)
```

De la paresse de newtype

On a mentionné le fait que *newtype* était généralement plus rapide que *data*. La seule chose possible avec *newtype* consiste à transformer un type existant en un autre type, donc en interne, Haskell peut représenter les valeurs des types définis par *newtype* de la même façon que celles du type original, et n'a qu'à se souvenir que leur type est distinct. Ceci fait qu'en plus d'être plus rapide, *newtype* est également plus paresseux. Regardons ce que cela signifie.

Comme on l'a dit précédemment, Haskell est paresseux par défaut, ce qui signifie que c'est seulement lorsque l'on essaie d'afficher les résultats de fonctions que ceux-ci sont calculés. De plus, seuls les calculs nécessaires pour trouver le résultat de notre fonction sont exécutés. La valeur `undefined` en Haskell représente un calcul erroné. Si on essaie de l'évaluer (c'est-à-dire, de forcer Haskell à la calculer) en l'affichant sur le terminal, Haskell piquera une crise (ou en termes techniques, lèvera une exception) :

```
ghci> undefined
*** Exception: Prelude.undefined
```

Cependant, si on crée une liste qui contient quelques valeurs `undefined` mais qu'on ne demande que sa tête, qui n'est pas `undefined`, tout se passera bien parce qu'Haskell n'a pas besoin d'évaluer les autres éléments de la liste pour trouver son premier élément :

```
ghci> head [3,4,5,undefined,2,undefined]
3
```

À présent, considérez ce type :

```
data CoolBool = CoolBool { getCoolBool :: Bool }
```

Un type de données algébrique tout droit sorti de l'usine, défini via le mot-clé *data*. Il n'a qu'un constructeur de valeurs, qui n'a qu'un champ de type `Bool`. Créons une fonction qui filtre par motif un `CoolBool` et retourne la valeur `"hello"` quelle que soit la valeur du booléen dans `CoolBool` :

```
helloMe :: CoolBool -> String
helloMe (CoolBool _) = "hello"
```

Plutôt que d'appliquer cette fonction sur une valeur `CoolBool` normale, envoyons lui plutôt une balle courbe en l'appliquant sur un `undefined` !

```
ghci> helloMe undefined
"*** Exception: Prelude.undefined
```

Aïe ! Une exception ! Pourquoi est-ce que cette exception a été levée ? Les types définis avec le mot-clé *data* peuvent avoir plusieurs constructeurs de valeurs (bien que `CoolBool` n'en ait qu'un). Ainsi, pour savoir si la valeur donnée à notre fonction correspond au motif `(CoolBool _)`, Haskell doit l'évaluer juste assez pour voir quel constructeur de valeur a été utilisé pour la créer. Et lorsqu'on essaie d'évaluer la valeur `undefined`, même un tout petit peu, une exception est levée.

Plutôt que le mot-clé *data* pour `CoolBool`, essayons d'utiliser *newtype* :

```
newtype CoolBool = CoolBool { getCoolBool :: Bool }
```

Pas besoin de changer la fonction **`helloMe`**, parce que la syntaxe de filtrage par motif est la même pour *newtype* que pour *data*. Faisons-la même chose ici et appliquons **`helloMe`** à une valeur **`undefined`** :

```
ghci> helloMe undefined
"hello"
```

Ça a marché ! Hmmm, pourquoi cela ? Eh bien, comme on l'a dit, quand on utilise **`newtype`**, Haskell peut représenter en interne les valeurs du nouveau type de la même façon que les valeurs originales. Il n'a pas besoin d'ajouter une nouvelle boîte autour, mais seulement de se rappeler du fait que les valeurs ont un type différent. Et parce qu'Haskell sait que les types créés avec le mot-clé *newtype* ne peuvent avoir qu'un seul constructeur, il n'a pas besoin d'évaluer la valeur passée à la fonction pour savoir qu'elle se conforme au motif **`(CoolBool _)`**, puisque les types créés avec *newtype* ne peuvent avoir qu'un seul constructeur de valeurs et qu'un seul champ !

Cette différence de comportement peut sembler triviale, mais elle est en fait assez importante pour nous aider à réaliser que, bien que les types créés avec *data* et *newtype* se comportent de façon similaire du point de vue du programmeur parce qu'ils ont chacun des constructeurs de valeurs et des champs, ils sont en fait deux mécanismes différents. Alors que *data* peut créer de nouveaux types à partir de rien, *newtype* ne peut créer de nouveau type qu'à partir d'un type existant. Le filtrage par motif sur les valeurs *newtype* ne sort pas les choses d'une boîte (comme le fait *data*), mais convertit plutôt un type en un autre immédiatement.

`type`* vs. *`newtype`* vs. *`data`

À ce point, vous êtes peut-être perdu concernant les différences entre *type*, *data* et *newtype*, alors rafraîchissons-nous la mémoire.

Le mot-clé **type** crée des synonymes de types. Cela signifie qu'on donne simplement un autre nom à un type existant, pour qu'il soit plus simple d'en parler. Par exemple, ceci :

```
type IntList = [Int]
```

Tout ce que cela fait, c'est nous permettre de faire référence à **`[Int]`** en tapant **`IntList`**. Les deux peuvent être utilisés de manière interchangeable. On n'obtient pas de constructeur de valeurs **`IntList`** ou quoi que ce soit du genre. Puisque **`[Int]`** et **`IntList`** sont deux façons de parler du même type, peu importe laquelle on utilise dans nos annotations de types :

```
ghci> ([1,2,3] :: IntList) ++ ([1,2,3] :: [Int])
[1,2,3,1,2,3]
```

On utilise les synonymes de types lorsqu'on veut rendre nos signatures de type plus descriptives en donnant des noms aux types qui nous disent quelque chose sur le contexte des fonctions où ils sont utilisés. Par exemple, quand on a utilisé une liste associative qui a pour type **`[(String,String)]`** pour représenter un répertoire téléphonique, on lui a donné le synonyme de type **`PhoneBook`** afin que les signatures de type des fonctions soient plus simples à lire.

Le mot-clé **newtype** sert à prendre des types existants et les emballer dans de nouveaux types, principalement parce que ça facilite la création d'instances de certaines classes de types. Quand on utilise *newtype* pour envelopper un type existant, le type qu'on obtient est différent du type original. Si l'on crée le nouveau type suivant :

```
newtype CharList = CharList { getCharList :: [Char] }
```

On ne peut pas utiliser **`++`** pour accoler une **`CharList`** et une liste ayant pour type **`[Char]`**. On ne peut même pas utiliser **`++`** pour coller ensemble deux **`CharList`**, parce que **`++`** ne fonctionne que pour les listes, et le type de **`CharList`** n'est pas une liste, même si l'on peut dire qu'il en contient une. On peut, toutefois, convertir deux **`CharList`** en listes, utiliser **`++`** sur celles-ci, puis les reconvertir en **`CharList`**.

Quand on utilise la syntaxe des enregistrements dans notre déclaration *newtype*, on obtient des fonctions pour convertir entre le type original et le nouveau type : dans une sens, avec le constructeur de valeurs de notre *newtype*, et dans l'autre sens, avec la fonction qui extrait la valeur du champ. Le nouveau type n'est pas automatiquement fait instance des classes de types du type original, donc il faut les dériver ou les écrire manuellement.

En pratique, on peut imaginer les déclarations *newtype* comme des déclarations *data* qui n'ont qu'un constructeur de valeurs, et un seul champ. Si vous vous retrouvez à écrire une telle déclaration *data*, pensez à utiliser *newtype*.

Le mot-clé **data** sert à créer vos propres types de données, et avec ceux-ci, tout est possible. Ils peuvent avoir autant de constructeurs de valeurs et de champs que vous le voulez, et peuvent être utilisés pour implémenter n'importe quel type de données algébrique vous-même. Tout, des listes aux **`Maybe`** en passant par les arbres.

Si vous voulez seulement de plus jolies signatures de type, vous voulez probablement des synonymes de types. Si vous voulez prendre un type existant et l'envelopper dans un nouveau type pour en faire une instance d'une classe de types, vous voulez sûrement un *newtype*. Et si vous voulez créer quelque chose de neuf, les chances sont bonnes que le mot-clé *data* soit ce qu'il vous faut.

Monoïdes

Les classes de types en Haskell sont uilisées pour présenter une interface pour des types qui partagent un comportement. On a commencé avec des classes de types simples comme `Eq`, pour les types dont on peut tester l'égalité, et `Ord` pour ceux qu'on peut mettre en ordre, puis on est passé à des classes plus intéressantes, comme `Functor` et `Applicative`.

Lorsqu'on crée un type, on réfléchit aux comportements qu'il supporte, i.e. pour quoi peut-il se faire passer, et à partir de cela, on décide de quelles classes de types on en fera une instance. Si cela a un sens de tester l'égalité de nos valeurs, alors on crée une instance d'`Eq`. Si l'on voit que notre type est un foncteur, alors on crée une instance de `Functor`, et ainsi de suite.

Maintenant, considérez cela : `*` est une fonction qui prend deux nombres et les multiplie entre eux. Si l'on multiplie un nombre par `1`, le résultat est toujours égal à ce nombre. Peu importe qu'on fasse `1 * x` ou `x * 1`, le résultat est toujours `x`. De façon similaire, `++` est aussi une fonction qui prend deux choses et en retourne une troisième. Au lieu de les multiplier, elle les concatène. Et tout comme `*`, elle a aussi une valeur qui ne change pas l'autre quand on la combine avec `++`. Cette valeur est la liste vide : `[]`.

```
ghci> 4 * 1
4
ghci> 1 * 9
9
ghci> [1,2,3] ++ []
[1,2,3]
ghci> [] ++ [0.5, 2.5]
[0.5,2.5]
```

On dirait que `*` et `1` partagent une propriété commune avec `++` et `[]` :

- La fonction prend deux paramètres.
- Les paramètres et la valeur retournée ont le même type.
- Il existe une valeur de ce type qui ne change pas l'autre valeur lorsqu'elle est appliquée à la fonction binaire.

Autre chose que ces deux opérations ont en commun et qui n'est pas si évident : lorsqu'on a trois valeurs ou plus et qu'on utilise la fonction binaire pour les réduire à une seule valeur, l'ordre dans lequel on applique la fonction n'importe pas. Peu importe qu'on fasse `(3 * 4) * 5` ou `3 * (4 * 5)`. De toute manière, le résultat est `60`. De même pour `++` :

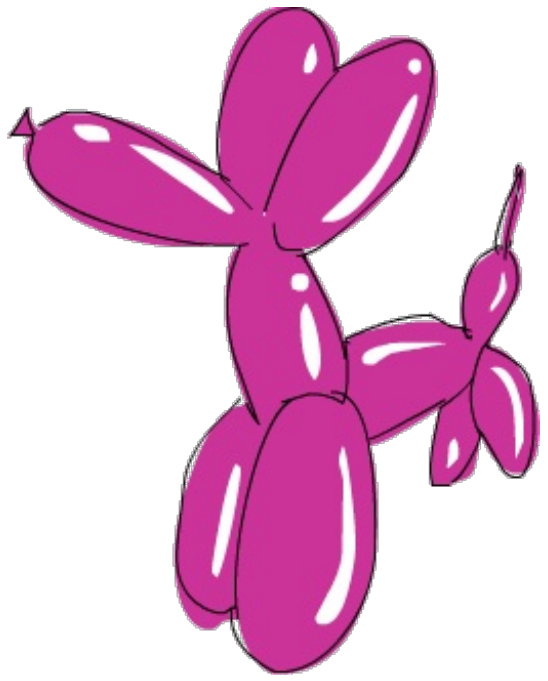
```
ghci> (3 * 2) * (8 * 5)
240
ghci> 3 * (2 * (8 * 5))
240
ghci> "la" ++ ("di" ++ "da")
"ladida"
ghci> ("la" ++ "di") ++ "da"
"ladida"
```

On appelle cette propriété l'*associativité*. `*` est associative, de même que `++`, mais par exemple `-` ne l'est pas. Les expressions `(5 - 3) - 4` et `5 - (3 - 4)` n'ont pas le même résultat.

En remarquant ces propriétés et en les écrivant, on vient de tomber sur les monoïdes ! On a un monoïde lorsqu'on dispose d'une fonction binaire associative et d'un élément neutre pour cette fonction. Quand quelque chose est un élément neutre pour une fonction, cela signifie que lorsqu'on appelle la fonction avec cette chose et une autre valeur, le résultat est toujours égal à l'autre valeur. `1` est l'élément neutre vis-à-vis de `*`, et `[]` est le neutre de `++`. Il y a beaucoup d'autres monoïdes à trouver dans le monde d'Haskell, c'est pourquoi la classe de types `Monoid` existe. Elle est pour ces types qui peuvent agir comme des monoïdes. Voyons comment la classe de types est définie :

```haskell
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

La classe de types **`Monoid`** est définie dans **`Data.Monoid`**. Prenons un peu de temps pour la découvrir.



Tout d'abord, on voit que seuls des types concrets peuvent être faits instance de **`Monoid`**, parce que le **`m`** de la définition de classe ne prend pas de paramètres. C'est différent de **`Functor`** et **`Applicative`** qui nécessitent que leurs instances soient des constructeurs de types prenant un paramètre.

La première fonction est **`mempty`**. Ce n'est pas vraiment une fonction puisqu'elle ne prend pas de paramètres, c'est plutôt une constante polymorphique, un peu comme **`minBound`** de **`Bounded`**. **`mempty`** représente l'élément neutre de ce monoïde.

Ensuite, on a **`mappend`**, qui, comme vous l'avez probablement deviné, est la fonction binaire. Elle prend deux valeurs du même type et retourne une valeur de ce type. Il est bon de mentionner que la décision d'appeler **`mappend`** de la sorte est un peu malheureuse, parce que ce nom semble impliquer qu'on essaie de juxtaposer deux choses (NDT : en anglais, "append" signifie "juxtaposer"). Bien que **`++`** prenne deux listes et les juxtapose, **`*`** ne juxtapose pas vraiment, elle multiplie seulement deux nombres. Quand nous rencontrerons d'autres instances de **`Monoid`**, on verra que la plupart d'entre elles ne juxtaposent pas non plus leurs valeurs, évitez donc de penser en termes de juxtaposition, et pensez plutôt que **`mappend`** est une fonction binaire qui prend deux valeurs monoïdales et en retourne une troisième.

La dernière fonction de la définition de la classe de types est **`mconcat`**. Elle prend une liste de valeurs monoïdales et les réduit à une unique valeur en faisant **`mappend`** entre les éléments de la liste. Elle a une implémentation par défaut, qui prend **`mempty`** comme valeur initiale et plie la liste depuis la droite avec **`mappend`**. Puisque l'implémentation par défaut convient pour la plupart des instances, on ne se souciera pas trop de **`mconcat`** pour l'instant. Quand on crée une instance de **`Monoid`**, il suffit d'implémenter **`mempty`** et **`mappend`**. La raison de la présence de **`mconcat`** ici est que, pour certaines instances, il peut y avoir une manière plus efficace de l'implémenter, mais pour la plupart des instances, l'implémentation par défaut convient.

Avant de voir des instances spécifiques de **`Monoid`**, regardons brièvement les lois des monoïdes. On a mentionné qu'il devait exister une valeur neutre vis-à-vis de la fonction binaire, et que la fonction binaire devait être associative. Il est possible de créer des instances de **`Monoid`** ne suivant pas ces règles, mais ces instances ne servent à personne parce que, quand on utilise la classe de types **`Monoid`**, on se repose sur le fait que ses instances agissent comme des monoïdes. Sinon, à quoi cela servirait-il ? C'est pourquoi, en créant des instances, on doit s'assurer qu'elles obéissent à ces lois :

- `mempty `mappend` x = x`
- `x `mappend` mempty = x`
- `(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)`

Les deux premières déclarent que **`mempty`** doit être le neutre de **`mappend`**, la troisième dit que **`mappend`** doit être associative, i.e. l'ordre dans lequel on applique **`mappend`** pour réduire plusieurs valeurs monoïdales en une n'importe pas. Haskell ne fait pas respecter ces lois, c'est donc à nous en tant que programmeur de faire attention à ce que nos instances y obéissent.

Les listes sont des monoïdes

Oui, les listes sont des monoïdes ! Comme on l'a vu, la fonction **`++`** et la liste vide **`[]`** forment un monoïde. L'instance est très simple :

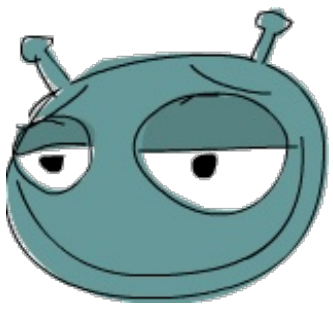
```haskell
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

Les listes sont une instance de la classe de types **`Monoid`** quel que soit le type des éléments qu'elles contiennent. Remarquez qu'on a écrit **`instance Monoid [a]`** et pas **`instance Monoid []`**, parce que **`Monoid`** nécessite un type concret en instance.

En testant cela, pas de surprises :

```haskell
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]
ghci> ("one" `mappend` "two") `mappend` "tree"
"onetwotree"
ghci> "one" `mappend` ("two" `mappend` "tree")
"onetwotree"
ghci> "one" `mappend` "two" `mappend` "tree"
"onetwotree"
ghci> "pang" `mappend` mempty
```

```
"pang"
ghci> mconcat [[1,2],[3,6],[9]]
[1,2,3,6,9]
ghci> mempty :: [a]
[]
```



Remarquez qu'à la dernière ligne, nous avons dû écrire une annotation de type explicite, parce que si l'on faisait seulement `mempty`, GHCi ne saurait pas quelle instance utiliser, on indique donc qu'on veut celle des listes. On a pu utiliser le type général `[a]` (plutôt qu'un type spécifique comme `[Int]` ou `[String]`) parce que la liste vide peut se faire passer pour n'importe quel type de liste.

Puisque `mconcat` a une implémentation par défaut, on l'obtient gratuitement lorsqu'on crée une instance de `Monoid`. Dans le cas des listes, `mconcat` s'avère être juste `concat`. Elle prend une liste de listes et l'aplatit, parce que c'est équivalent à faire `++` entre toutes les listes adjacentes d'une liste.

Les lois des monoïdes sont en effet respectées par l'instance des listes. Quand on a plusieurs listes et qu'on les `mappend` (ou `++`) ensemble, peu importe l'ordre dans lequel l'opération est effectuée, parce qu'au final, elles sont toutes à la suite l'une de l'autre de toute façon. De plus, la liste vide se comporte bien comme un élément neutre, donc tout va bien. Remarquez que les monoïdes ne nécessitent pas que `a `mappend` b` soit égal à `b `mappend` a`. Dans le cas des listes, clairement ce n'est pas le cas.

```
ghci> "one" `mappend` "two"
"onetwo"
ghci> "two" `mappend` "one"
"twoone"
```

Et ce n'est pas un problème. Le fait que pour la multiplication, `3 * 5` et `5 * 3` soient identiques n'est qu'une propriété de la multiplication, mais n'a pas à être vrai pour tous les monoïdes (et ne l'est pas pour la plupart d'ailleurs).

`Product` et `Sum`

On a déjà examiné une façon pour les nombres d'être considérés comme des monoïdes. La fonction binaire est `*` et l'élément neutre est `1`. Il s'avère que ce n'est pas la seule façon pour les nombres de former un monoïde. Une autre façon consiste à prendre pour fonction binaire `+` et pour élément neutre `0`.

```
ghci> 0 + 4
4
ghci> 5 + 0
5
ghci> (1 + 3) + 5
9
ghci> 1 + (3 + 5)
9
```

Les lois des monoïdes tiennent, parce qu'ajouter 0 à tout nombre le laisse inchangé. Et l'addition est associative, donc pas de problème. À présent qu'il y a deux manières toute aussi valide l'une que l'autre pour les nombres d'être des monoïdes, quelle voie choisir ? Eh bien, pas besoin de choisir. Souvenez-vous, quand il y a plusieurs façons pour un type d'être une instance de la même classe de types, on peut l'envelopper dans un *newtype* et faire de ce nouveau type une autre instance de la même classe de types. On a le beurre et l'argent du beurre.

Le module `Data.Monoid` exporte deux types pour cela, `Product` et `Sum`. `Product` est défini comme :

```
newtype Product a =  Product { getProduct :: a }
    deriving (Eq, Ord, Read, Show, Bounded)
```

Simple, un enrobage avec *newtype*, avec un paramètre de type et quelques instance dérivées. Son instance de `Monoid` ressemble à :

```
instance Num a => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

`mempty` est simplement `1` enveloppé dans un constructeur `Product`. `mappend` filtre par motif sur le constructeur `Product`, multiplie les deux nombres, et enveloppe à nouveau le résultat. Comme vous le voyez, il y a une contrainte de classe `Num a`. Cela veut dire que `Product a` est une instance de `Monoid` pour tout `a` qui est une instance de `Num`. Pour utiliser `Product a` en tant que monoïde, il faut faire un peu d'emballage et de déballage.

```
ghci> getProduct $ Product 3 `mappend` Product 9
27
ghci> getProduct $ Product 3 `mappend` mempty
3
ghci> getProduct $ Product 3 `mappend` Product 4 `mappend` Product 2
```

```
24
ghci> getProduct . mconcat . map Product $ [3,4,2]
24
```

C'est pratique pour démontrer le comportement de la classe de types `Monoid`, mais personne de bien sensé n'irait multiplier des nombres de cette façon plutôt que d'écrire simplement `3 * 9` et `3 * 1`. Mais un peu plus tard, nous verrons comment ces instances de `Monoid` qui peuvent sembler triviales s'avèrent pratiques.

`Sum` est défini comme `Product` et l'instance est également similaire. On l'utilise ainsi :

```
ghci> getSum $ Sum 2 `mappend` Sum 9
11
ghci> getSum $ mempty `mappend` Sum 3
3
ghci> getSum . mconcat . map Sum $ [1,2,3]
6
```

`Any` et *`All`*

Un autre type qui peut agir comme un monoïde de deux façons distinctes et légitimes est `Bool`. La première façon utilise la fonction *ou* `||` comme fonction binaire, avec `False` pour élément neutre. En logique, *ou* est `True` si n'importe lequel de ses paramètres est `True`, sinon c'est `False`. Ainsi, en utilisant `False` comme élément neutre, on retourne bien `False` en faisant *ou* avec `False`, et `True` en faisant *ou* avec `True`. Le *newtype* `Any` est une instance de monoïde de cette manière. Il est défini ainsi :

```
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Ord, Read, Show, Bounded)
```

Quant à l'instance :

```
instance Monoid Any where
        mempty = Any False
        Any x `mappend` Any y = Any (x || y)
```

Il s'appelle `Any` parce que `x `mappend` y` sera `True` si n'importe laquelle (NDT : en anglais, *any*) de ses valeurs est `True`. Même si trois ou plus `Bool` enveloppés dans `Any` sont `mappend` ensemble, le résultat sera toujours `True` lorsque n'importe lequel d'entre eux est `True` :

```
ghci> getAny $ Any True `mappend` Any False
True
ghci> getAny $ mempty `mappend` Any True
True
ghci> getAny . mconcat . map Any $ [False, False, False, True]
True
ghci> getAny $ mempty `mappend` mempty
False
```

L'autre façon de faire une instance de `Monoid` pour `Bool` est un peu l'opposé : utiliser `&&` comme fonction binaire et `True` comme élément neutre. Le *et* logique retourne `True` si ses deux paramètres valent `True`. Voici la déclaration *newtype*, rien d'extraordinaire :

```
newtype All = All { getAll :: Bool }
        deriving (Eq, Ord, Read, Show, Bounded)
```

Et l'instance :

```
instance Monoid All where
        mempty = All True
        All x `mappend` All y = All (x && y)
```

Quand on `mappend` des valeurs de type `All`, le résultat sera `True` seulement si toutes (NDT : en anglais, *all*), les valeurs passées à `mappend` sont `True` :

```
ghci> getAll $ mempty `mappend` All True
True
ghci> getAll $ mempty `mappend` All False
False
ghci> getAll . mconcat . map All $ [True, True, True]
True
ghci> getAll . mconcat . map All $ [True, True, False]
```

```
False
```

Tout comme avec la multiplication et l'addition, on utilise généralement les fonctions binaires directement plutôt que d'envelopper les valeurs dans des *newtype* pour ensuite utiliser `mappend` et `mempty`. `mconcat` semble utile pour `Any` et `All` mais généralement, il est plus simple d'utiliser les fonctions `or` et `and` qui prennent une liste de `Bool` et retourne `True` si, respectivement, l'une d'elles est `True` ou toutes sont `True`.

Le monoïde `Ordering`

Hé, vous vous souvenez du type `Ordering` ? C'est le type retourné lorsqu'on compare des choses, qui peut avoir trois valeurs : `LT`, `EQ` et `GT`, qui signifient *inférieur*, *égal* et *supérieur* respectivement :

```
ghci> 1 `compare` 2
LT
ghci> 2 `compare` 2
EQ
ghci> 3 `compare` 2
GT
```

Avec les listes, les nombres et les valeurs booléennes, on voit que trouver des monoïdes est juste l'affaire de regarder ce qu'on utilisait déjà afin de voir si ces choses présentent un comportement monoïdal. Pour `Ordering`, il faut y regarder à deux fois avant de reconnaître le monoïde, mais il s'avère que son instance de `Monoid` est tout aussi intuitive que celles recontrées précédemment, et plutôt utile également :

```
instance Monoid Ordering where
    mempty = EQ
    LT `mappend` _ = LT
    EQ `mappend` y = y
    GT `mappend` _ = GT
```

L'instance est créée de la sorte : quand on `mappend` deux valeurs `Ordering`, celle de gauche est préservée, à moins que la valeur à gauche soit `EQ`, auquel cas celle de droite est le résultat. Le neutre est `EQ`. Au départ, ça peut sembler arbitraire, mais cela ressemble en fait beaucoup à la façon donc on compare lexicographiquement des mots. On compare d'abord les premières lettres de chaque mot, et si elles diffèrent, on peut immédiatement décider de l'ordre des mots dans un dictionnaire. Cependant, si elles sont égales, alors on doit comparer la prochaine paire de lettres et répéter le procédé.

Par exemple, si l'on voulait comparer lexicographiquement les mots "ox" et "on", on pourrait commencer par comparer la première lettre de chaque mot, voir qu'elles sont égales, puis comparer la seconde lettre de chaque mot. On voit que `'x'` est lexicographiquement supérieure à `'n'`, et on peut donc savoir l'ordre des deux mots. Pour se faire une intuition sur le fait qu'`EQ` soit l'identité, on peut remarquer que si l'on glissait la même lettre à la même position dans les deux mots, leur ordre lexicographique ne changerait pas. `"oix"` est toujours lexicographiquement supérieur à `"oin"`.

Il est important de noter que dans l'instance de `Monoid` pour `Ordering`, `x `mappend` y` n'est pas égal à `y `mappend` x`. Puisque le premier paramètre est conservé à moins d'être `EQ`, `LT `mappend` GT` retourne `LT`, alors que `GT `mappend` LT` retournera `GT` :

```
ghci> LT `mappend` GT
LT
ghci> GT `mappend` LT
GT
ghci> mempty `mappend` LT
LT
ghci> mempty `mappend` GT
GT
```

OK, comment est-ce que ce monoïde est utile ? Mettons que vous soiyez en train d'écrire une fonction qui prend deux chaînes de caractères, compare leur longueur, et retourne un `Ordering`. Mais si les chaînes sont de même longueur, au lieu de retourner `EQ` immédiatement, vous voulez les comparer lexicographiquement. Une façon de faire serait :

```
lengthCompare :: String -> String -> Ordering
lengthCompare x y = let a = length x `compare` length y
                        b = x `compare` y
                    in  if a == EQ then b else a
```

On appelle `a` le résultat de la comparaison des longueurs, et `b` celui de la comparaison lexicographique, et si les longueurs s'avèrent égales, on retourne l'ordre

lexicographique.

Mais, en mettant à profit notre compréhension d'`Ordering` comme un monoïde, on peut réécrire cette fonction d'une manière bien plus simple :

```
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`
                    (x `compare` y)
```

On peut essayer :

```
ghci> lengthCompare "zen" "ants"
LT
ghci> lengthCompare "zen" "ant"
GT
```

Souvenez-vous, quand on fait `mappend`, le paramètre de gauche est toujours gardé à moins d'être `EQ`, auquel cas celui de droite est gardé. C'est pourquoi on a placé la comparaison qu'on considère comme le critère le plus important en premier paramètre. Si l'on voulait étendre la fonction pour comparer le nombre de voyelles comme deuxième critère en importance, on pourrait modifier la fonction ainsi :

```
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`
                    (vowels x `compare` vowels y) `mappend`
                    (x `compare` y)
    where vowels = length . filter (`elem` "aeiou")
```

On a fait une fonction auxiliaire qui prend une chaîne de caractères et nous dit combien de voyelles elle contient en filtrant seulement les lettres qui sont dans la chaîne `"aeiou"`, puis en appliquant `length` au résultat filtré.

```
ghci> lengthCompare "zen" "anna"
LT
ghci> lengthCompare "zen" "ana"
LT
ghci> lengthCompare "zen" "ann"
GT
```

Très cool. Ici, on voit comme dans le premier exemple les longueurs sont différentes et donc `LT` est retourné, parce que `"zen"` est moins longue qu'`"anna"`. Dans le deuxième exemple, les longueurs sont égales, mais la deuxième chaîne a plus de voyelles, donc `LT` est retourné à nouveau. Dans le troisième exemple, elles ont la même longueur et le même nombre de voyelles, donc elles sont comparées lexicographiquement et `"zen"` gagne.

Le monoïde `Ordering` est très cool parce qu'il permet de comparer facilement des choses selon plusieurs critères en plaçant ces critères dans l'ordre du plus important au moins important.

`Maybe` le monoïde

Regardons les différentes façons de faire de `Maybe` une instance de `Monoid`, et comment ces instances sont utiles.

Une façon est de traiter `Maybe` comme un monoïde seulement lorsque son paramètre de type `a` est un monoïde également, et d'implémenter `mappend` de façon à ce qu'il utilise l'opération `mappend` des valeurs enveloppées dans le `Just`. `Nothing` est l'élément neutre, et ainsi si l'une des deux valeurs qu'on `mappend` est `Nothing`, on conserve l'autre valeur. Voici la déclaration d'instance :

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

Remarquez la contrainte de classe. Elle dit que `Maybe a` est une instance de `Monoid` seulement si `a` est une instance de `Monoid`. Si l'on `mappend` quelque chose avec `Nothing`, le résultat est cette chose. Si on `mappend` deux valeurs `Just`, le contenu des `Just` est `mappend` et est enveloppé de nouveau dans un `Just`. On peut faire cela parce que la contrainte de classe nous garantit que le type à l'intérieur du `Just` est une instance de `Monoid`.

```
ghci> Nothing `mappend` Just "andy"
Just "andy"
ghci> Just LT `mappend` Nothing
```

```
Just LT
ghci> Just (Sum 3) `mappend` Just (Sum 4)
Just (Sum {getSum = 7})
```

Cela s'avère utile lorsque vous utilisez des monoïdes comme résultats de calculs pouvant échouer. Grâce à cette instance, on n'a pas besoin de vérifier si les calculs ont échoué en regardant si les valeurs sont `Nothing` ou `Just`, on peut simplement continuer à les traiter comme des monoïdes ordinaires.

Mais, et si le type du contenu de `Maybe` n'est pas une instance de `Monoid` ? Remarquez que dans la déclaration d'instance précédente, on ne se repose sur le fait que le contenu soit un monoïde que lorsque `mappend` est appliqué à deux valeurs `Just`. Mais si l'on ne sait pas si le contenu est un monoïde, on ne peut pas faire `mappend` sur ces valeurs, donc que faire ? Eh bien, on peut par exemple toujours jeter la deuxième valeur et garder la première. Pour cela, le type `First a` existe, et voici sa définition :

```
newtype First a = First { getFirst :: Maybe a }
    deriving (Eq, Ord, Read, Show)
```

On prend un `Maybe a` et on l'enveloppe avec un *newtype*. L'instance de `Monoid` est comme suit :

```
instance Monoid (First a) where
    mempty = First Nothing
    First (Just x) `mappend` _ = First (Just x)
    First Nothing `mappend` x = x
```

Comme on l'a dit. `mempty` est `Nothing` emballé dans un constructeur de *newtype* `First`. Si le premier paramètre de `mappend` est une valeur `Just`, on ignore le second. Si le premier est `Nothing`, alors on retourne l'autre, qu'il soit un `Just` ou un `Nothing`.

```
ghci> getFirst $ First (Just 'a') `mappend` First (Just 'b')
Just 'a'
ghci> getFirst $ First Nothing `mappend` First (Just 'b')
Just 'b'
ghci> getFirst $ First (Just 'a') `mappend` First Nothing
Just 'a'
```

`First` est utile quand on a plein de valeurs `Maybe` et qu'on souhaite juste savoir s'il y a un `Just` dans le tas. La fonction `mconcat` s'avère utile :

```
ghci> getFirst . mconcat . map First $ [Nothing, Just 9, Just 10]
Just 9
```

Si l'on veut un monoïde sur `Maybe a` de manière à ce que le deuxième paramètre soit gardé lorsque les deux paramètres de `mappend` sont des valeurs `Just`, `Data.Monoid` fournit un type `Last a`, qui fonctionne comme `First a`, à l'exception que c'est la dernière valeur différente de `Nothing` qui est gardée par `mappend` et `mconcat` :

```
ghci> getLast . mconcat . map Last $ [Nothing, Just 9, Just 10]
Just 10
ghci> getLast $ Last (Just "one") `mappend` Last (Just "two")
Just "two"
```

Utiliser des monoïdes pour plier des structures de données

Une des façons les plus intéressantes de mettre les monoïdes à l'usage est de les utiliser pour nous aider à définir des plis sur diverses structures de données. Jusqu'ici, nous n'avons fait que des plis sur des listes, mais les listes ne sont pas les seules structures de données pliables. On peut définir des plis sur presque toute structure de données. Les arbres se prêtent particulièrement bien à l'exercice du pli.

Puisqu'il y a tellement de structures de données qui fonctionnent bien avec les plis, la classe de types `Foldable` a été introduite. Comme `Functor` est pour les choses sur lequelles on peut mapper, `Foldable` est pour les choses qui peuvent être pliées ! Elle est trouvable dans `Data.Foldable` et puisqu'elle exporte des fonctions dont les noms sont en collision avec ceux du `Prelude`, elle est préférablement importée qualifiée (et servie avec du basilic) :

```
import qualified Foldable as F
```

Pour nous éviter de précieuses frappes de clavier, on l'importe qualifiée par `F`. Bien, donc quelles sont les fonctions que cette classe de types définit ? Eh bien, parmi celles-ci sont `foldr`, `foldl`, `foldr1`, `foldl1`. Hein ? Mais on connaît déjà ces fonctions, quoi de neuf ? Comparons le type de `foldr` dans `Foldable` avec celui de `foldr` du `Prelude` pour voir ce qui diffère :

```
ghci> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
ghci> :t F.foldr
F.foldr :: (F.Foldable t) => (a -> b -> b) -> b -> t a -> b
```

Ah ! Donc, alors que `foldr` prend une liste et la plie, le `foldr` de `Data.Foldable` accepte tout type qui peut être plié, pas seulement les listes ! Comme on peut s'y attendre, les deux fonctions font la même chose sur les listes :

```
ghci> foldr (*) 1 [1,2,3]
6
ghci> F.foldr (*) 1 [1,2,3]
6
```

Ok, quelles autres structures peuvent être pliées ? Eh bien, le `Maybe` qu'on connaît tous et qu'on aime tant !

```
ghci> F.foldl (+) 2 (Just 9)
11
ghci> F.foldr (||) False (Just True)
True
```

Mais plier une simple valeur `Maybe` n'est pas très intéressant, parce que quand il s'agit de se plier, elle se comporte comme une liste avec un élément si c'est un `Just`, et comme une liste vide si c'est `Nothing`. Examinons donc une structure de données un peu plus complexe.

Vous rappelez-vous la structure de données arborescente du chapitre Créer nos propres types et classes de types ? On l'avait définie ainsi :

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

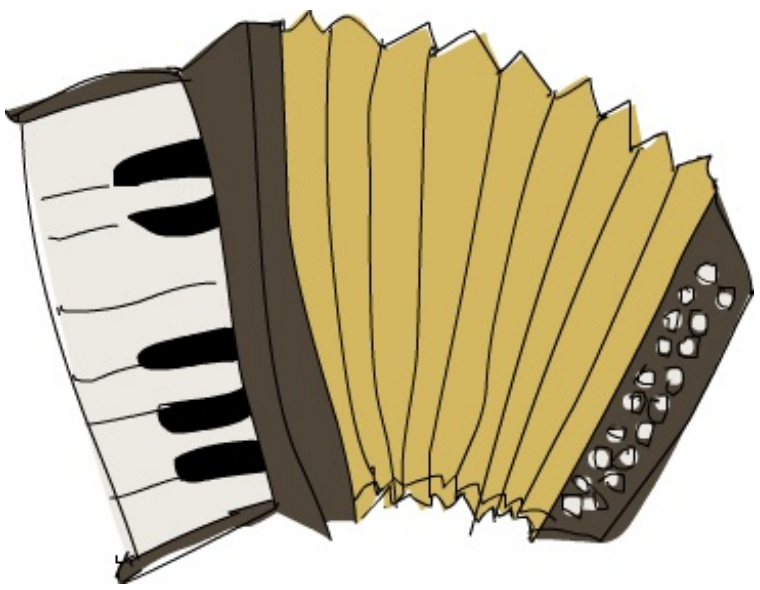
On disait qu'un arbre était soit un arbre vide qui ne contient aucune valeur, soit un neud qui contient une valeur ainsi que deux autres arbres. Après l'avoir défini, on en a fait une instance de `Functor` et on a gagné la possibilité de faire `fmap` sur ce type. À présent, on va en faire une instance de `Foldable` afin de pouvoir le plier. Une façon de faire d'un constructeur de types une instance de `Foldable` consiste à implémenter directement `foldr`. Une autre façon, souvent bien plus simple, consiste à implémenter la fonction `foldMap`, qui fait aussi partie de la classe de types `Foldable`. `foldMap` a pour type :

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

Son premier paramètre est une fonction qui prend une valeur du type que notre structure de données pliable contient (ici dénoté `a`) et retourne une valeur monoïdale. Son second paramètre est une structure pliable contenant des valeurs de type `a`. Elle mappe la fonction sur la structure pliable, produisant ainsi une structure pliable contenant des valeurs monoïdales. Ensuite, en faisant `mappend` entre toutes ces valeurs monoïdales, elle les joint en une unique valeur monoïdale. Cette fonction peut paraître bizarre pour l'instant, mais on va voir qu'elle est très simple à implémenter. Ce qui est aussi cool, c'est qu'il suffit simplement d'implémenter cette fonction pour que notre type soit une instance de `Foldable`. Ainsi, si l'on implémente `foldMap` pour un type, on obtient `foldr` et `foldl` sans effort !

Voici comment un `Tree` est fait instance de `Foldable` :

```
instance F.Foldable Tree where
    foldMap f Empty = mempty
    foldMap f (Node x l r) = F.foldMap f l `mappend`
                             f x           `mappend`
                             F.foldMap f r
```



On pense ainsi : si l'on nous donne une fonction qui prend un élément de notre arbre et retourne une valeur monoïdale, comment réduit-on notre arbre à une simple valeur monoïdale ? Quand nous faisions `fmap` sur notre arbre, on appliquait la fonction mappée au nœud, puis on mappait récursivement la fonction sur le sous-arbre gauche et sur le sous-arbre droit. Ici, on nous demande non seulement de mapper la fonction, mais également de joindre les résultats en une simple valeur monoïdale à l'aide de `mappend`. On considère d'abord le cas de l'arbre vide - un pauvre arbre tout triste et solitaire, sans valeur ni sous-arbre. Il n'a pas de valeur qu'on puisse passer à notre fonction créant des monoïdes, donc si notre arbre est vide, la valeur monoïdale sera `mempty`.

Le cas d'un nœud non vide est un peu plus intéressant. Il contient deux sous-arbres ainsi qu'une valeur. Dans ce cas, on `foldMap` récursivement la même fonction `f` sur les sous-arbres gauche et droit. Souvenez-vous, notre `foldMap` retourne une simple valeur monoïdale. On applique également la fonction `f` à la valeur du nœud. À présent, nous avons trois valeurs monoïdales (deux venant des sous-arbres et une venant de l'application de `f` sur la valeur du nœud) et il suffit de les écraser en une seule valeur. Pour ce faire, on utilise `mappend`, et naturellement, le sous-arbre gauche vient avant la valeur du nœud, suivi du sous-arbre droit.

Remarquez qu'on n'a pas eu besoin d'écrire la fonction qui prend une valeur et retourne une valeur monoïdale. On la reçoit en paramètre de la fonction `foldMap` et tout ce qu'on a besoin de décider est où l'appliquer et comment joindre les monoïdes qui en résultent.

Maintenant qu'on a une instance de `Foldable` pour notre type arborescent, on a `foldr` et `foldl` gratuitement ! Considérez cet arbre :

```
testTree = Node 5
            (Node 3
                (Node 1 Empty Empty)
                (Node 6 Empty Empty)
            )
            (Node 9
                (Node 8 Empty Empty)
                (Node 10 Empty Empty)
            )
```

Il a `5` pour racine, puis son nœud gauche contient `3` avec `1` à sa gauche et `6` à sa droite. Le nœud droit de la racine contient `9` avec un `8` à sa gauche et un `10` tout à droite. Avec une instance de `Foldable`, on peut faire tous les plis qu'on fait sur des listes :

```
ghci> F.foldl (+) 0 testTree
42
ghci> F.foldl (*) 1 testTree
64800
```

`foldMap` ne sert pas qu'à créer les instances de `Foldable`, elle sert également à réduire une structure à une simple valeur monoïdale. Par exemple, si l'on voulait savoir si n'importe quel nombre de notre arbre est égal à `3`, on pourrait faire :

```
ghci> getAny $ F.foldMap (\x -> Any $ x == 3) testTree
True
```

Ici, `\x -> Any $ x == 3` est une fonction qui prend un nombre et retourne une valeur monoïdale, dans ce cas un `Bool` enveloppé en `Any`. `foldMap` applique la fonction à chaque élément de l'arbre, puis réduit les monoïdes résultants en une unique valeur monoïdale à l'aide de `mappend`. Si l'on faisait :

```
ghci> getAny $ F.foldMap (\x -> Any $ x > 15) testTree
False
```

Tous les nœuds de notre arbre contiendraient la valeur `Any False` après avoir appliqué la fonction dans la lambda à chacun d'eux. Pour valoir `True`, `mappend` sur `Any` doit avoir au moins un `True` passé en paramètre. C'est pourquoi le résultat final est `False`, ce qui est logique puisqu'aucune valeur de notre arbre n'excède `15`.

On peut aussi facilement transformer notre arbre en une liste en faisant `foldMap` avec la fonction `\x -> []`. En projetant d'abord la fonction sur l'arbre, chaque élément devient une liste singleton. Puis, `mappend` a lieu entre tous ces singletons et retourne une unique liste qui contient tous les éléments de notre arbre :

```
ghci> F.foldMap (\x -> [x]) testTree
[1,3,6,5,8,9,10]
```

Ce qui est vraiment cool, c'est que toutes ces techniques ne sont pas limitées aux arbres, elles fonctionnent pour toute instance de `Foldable`.

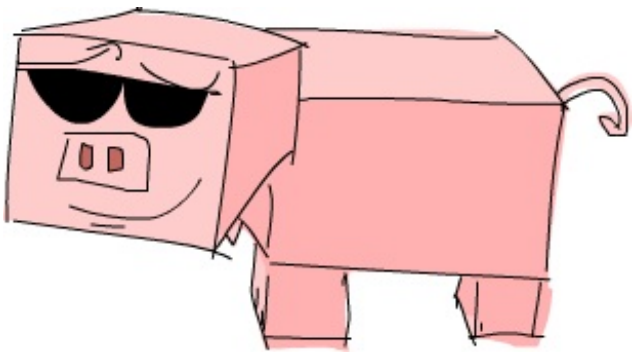
← Résoudre des problèmes fonctionnellement | Table des matières | Pour une poignée de monades →

Pour une poignée de monades

⬅ Foncteurs, foncteurs applicatifs et monoïdes | Table des matières | Et pour quelques monades de plus ➡

Quand on a parlé pour la première fois des foncteurs, on a vu que c'était un concept utile pour les valeurs sur lesquelles on pouvait mapper. Puis, on a poussé ce concept un cran plus loin en introduisant les foncteurs applicatifs, qui nous permettent de voir les valeurs de certains types de données comme des valeurs dans des contextes et d'utiliser des fonctions normales sur ces valeurs tout en préservant la sémantique de ces contextes.

Dans ce chapitre, nous allons découvrir les monades, qui sont juste des foncteurs applicatifs gonflés aux hormones, tout comme les foncteurs applicatifs étaient juste des foncteurs gonflés aux hormones.



Quand on a débuté les foncteurs, on a vu qu'il était possible de mapper des fonctions sur divers types de données. On a vu qu'à cet effet, la classe de types `Functor` avait été introduite, et cela a soulevé la question : quand on a une fonction de type `a -> b` et un type de données `f a`, comment mapper cette fonction sur le type de données pour obtenir un `f b` ? On a vu comment mapper quelque chose sur un `Maybe a`, une liste `[a]`, une action I/O `IO a`, etc. On a même vu comment mapper une fonction `a -> b` sur d'autres fonctions de type `r -> a` pour obtenir des fonctions `r -> b`. Pour répondre à la question de comment mapper une fonction sur un type de données, tout ce qu'il nous a fallu considérer était le type de `fmap` :

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

Puis, à le faire marcher pour notre type de données en écrivant l'instance de `Functor` appropriée.

Puis, on a vu une amélioration possible des foncteurs, et nous nous sommes dit, hé, et si cette fonction `a -> b` était déjà enveloppée dans une valeur fonctorielle ? Comme par exemple, si l'on avait `Just (*3)`, comment appliquer cela à `Just 5` ? Et si l'on voulait l'appliquer non pas à `Just 5`, mais plutôt à `Nothing` ? Ou si l'on avait `[(*2), (+4)]`, comment l'appliquerions-nous à `[1, 2, 3]` ? Comment cela marcherait-il ? Pour cela, la classe de types `Applicative` est introduite, afin de répondre au problème à l'aide du type :

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

On a aussi vu que l'on pouvait prendre une valeur normale et l'envelopper dans un type de données. Par exemple, on peut prendre un `1` et l'envelopper de manière à ce qu'il devienne un `Just 1`. On en faire un `[1]`. Ou une action I/O qui ne fait rien, mais retourne `1`. La fonction qui fait cela s'appelle `pure`.

Comme on l'a dit, une valeur applicative peut être vue comme une valeur avec un contexte. Une valeur spéciale en gros. Par exemple, le caractère `'a'` est un caractère normal, alors que `Just 'a'` a un contexte additionnel. Au lieu d'être un `Char`, c'est un `Maybe Char`, ce qui nous indique que sa valeur peut être un caractère, mais qu'il aurait aussi pu être absent.

La classe de types `Applicative` nous permettait d'utiliser des fonctions normales sur des valeurs dans des contextes tout en préservant le contexte de façon très propre. Observez :

```
ghci> (*) <$> Just 2 <*> Just 8
Just 16
ghci> (++) <$> Just "klingon" <*> Nothing
Nothing
ghci> (-) <$> [3,4] <*> [1,2,3]
[2,1,0,3,2,1]
```

Ah, cool, à présent qu'on les traite comme des valeurs applicatives, les valeurs `Maybe a` représentent des calculs qui peuvent avoir échoué, les valeurs `[a]` représentent des calculs qui ont plusieurs résultats (calculs non déterministes), les `IO a` représentent des valeurs qui ont des effets de bord, etc.

Les monades sont une extension naturelle des foncteurs applicatifs, et avec elles on cherche à résoudre ceci : si vous avez une valeur dans un contexte, `m a`, comment appliquer dessus une fonction qui prend une valeur normale `a` et retourne une valeur dans un contexte ? C'est-à-dire, comment appliquer une fonction de type `a -> m b` à une valeur de type `m a` ? En gros, on veut cette fonction :

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

Si l'on a une valeur spéciale, et une fonction qui prend une valeur normale mais retourne une valeur spéciale, comment donner la valeur spéciale à cette fonction ? C'est la question à laquelle on s'attaquera en étudiant les monades. On écrit `m a` plutôt que `f a` parce que le `m` signifie `Monad`, mais les monades sont seulement des foncteurs applicatifs supportant `(>>=)`. La fonction `(>>=)` est prononcée *bind* (NDT : en français, cela signifie lier).

Quand on a une valeur normale `a` et une fonction normale `a -> b`, il est très facile d'alimenter la fonction avec la valeur - on applique simplement la fonction avec la valeur et voilà. Mais quand on a affaire à des valeurs qui viennent de certains contextes, il faut un peu plus y réfléchir pour voir comment ces valeurs spéciales peuvent être données aux fonctions, et comment prendre en compte leur comportement, mais vous verrez que c'est simple comme bonjour.

Trempons-nous les pieds avec Maybe

Maintenant qu'on a une vague idée de ce que sont les monades, voyons si l'on peut éclaircir cette notion un tant soit peu.

Sans surprise, `Maybe` est une monade, alors explorons cela encore un peu et voyons si l'on peut combiner cela avec ce qu'on sait des monades.

Soyez certain de bien comprendre les foncteurs applicatifs à présent. C'est bien si vous ressentez comment les instances d'`Applicative` fonctionnent et le genre de calculs qu'elles représentent, parce que les monades ne font que prendre nos connaissances des foncteurs applicatifs, et les améliorer.

Une valeur ayant pour type `Maybe a` représente une valeur de type `a` avec le contexte de l'échec potentiel attaché. Une valeur `Just "dharma"` signifie que la chaîne de caractères `"dharma"` est présente, alors qu'une valeur `Nothing` représente son absence, ou si vous imaginez la chaîne de caractères comme le résultat d'un calcul, cela signifie que le calcul a échoué.

Quand on a regardé `Maybe` comme un foncteur, on a vu que si l'on veut `fmap` une fonction sur celui-ci, elle est mappée sur ce qui est à l'intérieur des valeurs `Just`, les `Nothing` étant conservés tels quels parce qu'il n'y a pas de valeur sur laquelle mapper !

Ainsi :

```
ghci> fmap (++"!") (Just "wisdom")
Just "wisdom!"
ghci> fmap (++"!") Nothing
Nothing
```

En tant que foncteur applicatif, cela fonctionne similairement. Cependant, les foncteurs applicatifs prennent une fonction enveloppée. `Maybe` est un foncteur applicatif de manière à ce que lorsqu'on utilise `<*>` pour appliquer une fonction dans un `Maybe` à une valeur dans un `Maybe`, elles doivent toutes deux être des valeurs `Just` pour que le résultat soit une valeur `Just`, autrement le résultat est `Nothing`. C'est logique, puisque s'il vous manque soit la fonction, soit son paramètre, vous ne pouvez pas l'inventer, donc vous devez propager l'échec :

```
ghci> Just (+3) <*> Just 3
Just 6
ghci> Nothing <*> Just "greed"
Nothing
ghci> Just ord <*> Nothing
Nothing
```

Quand on utilise le style applicatif pour appliquer des fonctions normales sur des valeurs `Maybe`, c'est similaire. Toutes les valeurs doivent être des `Just`, sinon le tout est `Nothing` !

```
ghci> max <$> Just 3 <*> Just 6
Just 6
ghci> max <$> Just 3 <*> Nothing
Nothing
```

À présent, pensons à ce que l'on ferait pour faire `>>=` sur `Maybe`. Comme on l'a dit, `>>=` prend une valeur monadique, et une fonction qui prend une valeur normale et retourne une valeur monadique, et parvient à appliquer cette fonction à la valeur monadique. Comment fait-elle cela, si la fonction n'accepte qu'une valeur normale ? Eh bien, pour ce faire, elle doit prendre en compte le contexte de la valeur monadique.

Dans ce cas, `>>=` prendrait un `Maybe a` et une fonction de type `a -> Maybe b` et appliquerait la fonction au `Maybe a`. Pour comprendre comment elle fait cela, on peut utiliser notre intuition venant du fait que `Maybe` est un foncteur applicatif. Mettons qu'on ait une fonction `\x -> Just (x + 1)`. Elle prend un nombre, lui ajoute `1` et l'enveloppe dans un `Just` :

```
ghci> (\x -> Just (x+1)) 1
Just 2
ghci> (\x -> Just (x+1)) 100
Just 101
```

Si on lui donne `1`, elle s'évalue en `Just 2`. Si on lui donne le nombre `100`, le résultat est `Just 101`. Très simple. Voilà l'obstacle : comment donner une valeur `Maybe` a cette fonction ? Si on pense à ce que fait `Maybe` en tant que foncteur applicatif, répondre est plutôt simple. Si on lui donne une valeur `Just`, elle prend ce qui est dans le `Just` et applique la fonction dessus. Si on lui donne un `Nothing`, hmm, eh bien, on a une fonction, mais rien pour l'appliquer. Dans ce cas, faisons juste comme avant et retournons `Nothing`.

Plutôt que de l'appeler `>>=`, appelons-la `applyMaybe` pour l'instant. Elle prend un `Maybe a` et une fonction et retourne un `Maybe b`, parvenant à appliquer la fonction sur le `Maybe a`. Voici le code :

```
applyMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
applyMaybe Nothing f  = Nothing
applyMaybe (Just x) f = f x
```

Ok, à présent jouons un peu. On va l'utiliser en fonction infixe pour que la valeur `Maybe` soit sur le côté gauche et la fonction sur la droite :

```
ghci> Just 3 `applyMaybe` \x -> Just (x+1)
Just 4
ghci> Just "smile" `applyMaybe` \x -> Just (x ++ " :)")
Just "smile :)"
ghci> Nothing `applyMaybe` \x -> Just (x+1)
Nothing
ghci> Nothing `applyMaybe` \x -> Just (x ++ " :)")
Nothing
```

Dans l'exemple ci-dessus, on voit que quand on utilisait `applyMaybe` avec une valeur `Just` et une fonction, la fonction était simplement appliquée dans le `Just`. Quand on essayait de l'utiliser sur un `Nothing`, le résultat était `Nothing`. Et si la fonction retourne `Nothing` ? Voyons :

```
ghci> Just 3 `applyMaybe` \x -> if x > 2 then Just x else Nothing
Just 3
ghci> Just 1 `applyMaybe` \x -> if x > 2 then Just x else Nothing
Nothing
```

Comme prévu. Si la valeur monadique à gauche est `Nothing`, le tout est `Nothing`. Et si la fonction de droite retourne `Nothing`, le résultat est également `Nothing`. C'est très similaire au cas où on utilisait `Maybe` comme foncteur applicatif et on obtenait un `Nothing` en résultat lorsqu'il y avait un `Nothing` quelque part.

Il semblerait que, pour `Maybe`, on ait trouvé un moyen de prendre une valeur spéciale et de la donner à une fonction qui prend une valeur normale et en retourne une spéciale. On l'a fait en gardant à l'esprit qu'une valeur `Maybe` représentait un calcul pouvant échouer.

Vous vous demandez peut-être à quoi bon faire cela ? On pourrait croire que les foncteurs applicatifs sont plus puissants que les monades, puisqu'ils nous permettent de prendre une fonction normale et de la faire opérer sur des valeurs dans des contextes. On verra que les monades peuvent également faire ça, car elles sont des améliorations des foncteurs applicatifs, et qu'elles peuvent également faire des choses cool dont les foncteurs applicatifs sont incapables.

On va revenir à `Maybe` dans une minute, mais d'abord, découvrons la classe des types monadiques.

La classe de types Monad

Tout comme les foncteurs ont la classe de types `Functor` et les foncteurs applicatifs ont la classe de types `Applicative`, les monades viennent avec leur propre classe de types : `Monad` ! Wow, qui l'eut cru ? Voici à quoi ressemble la classe de types :

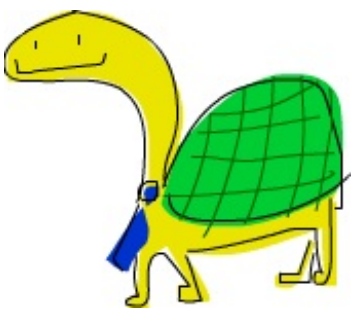
```
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```

Commençons par la première ligne. Elle dit `class Monad m where`. Mais, attendez, n'avons-nous pas dit que les monades étaient des foncteurs applicatifs améliorés ? Ne devrait-il pas y avoir une

contrainte de classe comme `class (Applicative m) => Monad m where` afin qu'un type doive être un foncteur applicatif pour être une monade ? Eh bien, ça devrait être le cas, mais quand Haskell a été créé, il n'est pas apparu à ses créateurs que les foncteurs applicatifs étaient intéressants pour Haskell, et donc ils n'étaient pas présents. Mais soyez rassuré, toute monade est un foncteur applicatif, même si la déclaration de classe `Monad` ne l'indique pas.

La première fonction de la classe de types `Monad` est `return`. C'est la même chose que `pure`, mais avec un autre nom. Son type est `(Monad m) => a -> m a`. Elle prend une valeur et la place dans un contexte minimal contenant cette valeur. En d'autres termes, elle prend une valeur et l'enveloppe dans une monade. Elle est toujours définie comme `pure` de la classe de types `Applicative`, ce qui signifie qu'on la connaît déjà. On a déjà utilisé `return` quand on faisait des entrées-sorties. On s'en servait pour prendre une valeur et créer une I/O factice, qui ne faisait rien de plus que renvoyer la valeur. Pour `Maybe` elle prend une valeur et l'enveloppe dans `Just`.

Un petit rappel : `return` n'est en rien comme le `return` qui existe dans la plupart des langages. Elle ne termine pas l'exécution de la fonction ou quoi que ce soit, elle prend simplement une valeur normale et la place dans un contexte.



La prochaine fonction est `>>=`, ou *bind*. C'est comme une application de fonction, mais au lieu de prendre une valeur normale pour la donner à une fonction normale, elle prend une valeur monadique (dans un contexte) et alimente une fonction qui prend une valeur normale et retourne une valeur monadique.

Ensuite, il y a `>>`. On n'y prêtera pas trop attention pour l'instant parce qu'elle vient avec une implémentation par défaut, et qu'on ne l'implémente presque jamais quand on crée des instances de `Monad`.

La dernière fonction de la classe de types `Monad` est `fail`. On ne l'utilise jamais explicitement dans notre code. En fait, c'est Haskell qui l'utilise pour permettre les échecs dans une construction syntaxique pour les monades que l'on découvrira plus tard. Pas besoin de se préoccuper de `fail` pour l'instant.

À présent qu'on sait à quoi la classe de types `Monad` ressemble, regardons comment `Maybe` est instancié comme `Monad` !

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f  = f x
    fail _ = Nothing
```

`return` est identique à `pure`, pas besoin de réfléchir. On fait comme dans `Applicative` et on enveloppe la valeur dans `Just`.

La fonction `>>=` est identique à notre `applyMaybe`. Quand on lui donne un `Maybe a`, on se souvient du contexte et on retourne `Nothing` si la valeur à gauche est `Nothing`, parce qu'il n'y a pas de valeur sur laquelle appliquer la fonction. Si c'est un `Just`, on prend ce qui est à l'intérieur et on applique `f` dessus.

On peut jouer un peu avec la monade `Maybe` :

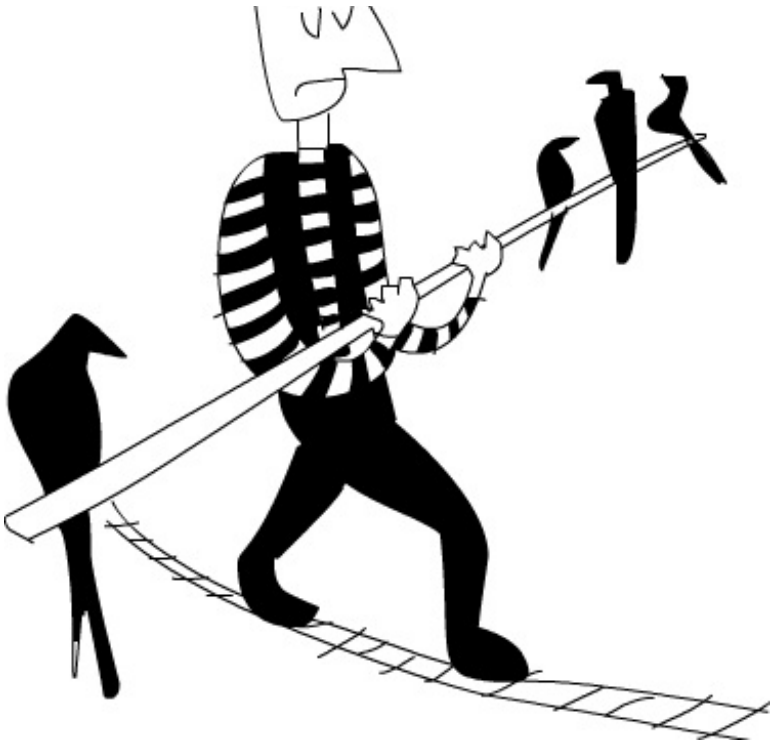
```
ghci> return "WHAT" :: Maybe String
Just "WHAT"
ghci> Just 9 >>= \x -> return (x*10)
Just 90
ghci> Nothing >>= \x -> return (x*10)
Nothing
```

Rien de neuf ou d'excitant à la première ligne puisqu'on a déjà utilisé `pure` avec `Maybe` et on sait que `return` est juste `pure` avec un autre nom. Les deux lignes suivantes présentent un peu mieux `>>=`.

Remarquez comment, lorsqu'on donne `Just 9` à la fonction `\x -> return (x*10)`, le `x` a pris la valeur `9` dans la fonction. Il semble qu'on ait réussi à extraire la valeur du contexte `Maybe` sans avoir recours à du filtrage par motif. Et on n'a pas perdu le contexte de notre valeur `Maybe`, parce que quand elle vaut `Nothing`, le résultat de `>>=` sera également `Nothing`.

Le funambule

À présent qu'on sait comment donner un `Maybe a` à une fonction ayant pour type `a -> Maybe b` tout en tenant compte du contexte de l'échec potentiel, voyons comment on peut utiliser `>>=` de



manière répétée pour gérer le calcul de plusieurs valeurs `Maybe a`.

Pierre a décidé de faire une pause dans son emploi au centre de pisciculture pour s'adonner au funanbulisme. Il n'est pas trop mauvais, mais il a un problème : des oiseaux n'arrêtent pas d'atterrir sur sa perche d'équilibre ! Ils viennent se reposer un instant, discuter avec leurs amis aviaires, avant de décoller en quête de miettes de pain. Cela ne le dérangerait pas tant si le nombre d'oiseaux sur le côté gauche de la perche était égal à celui sur le côté droit. Mais parfois, tous les oiseaux décident qu'ils préfèrent le même côté, et ils détruisent son équilibre, l'entraînant dans une chute embarrassante (il a un filet de sécurité).

Disons qu'il arrive à tenir en équilibre tant que le nombre d'oiseaux sur les côtés gauche et droit ne s'éloignent pas de plus de trois. Ainsi, s'il y a un oiseau à droite et quatre oiseaux à gauche, tout va bien. Mais si un cinquième oiseau atterrit sur sa gauche, il perd son équilibre et plonge malgré lui.

Nous allons simuler des oiseaux atterrissant et décollant de la perche et voir si Pierre tient toujours après un certain nombre d'arrivées et de départs. Par exemple, on souhaite savoir ce qui arrive à Pierre si le premier oiseau arrive sur sa gauche, puis quatre oiseaux débarquent sur sa droite, et enfin l'oiseau sur sa gauche décide de s'envoler.

On peut représenter la perche comme une simple paire d'entiers. La première composante compte les oiseaux sur sa gauche, la seconde ceux sur sa droite :

```
type Birds = Int
type Pole = (Birds,Birds)
```

Tout d'abord, on crée un synonyme du type `Int`, appelé `Birds`, parce qu'on va utiliser des entiers pour représenter un nombre d'oiseaux. Puis, on crée un synonyme pour `(Birds, Birds)` qu'on appelle `Pole` (à ne pas confondre avec une personne d'origine polonaise).

Ensuite, pourquoi ne pas créer une fonction qui prenne un nombre d'oiseaux et les fait atterrir sur un côté de la perche ? Voici les fonctions :

```
landLeft :: Birds -> Pole -> Pole
landLeft n (left,right) = (left + n,right)

landRight :: Birds -> Pole -> Pole
landRight n (left,right) = (left,right + n)
```

Simple. Essayons-les :

```
ghci> landLeft 2 (0,0)
(2,0)
ghci> landRight 1 (1,2)
(1,3)
ghci> landRight (-1) (1,2)
(1,1)
```

Pour faire décoller les oiseaux, on a utilisé un nombre négatif. Puisque faire atterrir des oiseaux sur un `Pole` retourne un `Pole`, on peut chaîner les applications de `landLeft` et `landRight` :

```
ghci> landLeft 2 (landRight 1 (landLeft 1 (0,0)))
(3,1)
```

Quand on applique la fonction `landLeft 1` sur `(0, 0)`, on obtient `(1, 0)`. Puis, on fait atterrir un oiseau sur le côté droit, ce qui donne `(1, 1)`. Finalement, deux oiseaux atterrissent à gauche, donnant `(3, 1)`. On applique une fonction en écrivant d'abord la fonction puis le paramètre, mais ici, il serait plus pratique d'avoir la perche en première et ensuite la fonction d'atterrissage. Si l'on crée une fonction :

```
x -: f = f x
```

On peut appliquer les fonctions en écrivant d'abord le paramètre puis la fonction :

```
ghci> 100 -: (*3)
300
ghci> True -: not
False
ghci> (0,0) -: landLeft 2
(2,0)
```

Ainsi, on peut faire atterrir de façon répétée des oiseaux, de manière plus lisible :

```
ghci> (0,0) -: landLeft 1 -: landRight 1 -: landLeft 2
(3,1)
```

Plutôt cool ! Cet exemple est équivalent au précédent où l'on faisait atterrir plusieurs fois des oiseaux sur la perche, mais il a l'air plus propre. Ici, il est plus facile de voir qu'on commence avec `(0, 0)` et qu'on fait atterrir un oiseau à gauche, puis un à droite, puis deux à gauche.

Jusqu'ici, tout est bien. Mais que se passe-t-il lorsque 10 oiseaux atterrissent du même côté ?

```
ghci> landLeft 10 (0,3)
(10,3)
```

10 oiseaux à gauche et seulement 3 à droite ? Avec ça, le pauvre Pierre est sûr de se casser la figure ! Ici, c'est facile de s'en rendre compte, mais si l'on avait une séquence comme :

```
ghci> (0,0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2)
(0,2)
```

On pourrait croire que tout va bien, mais si vous suivez attentivement les étapes, vous verrez qu'à un moment il y a 4 oiseaux sur la droite et aucun à gauche ! Pour régler ce problème, il faut retourner à nos fonctions `landLeft` et `landRight`. De ce qu'on voit, on veut que ces fonctions puissent échouer. C'est-à-dire, on souhaite qu'elles retournent une nouvelle perche tant que l'équilibre est respecté, mais qu'elles échouent si les oiseaux atterrissent en disproportion. Et quel meilleur moyen d'ajouter un contexte de possible échec que d'utiliser une valeur `Maybe` ? Reprenons ces fonctions :

```
landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left,right)
    | abs ((left + n) - right) < 4 = Just (left + n, right)
    | otherwise                    = Nothing

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left,right)
    | abs (left - (right + n)) < 4 = Just (left, right + n)
    | otherwise                    = Nothing
```

Plutôt que de retourner un `Pole`, ces fonctions retournent un `Maybe Pole`. Elles prennent toujours un nombre d'oiseaux et une perche existante, mais elles vérifient si l'atterrisage des oiseaux déséquilibre ou non Pierre. On a utilisé des gardes pour vérifier si la différence entre les extrémités de la perche est inférieure à 4. Si c'est le cas, on enveloppe la nouvelle perche dans un `Just` et on la retourne. Sinon, on retourne `Nothing` pour indiquer l'échec.

Mettons ces nouvelles-nées à l'épreuve :

```
ghci> landLeft 2 (0,0)
Just (2,0)
ghci> landLeft 10 (0,3)
Nothing
```

Joli ! Quand on fait atterrir des oiseaux sans détruire l'équilibre de Pierre, on obtient une nouvelle perche encapsulée dans un `Just`. Mais quand trop d'oiseaux arrivent d'un côté de la perche, on obtient `Nothing`. C'est cool, mais on dirait qu'on a perdu la capacité de chaîner des atterissages d'oiseaux sur la perche. On ne peut plus faire `landLeft 1 (landRight 1 (0,0))` parce que lorsqu'on applique `landRight 1` à `(0, 0)`, on n'obtient pas un `Pole` mais un `Maybe Pole`. `landLeft 1` prend un `Pole` et retourne un `Maybe Pole`.

On a besoin d'une manière de prendre un `Maybe Pole` et de le donner à une fonction qui prend un `Pole` et retourne un `Maybe Pole`. Heureusement, on a `>>=`, qui fait exactement ça pour `Maybe`. Essayons :

```
ghci> landRight 1 (0,0) >>= landLeft 2
Just (2,1)
```

Souvenez-vous, `landLeft 2` a pour type `Pole -> Maybe Pole`. On ne pouvait pas lui donner directement un `Maybe Pole` résultant de `landRight 1 (0, 0)`, alors on utilise `>>=` pour prendre une valeur dans un contexte et la passer à `landLeft 2`. `>>=` nous permet en effet de traiter la valeur `Maybe` comme une valeur avec un contexte, parce que si l'on donne `Nothing` à `landLeft 2`, on obtient bien `Nothing` et l'échec est propagé :

```
ghci> Nothing >>= landLeft 2
Nothing
```

Ainsi, on peut à présent chaîner des atterrissages pouvant échouer parce que `>>=` nous permet de donner une valeur monadique à une fonction qui attend une

valeur normale.

Voici une séquence d'atterrisages :

```
ghci> return (0,0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

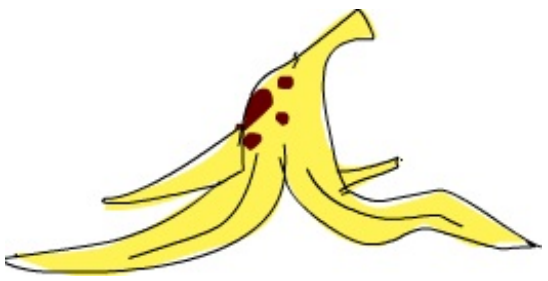
Au début, on utilise `return` pour prendre une perche et l'envelopper dans un `Just`. On aurait pu appeler directement `landRight 2` sur `(0, 0)`, c'était la même chose, mais la notation est plus consistente en utilisant `>>=` pour chaque fonction. `Just (0, 0)` alimente `landRight 2`, résultant en `Just (0, 2)`. À son tour, alimentant `landLeft 2`, résultant en `Just (2, 2)`, et ainsi de suite.

Souvenez-vous de l'exemple précédent où l'on introduisait la notion d'échec dans la routine de Pierre :

```
ghci> (0,0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2)
(0,2)
```

Cela ne simulait pas très bien son interaction avec les oiseaux, parce qu'au milieu, il perdait son équilibre, mais le résultat ne le reflétait pas. Redonnons une chance à cette fonction en utilisant l'application monadique (`>>=`) plutôt que l'application normale.

```
ghci> return (0,0) >>= landLeft 1 >>= landRight 4 >>= landLeft (-1) >>= landRight (-2)
Nothing
```



Génial. Le résultat final représente l'échec, qui est ce qu'on attendait. Regardons comment ce résultat est obtenu. D'abord, `return` place `(0, 0)` dans un contexte par défaut, en faisant un `Just (0, 0)`. Puis, `Just (0,0) >>= landLeft 1` a lieu. Puisque `Just (0, 0)` est une valeur `Just`, `landLeft 1` est appliquée à `(0, 0)`, retournant `Just (1, 0)`, parce que l'oiseau seul ne détruit pas l'équilibre. Ensuite, `Just (1,0) >>= landRight 4` prend place, et le résultat est `Just (1, 4)` car l'équilibre reste intact, bien qu'à sa limite. `Just (1, 4)` est donné à `landLeft (-1)`. Cela signifie que `landLeft (-1) (1, 4)` prend place. Conformément au fonctionnement de `landLeft`, cela retourne `Nothing`, parce que la perche résultante est déséquilibrée. À présent on a un `Nothing` qui est donné à `landRight (-2)`, mais parce que c'est un `Nothing`, le résultat est automatiquement `Nothing`, puisqu'on n'a rien sur quoi appliquer `landRight (-2)`.

On n'aurait pas pu faire ceci en utilisant `Maybe` comme un foncteur applicatif. Si vous essayez, vous vous retrouverez bloqué, parce que les foncteurs applicatifs ne permettent pas que les valeurs applicatives interagissent entre elles. Au mieux, elles peuvent être utilisées en paramètre d'une fonction en utilisant le style applicatif. Les opérateurs applicatifs iront chercher leurs résultats, et les passer à la fonction de façon appropriée en fonction du foncteur applicatif, puis placeront les valeurs applicatives résultantes ensemble, mais il n'y aura que peu d'interaction entre elles. Ici, cependant, chaque étape dépend du résultat de la précédente. À chaque atterrissage, le résultat possible de l'atterrissage précédent est examiné pour voir l'équilibre de la perche. Cela détermine l'échec ou non de l'atterrissage.

On peut aussi créer une fonction qui ignore le nombre d'oiseaux sur la perche et fait tomber Pierre. On peut l'appeler `banana` :

```
banana :: Pole -> Maybe Pole
banana _ = Nothing
```

On peut maintenant la chaîner à nos atterrissages d'oiseaux. Elle causera toujours la chute de notre funambule, parce qu'elle ignore ce qu'on lui passe et retourne un échec. Regardez :

```
ghci> return (0,0) >>= landLeft 1 >>= banana >>= landRight 1
Nothing
```

La valeur `Just (1, 0)` est passée à `banana`, mais elle produit `Nothing`, ce qui force le tout à résulter en `Nothing`. Comme c'est triste !

Plutôt que de créer des fonctions qui ignorent leur entrée et retournent une valeur monadique prédéterminée, on peut utiliser la fonction `>>`, dont l'implémentation par défaut est :

```
(>>) :: (Monad m) => m a -> m b -> m b
m >> n = m >>= \_ -> n
```

Normalement, passer une valeur à une fonction qui ignore son paramètre et retourne tout le temps une valeur prédéterminée résulterait toujours en cette valeur prédéterminée. Avec les monades cependant, leur contexte et signification doit être également considéré. Voici comment `>>` agit sur `Maybe` :

```
ghci> Nothing >> Just 3
```

```
Nothing
ghci> Just 3 >> Just 4
Just 4
ghci> Just 3 >> Nothing
Nothing
```

Si vous remplacez `>>` par `>>= \_ ->`, il est facile de voir pourquoi cela arrive.

On peut remplacer notre fonction `banana` dans la chaîne par un `>>` suivi d'un `Nothing` :

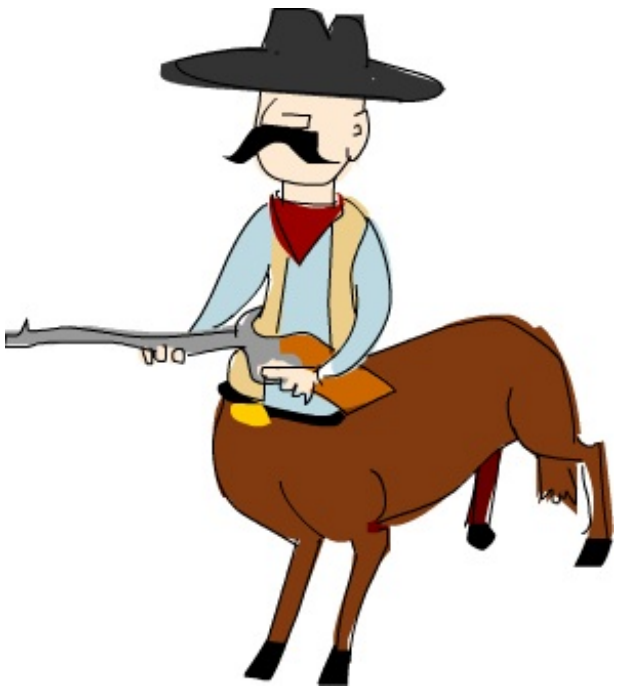
```
ghci> return (0,0) >>= landLeft 1 >> Nothing >>= landRight 1
Nothing
```

Et voilà, échec évident garanti !

Il est aussi intéressant de regarder ce qu'on aurait dû écrire si l'on n'avait pas fait le choix astucieux de traiter les valeurs `Maybe` comme des valeurs dans un contexte pour les donner à nos fonctions. Voici à quoi une série d'atterrissages aurait ressemblé :

```
routine :: Maybe Pole
routine = case landLeft 1 (0,0) of
    Nothing -> Nothing
    Just pole1 -> case landRight 4 pole1 of
        Nothing -> Nothing
        Just pole2 -> case landLeft 2 pole2 of
            Nothing -> Nothing
            Just pole3 -> landLeft 1 pole3
```

On fait atterrir un oiseau à gauche, puis on examine les possibilités d'échec et de succès. Dans le cas d'un échec, on retourne `Nothing`. Dans le cas d'une réussite, on fait atterrir des oiseaux à droite et on répète le tout encore et encore. Convertir cette monstruosité en une jolie chaîne d'applications monadiques à l'aide de `>>=` est un exemple classique prouvant comment la monade `Maybe` nous sauve beaucoup de temps lorsqu'on doit effectuer des calculs successifs qui peuvent échouer.



Remarquez comme l'implémentation de `>>=` pour `Maybe` fait apparaître exactement la logique de vérifier si une valeur est `Nothing`, et quand c'est le cas, de retourner `Nothing` immédiatement, alors que quand ce n'est pas le cas, on continue avec ce qui est dans le `Just`.

Dans cette section, on a pris quelques fonctions qu'on avait et vu qu'elles pouvaient fonctionner encore mieux si les valeurs qu'elles retournaient supportaient la notion d'échec. En transformant ces valeurs en valeurs `Maybe` et en remplaçant l'application normale des fonctions par `>>=`, on a obtenu un mécanisme de propagation d'erreur presque sans effort, parce que `>>=` préserve le contexte des valeurs qu'elle applique aux fonctions. Dans ce cas, le contexte indiquait que nos valeurs pouvaient avoir échoué, et lorsqu'on appliquait nos fonctions sur ces valeurs, la possibilité d'échec était prise en compte.

Notation do

Les monades en Haskell sont tellement utiles qu'elles ont leur propre syntaxe spéciale appelée notation `do`. On a déjà rencontré la notation `do` quand on faisait des entrées-sorties, et on a alors dit qu'elle servait à coller plusieurs actions I/O en une seule. Eh bien, il s'avère que la notation `do` n'est pas seulement pour `IO`, mais peut être utilisée pour n'importe quelle monade. Le principe est identique : coller ensemble plusieurs valeurs monadiques en séquence. Nous allons regarder comment la notation `do` fonctionne et en quoi elle est utile.

Considérez l'exemple familier de l'application monadique :

```
ghci> Just 3 >>= (\x -> Just (show x ++ "!"))
Just "3!"
```

Déjà vu. Donner une valeur monadique à une fonction qui en retourne une autre, rien de grandiose. Remarquez comme en faisant ainsi, `x` devient `3` à l'intérieur de la lambda. Une fois dans la lambda, c'est juste une valeur normale plutôt qu'une valeur monadique. Et si l'on avait encore un `>>=` dans cette fonction ? Regardez :

```
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Just "3!"
```

Ah, une utilisation imbriquée de `>>=` ! Dans la lambda la plus à l'extérieur, on donne `Just "!"` à la lambda `\y -> Just (show x ++ y)`. Dans cette lambda

là, le `y` devient `"!"`, et le `x` est toujours `3` parce qu'on l'obtient de la lambda la plus englobante. Cela me rappelle un peu l'expression suivante :

```
ghci> let x = 3; y = "!" in show x ++ y
"3!"
```

La différence principale entre les deux, c'est que les valeurs dans la première sont monadiques. Ce sont des valeurs dans le contexte de l'échec. On peut remplacer n'importe laquelle d'entre elle par un échec :

```
ghci> Nothing >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Nothing
ghci> Just 3 >>= (\x -> Nothing >>= (\y -> Just (show x ++ y)))
Nothing
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Nothing))
Nothing
```

À la première ligne, donner `Nothing` à la fonction résulte naturellement en `Nothing`. À la deuxième ligne, on donne `Just 3` à une fonction et `x` devient `3`, mais ensuite on donne `Nothing` à la lambda intérieure, et le résultat est `Nothing`, ce qui cause la lamba extérieure à produire `Nothing` à son tour. C'est donc un peu comme assigner des valeurs à des variables dans des expressions *let*, seulement les valeurs sont monadiques.

Pour mieux illustrer ce point, écrivons un script dans lequel chaque `Maybe` prend une ligne :

```
foo :: Maybe String
foo = Just 3   >>= (\x ->
      Just "!" >>= (\y ->
      Just (show x ++ y)))
```

Pour nous éviter d'écrire ces énervantes lambdas successives, Haskell nous offre la notation `do`. Elle nous permet de réécrire le bout de code précédent ainsi :

```
foo :: Maybe String
foo = do
    x <- Just 3
    y <- Just "!"
    Just (show x ++ y)
```

Il semblerait qu'on ait gagné la capacité d'extraire temporairement des valeurs `Maybe` sans avoir à vérifier si elles sont des `Just` ou des `Nothing` à chaque étape. C'est cool ! Si n'importe laquelle des valeurs qu'on essaie d'extraire est `Nothing`, l'expression `do` entière retourne `Nothing`. On attrape les valeurs (potentiellement existantes) et on laisse `>>=` s'occuper du contexte de ces valeurs. Il est important de se rappeler que les expressions `do` sont juste une syntaxe différente pour chaîner les valeurs monadiques.

Dans une expression `do`, chaque ligne est une valeur monadique. Pour inspecter le résultat, on utilise `<-`. Si on a un `Maybe String` et qu'on le lie à une variable par `<-`, cette variable aura pour type `String`, tout comme, lorsqu'on utilisait `>>=` pour passer une valeur monadique à des lambdas. La dernière valeur monadique d'une expression `do`, comme `Just (show x ++ y)` ici, ne peut pas être utilisée avec `<-` pour lier son résultat, parce que cela ne serait pas logique si l'on traduisait à nouveau le `do` en chaînes d'applications avec `>>=`. Plutôt, le résultat de cette dernière expression est le résultat de la valeur monadique entière, prenant en compte l'échec possible des précédentes.

Par exemple, examinez la ligne suivante :

```
ghci> Just 9 >>= (\x -> Just (x > 8))
Just True
```

Puisque le paramètre de gauche de `>>=` est une valeur `Just`, la lambda est appliquée à `9` et le résultat est `Just True`. Si l'on réécrit cela en notation `do`, on obtient :

```
marySue :: Maybe Bool
marySue = do
    x <- Just 9
    Just (x > 8)
```

Si l'on compare les deux, il est facile de comprendre pourquoi le résultat de la valeur monadique complète est celui de la dernière valeur monadique écrite dans l'expression `do`, après avoir chaîné toutes les précédentes.

La routine de notre funambule peut être exprimée avec la notation `do`. `landLeft` et `landRight` prennent un nombre d'oiseaux et une perche et produisent une perche enveloppée dans un `Just`, à moins que le funambule ne glisse, auquel cas `Nothing` est produit. On a utilisé `>>=` pour chaîner les étapes successives parce que chacune d'elles dépendait de la précédente, et chacune ajoutait un contexte d'échec éventuel. Voici deux oiseaux atterrissant à gauche, suivis de deux oiseaux à droite et d'un autre à gauche :

```
routine :: Maybe Pole
routine = do
    start <- return (0,0)
    first <- landLeft 2 start
    second <- landRight 2 first
    landLeft 1 second
```

Voyons si cela réussit :

```
ghci> routine
Just (3,2)
```

Oui ! Bien. Quand nous faisions ces routines explicitement avec `>>=`, on faisait souvent quelque chose comme `return (0, 0) >>= landLeft 2`, parce que `landLeft 2` est une fonction qui retourne une valeur `Maybe`. Avec les expressions `do` cependant, chaque ligne doit comporter une valeur monadique. Il faut donc passer explicitement le `Pole` précédent aux fonctions `landLeft` et `landRight`. Si l'on examinait les variables auxquelles on a lié nos valeurs `Maybe`, `start` vaudrait `(0, 0)`, `first` vaudrait `(2, 0)`, et ainsi de suite.

Puisque les expressions `do` sont écrites ligne par ligne, elles peuvent avoir l'air de code impératif pour certaines personnes. Mais en fait, elles sont seulement séquentielles, puisque chaque valeur de chaque ligne dépend du résultat des précédentes, ainsi que de leur contexte (dans ce cas, savoir si elles ont réussi ou échoué).

Encore une fois, regardons à quoi ressemblerait le code si nous n'avions pas utilisé les aspects monadiques de `Maybe` :

```
routine :: Maybe Pole
routine =
    case Just (0,0) of
        Nothing -> Nothing
        Just start -> case landLeft 2 start of
            Nothing -> Nothing
            Just first -> case landRight 2 first of
                Nothing -> Nothing
                Just second -> landLeft 1 second
```

Voyez comme dans le cas d'un succès, le tuple à l'intérieur de `Just (0, 0)` devient `start`, le résultat de `landLeft 2 start` devient `first`, etc.

Si l'on veut lancer à Pierre une peau de banane avec la notation `do`, on peut le faire ainsi :

```
routine :: Maybe Pole
routine = do
    start <- return (0,0)
    first <- landLeft 2 start
    Nothing
    second <- landRight 2 first
    landLeft 1 second
```

Quand on écrit une ligne en notation `do` sans lier sa valeur monadique avec `<-`, c'est comme si l'on avait mis `>>` après la valeur monadique dont on souhaite ignorer le résultat. On séquence la valeur monadique mais on ignore son résultat parce qu'on s'en fiche, et parce que c'est plus joli que d'écrire `_ <- Nothing`, qui est équivalent.

À vous de choisir quand utiliser la notation `do` et quand utiliser explicitement `>>=`. Je pense que cet exemple se prête plus à l'utilisation explicite de `>>=` parce que chaque étape ne nécessite spécifiquement que le résultat de la précédente. Avec la notation `do`, il a fallu écrire à chaque fois sur quelle perche les oiseaux atterrissaient, mais c'était à chaque fois celle qui précédait immédiatement. Cependant, cela a fondé notre intuition de la notation `do`.

Avec la notation `do`, lorsqu'on lie des valeurs monadiques à des noms, on peut utiliser du filtrage par motif, tout comme dans les expressions `let` et les paramètres de fonctions. Voici un exemple de filtrage par motif dans une expression `do` :

```
justH :: Maybe Char
justH = do
    (x:xs) <- Just "hello"
    return x
```

On utilise le filtrage par motif pour obtenir le premier caractère de la chaîne `"hello"` et on la présente comme résultat. Ainsi, `justH` est évalué en `Just 'h'`.

Mais si ce filtrage par motif échoue ? Quand un filtrage par motif dans une fonction échoue, le prochain motif est essayé. Si le filtrage parcourt tous les motifs de la fonction donnée sans succès, une erreur est levée et notre programme plante. D'un autre côté, un échec de filtrage par motif dans une expression `let` résulte en une erreur immédiate, parce qu'elles n'ont pas de mécanisme de motif suivant. Quand un filtrage par motif échoue dans une expression `do`, la fonction `fail` est appelée. Elle fait partie de la classe de types `Monad` et permet à un filtrage par motif échoué de résulter en un échec dans le contexte de cette monade, plutôt que de planter notre programme. Son implémentation par défaut est :

```
fail :: (Monad m) => String -> m a
fail msg = error msg
```

Donc, par défaut, elle plante notre programme, mais les monades qui contiennent un contexte d'échec éventuel (comme `Maybe`) implémentent généralement leur propre version. Pour `Maybe`, elle est implémentée ainsi :

```
fail _ = Nothing
```

Elle ignore le message et crée un `Nothing`. Ainsi, quand le filtrage par motif échoue pour une valeur `Maybe` dans une notation `do`, la valeur de la monade entière est `Nothing`. C'est préférable à un plantage du programme. Voici une expression `do` avec un motif voué à l'échec :

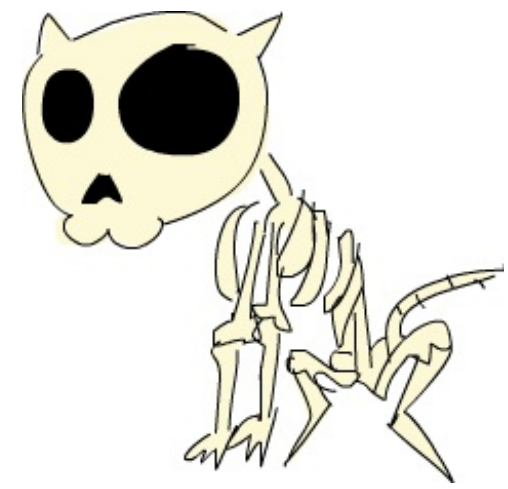
```
wopwop :: Maybe Char
wopwop = do
    (x:xs) <- Just ""
    return x
```

Le filtrage par motif échoue, donc l'effet est le même que si la ligne entière comprenant le motif était remplacée par `Nothing`. Essayons :

```
ghci> wopwop
Nothing
```

Le motif échoué a provoqué l'échec dans le contexte de la monade plutôt que de provoquer un échec du programme entier, ce qui est plutôt sympa.

La monade des listes



Jusqu'ici, on a vu comment les valeurs `Maybe` pouvaient être vues comme des valeurs dans un contexte d'échec, et comment incorporer la gestion d'erreur dans notre code en utilisant `>>=` pour alimenter les fonctions. Dans cette section, on va voir comment utiliser les aspects monadiques des listes pour introduire le non déterminisme dans notre code de manière claire et lisible.

On a déjà parlé du fait que les listes représentent des valeurs non déterministes quand elles sont utilisées comme foncteurs applicatifs. Une valeur comme `5` est déterministe. Elle n'a qu'un résultat, et on sait exactement lequel c'est. D'un autre côté, une valeur comme `[3, 8, 9]` contient plusieurs résultats, on peut donc la voir comme une valeur qui est en fait plusieurs valeurs à la fois. Utiliser les listes comme des foncteurs applicatifs démontre ce non déterminisme élégamment :

```
ghci> (*) <$> [1,2,3] <*> [10,100,1000]
[10,100,1000,20,200,2000,30,300,3000]
```

Toutes les combinaisons possibles de multiplication d'un élément de la liste de gauche par un élément de la liste de droite sont inclues dans la liste résultante. Quand on fait du non déterminisme, il y a beaucoup de choix possibles, donc on les essaie tous, et le résultat est également une valeur non déterministe, avec encore plus de résultats.

Ce contexte de non déterminisme se transfère très joliment aux monades. Regardons immédiatement l'instance de `Monad` pour les listes :

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

`return` fait la même chose que `pure`, on est donc déjà familiarisé avec `return` pour les listes. Elle prend une valeur et la place dans un contexte par défaut minimal retournant cette valeur. En d'autres termes, elle crée une liste qui ne contient que cette valeur comme résultat. C'est utile lorsque l'on veut envelopper une valeur normale dans une liste afin qu'elle puisse interagir avec des valeurs non déterministes.

Pour comprendre comment `>>=` fonctionne pour les listes, il vaut mieux la voir en action pour développer notre intuition d'abord. `>>=` consiste à prendre une

valeur dans un contexte (une valeur monadique) et la donner à une fonction qui prend une valeur normale et en retourne une dans un contexte. Si cette fonction produisait simplement une valeur normale plutôt qu'une dans un contexte, `>>=` ne nous serait pas très utile parce qu'après une utilisation, le contexte serait perdu. Essayons donc de donner une valeur non déterministe à une fonction.

```
ghci> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```

Quand on utilisait `>>=` avec `Maybe`, la valeur monadique était passée à la fonction en prenant en compte le possible échec. Ici, elle prend en compte le non déterminisme pour nous. `[3, 4, 5]` est une valeur non déterministe et on la donne à une fonction qui retourne également une valeur non déterministe. Le résultat est également non déterministe, et contient tous les résultats possibles consistant à prendre un élément de la liste `[3, 4, 5]` et de le passer à la fonction `\x -> [x, -x]`. Cette fonction prend un nombre et produit deux résultats : le nombre inchangé et son opposé. Ainsi, quand on utilise `>>=` pour donner cette liste à la fonction, chaque nombre est gardé inchangé et opposé. Le `x` de la lambda prend chaque valeur de la liste qu'on donne à la fonction.

Pour voir comment cela a lieu, on peut simplement suivre l'implémentation. D'abord, on commence avec la liste `[3, 4, 5]`. Puis, on mappe la lambda sur la liste, et le résultat est :

```
[[3,-3],[4,-4],[5,-5]]
```

La lambda est appliquée à chaque élément et on obtient une liste de listes. Finalement, on aplatit la liste, et voilà ! Nous avons appliqué une fonction non déterministe à une valeur non déterministe !

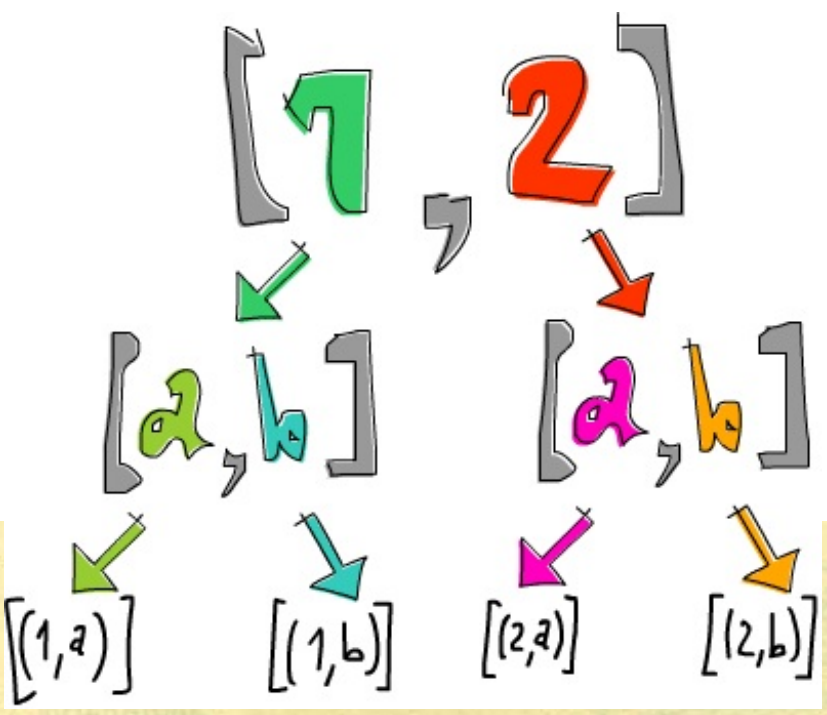
Le non déterminisme supporte également l'échec. La liste vide `[]` est presque l'équivalent de `Nothing`, parce qu'elle signifie l'absence de résultat. C'est pourquoi l'échec est défini comme la liste vide. Le message d'erreur est jeté. Jouons avec des listes qui échouent :

```
ghci> [] >>= \x -> ["bad","mad","rad"]
[]
ghci> [1,2,3] >>= \x -> []
[]
```

À la première ligne, une liste vide est donnée à la lambda. Puisque la liste n'a pas d'éléments, aucun élément ne peut être passé à la fonction et ainsi le résultat est une liste vide. C'est similaire à donner `Nothing` à une fonction. À la seconde ligne, chaque élément est passé à la fonction, mais l'élément est ignoré et la fonction retourne une liste vide. Parce que la fonction échoue quelle que soit son entrée, le résultat est un échec.

Tout comme les valeurs `Maybe`, on peut chaîner plusieurs listes avec `>>=`, propageant le non déterminisme :

```
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```



La liste `[1, 2]` est liée à `n` et `['a', 'b']` est liée à `ch`. Puis, on fait `return (n, ch)` (ou `[(n, ch)]`), ce qui signifie prendre une paire `(n, ch)` et la placer dans un contexte minimal. Dans ce cas, c'est la liste la plus petite contenant la valeur `(n, ch)` et étant le moins non déterministe possible. Son effet sur le contexte est donc minimal. Ce qu'on dit ici, c'est : pour chaque élément dans `[1, 2]`, et pour chaque élément de `['a', 'b']`, produis un tuple d'un élément de chaque liste.

En général, parce que `return` prend une valeur et l'enveloppe dans un contexte minimal, elle n'a pas d'effet additionnel (comme l'échec pour `Maybe`, ou ajouter du non déterminisme pour les listes), mais présente tout de même quelque chose en résultat.

Quand vous avez des valeurs non déterministes qui interagissent, vous pouvez voir leurs calculs comme un arbre où chaque résultat possible d'une liste représente une branche.

Voici l'exemple précédent réécrit avec la notation `do` :

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    return (n,ch)
```

Cela rend un peu plus évident le fait que `n` prenne chaque valeur dans `[1, 2]` et `ch` chaque valeur de `['a', 'b']`. Tout comme avec `Maybe`, on extrait des

éléments de valeurs monadiques, et on les traite comme des valeurs normales, et `>>=` s'occupe du contexte pour nous. Le contexte dans ce cas est le non déterminisme.

Utiliser les listes avec la notation `do` me rappelle quelque chose qu'on a déjà vu. Regardez le code suivant :

```
ghci> [ (n,ch) | n <- [1,2], ch <- ['a','b'] ]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Oui ! Les listes en compréhension ! Dans notre exemple avec la notation `do`, `n` prenait chaque résultat de `[1, 2]`, et pour chacun d'eux, `ch` recevait un résultat de `['a', 'b']`, et enfin la dernière ligne plaçait `(n, ch)` dans un contexte par défaut (une liste singleton) pour le présenter comme résultat sans introduire plus de non déterminisme. Dans cette liste en compréhension, la même chose a lieu, mais on n'a pas eu à écrire `return` à la fin pour présenter `(n, ch)` comme le résultat, parce que la fonction de sortie de la compréhension de liste s'en occupe pour nous.

En fait, les listes en compréhension sont juste un sucre syntaxique pour utiliser les listes comme des monades. Au final, les listes en compréhension, ainsi que les listes en notation `do`, sont traduites en utilisations de `>>=` pour faire les calculs non déterministes.

Les listes en compréhension nous permettent de filtrer la sortie. Par exemple, on peut filtrer une liste de nombres pour chercher seulement ceux qui contiennent le chiffre `7` :

```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

On applique `show` à `x` pour transformer notre nombre en chaîne de caractères, et on vérifie si le caractère `7` fait partie de cette chaîne. Plutôt malin. Pour voir comment le filtrage dans les listes en compréhension se traduit dans la monade des listes, il nous faut regarder la fonction `guard` et la classe de types `MonadPlus`. La classe de types `MonadPlus` est pour les monades qui peuvent aussi se comporter en monoïdes. Voici sa définition :

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

`mzero` est synonyme de `mempty` de la classe de types `Monoid`, et `mplus` correspond à `mappend`. Parce que les listes sont des monoïdes ainsi que des monades, elles peuvent être instance de cette classe de types :

```
instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

Pour les listes, `mzero` représente un calcul non déterministe qui n'a pas de résultats - un calcul échoué. `mplus` joint deux valeurs non déterministes en une. La fonction `guard` est définie ainsi :

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero
```

Elle prend une valeur booléenne et si c'est `True`, elle prend un `()` et le place dans un contexte minimal par défaut qui réussisse toujours. Sinon, elle crée une valeur monadique échouée. La voici en action :

```
ghci> guard (5 > 2) :: Maybe ()
Just ()
ghci> guard (1 > 2) :: Maybe ()
Nothing
ghci> guard (5 > 2) :: [()]
[()]
ghci> guard (1 > 2) :: [()]
[]
```

Ça a l'air intéressant, mais comment est-ce utile ? Dans la monade des listes, on l'utilise pour filtrer des calculs non déterministes. Observez :

```
ghci> [1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)
[7,17,27,37,47]
```

Le résultat ici est le même que le résultat de la liste en compréhension précédente. Comment `guard` fait-elle cela ? Regardons d'abord comment elle fonctionne en conjonction avec `>>` :

```
ghci> guard (5 > 2) >> return "cool" :: [String]
["cool"]
ghci> guard (1 > 2) >> return "cool" :: [String]
[]
```

Si `guard` réussit, le résultat qu'elle contient est un tuple vide. On utilise alors `>>` pour ignorer ce tuple vide et présenter quelque chose d'autre en retour. Cependant, si `guard` échoue, alors le `return` échouera aussi, parce que donner une liste vide à une fonction via `>>=` résulte toujours en une liste vide. `guard` dit simplement : si ce booléen est `False`, alors produis un échec immédiatement, sinon crée une valeur réussie factice avec `()`. Tout ce que cela permet est d'autoriser le calcul à continuer.

Voici l'exemple précédent réécrit en notation `do` :

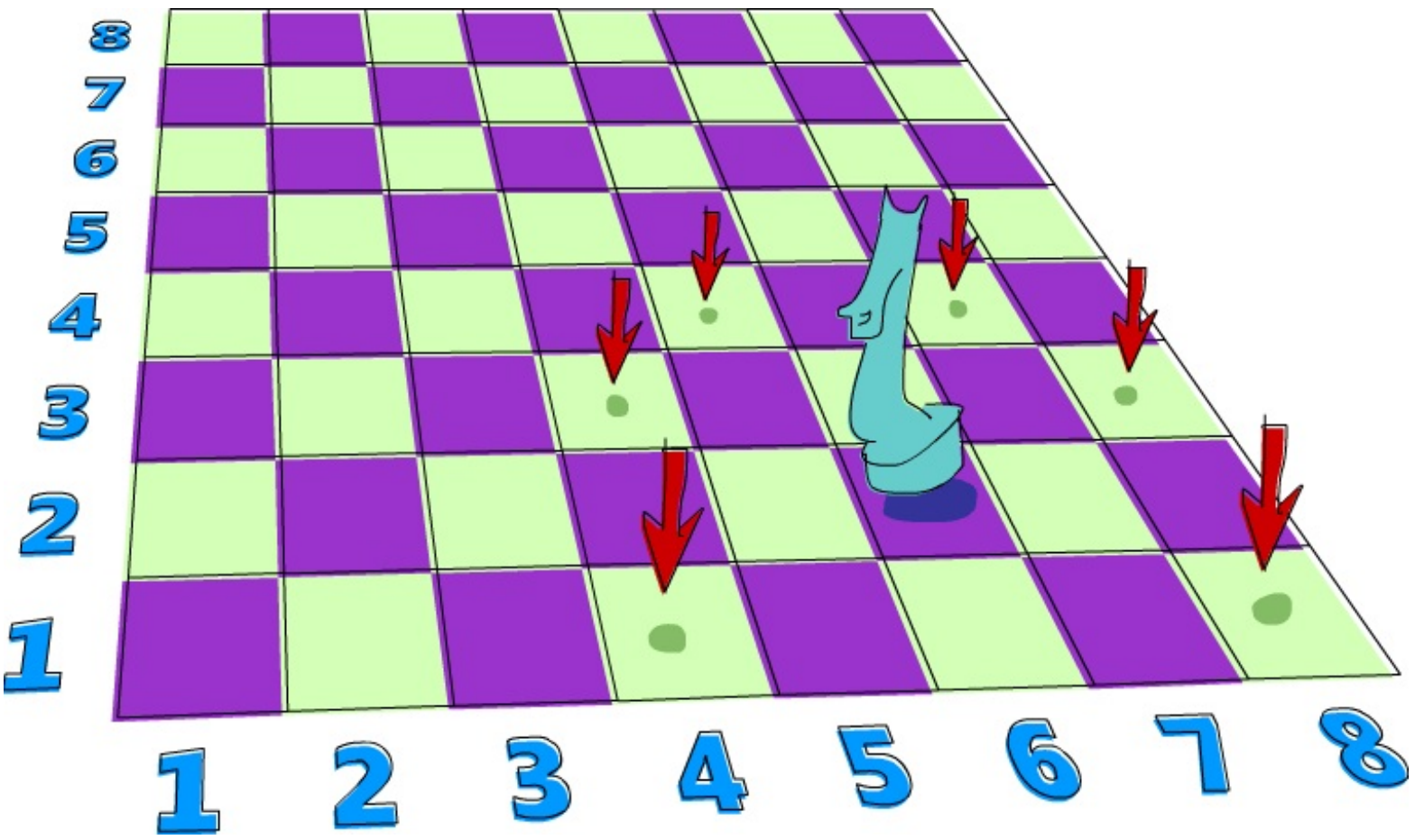
```
sevensOnly :: [Int]
sevensOnly = do
    x <- [1..50]
    guard ('7' `elem` show x)
    return x
```

Si nous avions oublié de présenter `x` en résultat final avec `return`, la liste résultat serait seulement une liste de tuples vides. Voici la même chose sous forme de liste en compréhension :

```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

La quête d'un cavalier

Voici un problème qui se prête particulièrement bien à une résolution non déterministe. Disons que vous ayez un échiquier, et seulement un cavalier placé dessus. On souhaite savoir si le cavalier peut atteindre une position donnée en trois mouvements. On utilisera simplement une paire de nombres pour représenter la position du cavalier sur l'échiquier. Le premier nombre détermine la colonne, et le second la ligne.



Créons un synonyme de type pour la position actuelle du cavalier sur l'échiquier :

```
type KnightPos = (Int,Int)
```

Donc, supposons que le cavalier démarre en `(6, 2)`. Peut-il aller en `(6, 1)` en exactement trois mouvements ? Voyons. Si l'on commence en `(6, 2)`, quel est le meilleur mouvement à faire ensuite ? Je sais, pourquoi pas tous les mouvements possibles ! On a à notre disposition le non déterminisme, donc plutôt que d'avoir à choisir un mouvement, choisissons-les tous à la fois. Voici une fonction qui prend la position d'un cavalier et retourne toutes ses prochaines positions :

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = do
    (c',r') <- [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
               ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
               ]
    guard (c' `elem` [1..8] && r' `elem` [1..8])
```

```
    return (c',r')
```

Le cavalier peut toujours avancer d'une case horizontalement et de deux verticalement, ou bien de deux cases horizontalement et d'une verticalement. `(c', r')` prend chaque valeur de la liste des mouvements, puis `guard` s'assure que le déplacement obtenu `(c', r')`, reste bien dans les limites de l'échiquier. Si ce n'est pas le cas, elle produit une liste vide qui cause un échec, et `return (c', r')` n'est pas exécutée pour cette position là.

Cette fonction peut être réécrite sans utiliser les listes comme des monades, mais on l'a fait quand même pour l'entraînement. Voici la même fonction avec `filter` :

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = filter onBoard
    [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
    ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
    ]
    where onBoard (c,r) = c `elem` [1..8] && r `elem` [1..8]
```

Les deux font la même chose, choisissez celle que vous trouvez plus jolie. Essayons :

```
ghci> moveKnight (6,2)
[(8,1),(8,3),(4,1),(4,3),(7,4),(5,4)]
ghci> moveKnight (8,1)
[(6,2),(7,3)]
```

Ça fonctionne à merveille ! On prend une position et on essaie tous les mouvements possibles à la fois, pour ainsi dire. À présent qu'on a une position non déterministe, on peut utiliser `>>=` pour la donner à `moveKnight`. Voici une fonction qui prend une position et retourne toutes les positions qu'on peut atteindre dans trois mouvements :

```
in3 :: KnightPos -> [KnightPos]
in3 start = do
    first <- moveKnight start
    second <- moveKnight first
    moveKnight second
```

Si vous lui passez `(6, 2)`, la liste résultante est assez grande, parce que s'il y a beaucoup de façons d'atteindre une position en trois mouvements, alors le résultat est dans la liste de multiples fois. La même chose sans notation `do` :

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

Utiliser `>>=` une fois nous donne tous les mouvements possibles depuis le point de départ, puis en utilisant `>>=` une deuxième fois, pour chacun de ces premiers mouvements, les mouvements suivants sont calculés, et de même pour le dernier mouvement.

Placer une valeur dans un contexte par défaut en faisant `return` pour ensuite la passer à une fonction avec `>>=` est équivalent à appliquer la fonction normalement à la valeur, mais on le fait quand même pour le style.

À présent, écrivons une fonction qui prend deux positions et nous dit si on peut aller de l'une à l'autre en exactement trois étapes :

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

On génère toutes les solutions possibles pour trois étapes, et on regarde si la position qu'on cherche est parmi celles-ci. Voyons donc si on peut aller de `(6, 2)` en `(6, 1)` en trois mouvements :

```
ghci> (6,2) `canReachIn3` (6,1)
True
```

Oui ! Qu'en est-il de `(6, 2)` et `(7, 3)` ?

```
ghci> (6,2) `canReachIn3` (7,3)
False
```

Non ! En tant qu'exercice, vous pouvez changer cette fonction afin que, lorsque vous pouvez atteindre une position depuis l'autre, elle vous dise quels mouvements faire. Plus tard, nous verrons comment modifier cette fonction pour qu'on puisse aussi lui passer le nombre de mouvements à faire plutôt que de le coder en dur comme ici.

Les lois des monades

Tout comme les foncteurs applicatifs, et les foncteurs avant eux, les monades viennent avec quelques lois que toutes les instances de `Monad` doivent respecter. Être simplement une instance de `Monad` ne fait pas automatiquement une monade, cela signifie simplement qu'on a créé une instance. Un type est réellement une monade si les lois des monades sont respectées. Ces lois nous permettent d'avoir des suppositions raisonnables sur le type et son comportement.

Haskell permet à n'importe quel type d'être une instance de n'importe quelle classe de types tant que les types correspondent. Il ne peut pas vérifier si les lois sont respectées, donc si l'on crée une nouvelle instance de la classe de types `Monad`, on doit être raisonnablement convaincu que ce type se comporte bien vis-à-vis des lois des monades. On peut faire confiance aux types de la bibliothèque standard pour satisfaire ces lois, mais plus tard, lorsque l'on écrira nos propres monades, il faudra vérifier manuellement que les lois tiennent. Mais ne vous inquiétez pas, elles ne sont pas compliquées.

Composition à gauche par return

La première loi dit que si l'on prend une valeur, qu'on la place dans un contexte par défaut via `return` et qu'on la donne à une fonction avec `>>=`, c'est la même chose que de donner directement la valeur à la fonction. Plus formellement :

- `return x >>= f` est égal à `f x`

Si vous regardez les valeurs monadiques comme des valeurs avec un contexte, et `return` comme prenant une valeur et la plaçant dans un contexte minimal qui présente toujours cette valeur, c'est logique, parce que si ce contexte est réellement minimal, alors donner la valeur monadique à la fonction ne devrait pas être différent de l'application de la fonction à la valeur normale, et c'est en effet le cas.

Pour la monade `Maybe`, `return` est définie comme `Just`. La monade `Maybe` se préoccupe des échecs possibles, et si l'on a une valeur qu'on met dans un tel contexte, c'est logique de la traiter comme un calcul réussi, puisque l'on connaît sa valeur. Voici `return` utilisé avec `Maybe` :

```
ghci> return 3 >>= (\x -> Just (x+100000))
Just 100003
ghci> (\x -> Just (x+100000)) 3
Just 100003
```

Pour la monade des listes, `return` place une valeur dans une liste singleton. L'implémentation de `>>=` prend chaque valeur de la liste et applique la fonction dessus, mais puisqu'il n'y a qu'une valeur dans une liste singleton, c'est la même chose que d'appliquer la fonction à la valeur :

```
ghci> return "WoM" >>= (\x -> [x,x,x])
["WoM","WoM","WoM"]
ghci> (\x -> [x,x,x]) "WoM"
["WoM","WoM","WoM"]
```

On a dit que pour `IO`, `return` retournait une action I/O qui n'avait pas d'effet de bord mais retournait la valeur en résultat. Il est logique que cette loi tienne ainsi pour `IO` également.

Composition à droite par return

La deuxième loi dit que si l'on a une valeur monadique et qu'on utilise `>>=` pour la donner à `return`, alors le résultat est la valeur monadique originale. Formellement :

- `m >>= return` est égal à `m`

Celle-ci peut être un peu moins évidente que la précédente, mais regardons pourquoi elle doit être respectée. Lorsqu'on donne une valeur monadique à une fonction avec `>>=`, cette fonction prend une valeur normale et retourne une valeur monadique. `return` est une telle fonction, si vous considérez son type. Comme on l'a dit, `return` place une valeur dans un contexte minimal qui présente cette valeur en résultat. Cela signifie, par exemple, pour `Maybe`, qu'elle n'introduit pas d'échec, et pour les listes, qu'elle n'ajoute pas de non déterminisme. Voici un essai sur quelques monades :

```
ghci> Just "move on up" >>= (\x -> return x)
Just "move on up"
ghci> [1,2,3,4] >>= (\x -> return x)
[1,2,3,4]
ghci> putStrLn "Wah!" >>= (\x -> return x)
Wah!
```

Si l'on regarde de plus près l'exemple des listes, l'implémentation de `>>=` est :

```
xs >>= f = concat (map f xs)
```

Donc, quand on donne `[1, 2, 3, 4]` à `return`, d'abord `return` est mappée sur `[1, 2, 3, 4]`, résultant en `[[1],[2],[3],[4]]` et ensuite ceci est concaténé et retourne notre liste originale.

Les lois de composition à gauche et à droite par `return` décrivent simplement comment `return` doit se comporter. C'est une fonction importante pour faire des valeurs monadiques à partir de valeurs normales, et ce ne serait pas bon que les valeurs qu'elle produit fassent des tas d'autres choses.

Associativité

La dernière loi des monades dit que quand on a une chaîne d'application de fonctions monadiques avec `>>=`, l'ordre d'imbrication ne doit pas importer. Formellement énoncé :

- `(m >>= f) >>= g` est egal à `m >>= (\x -> f x >>= g)`

Hmmm, que se passe-t-il donc là ? On a une valeur monadique `m`, et deux fonctions monadiques `f` et `g`. Quand on fait `(m >>= f) >>= g`, on donne `m` à `f`, ce qui résulte en une valeur monadique. On donne ensuite cette valeur monadique à `g`. Dans l'expression `m >>= (\x -> f x >>= g)`, on prend une valeur monadique et on la donne à une fonction qui donne le résultat de `f x` à `g`. Il n'est pas facile de voir que les deux sont égales, regardons donc un exemple qui rend cette égalité un peu plus claire.

Vous souvenez-vous de notre funambule Pierre qui marchait sur une corde tendue pendant que des oiseaux atterrissaient sur sa perche ? Pour simuler les oiseaux atterrissant sur sa perche, on avait chaîné plusieurs fonctions qui pouvaient mener à l'échec :

```
ghci> return (0,0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

On commençait avec `Just (0, 0)` et on liait cette valeur à la prochaine fonction monadique, `landRight 2`. Le résultat était une nouvelle valeur monadique qui était liée dans la prochaine fonction monadique, et ainsi de suite. Si nous parenthésions explicitement ceci, cela serait :

```
ghci> ((return (0,0) >>= landRight 2) >>= landLeft 2) >>= landRight 2
Just (2,4)
```

Mais on peut aussi écrire la routine ainsi :

```
return (0,0) >>= (\x ->
landRight 2 x >>= (\y ->
landLeft 2 y >>= (\z ->
landRight 2 z)))
```

`return (0, 0)` est identique à `Just (0, 0)` et quand on le donne à la lambda, `x` devient `(0, 0)`. `landRight` prend un nombre d'oiseaux et une perche (une paire de nombres) et c'est ce qu'elle reçoit. Cela résulte en `Just (0, 2)` et quand on donne ceci à la prochaine lambda, `y` est égal à `(0, 2)`. Cela continue jusqu'à ce que le dernier oiseau se pose, produisant `Just (2, 4)`, qui est effectivement le résultat de l'expression entière.

Ainsi, il n'importe pas de savoir comment vous imbriquez les appels de fonctions monadiques, ce qui importe c'est leur sens. Voici une autre façon de regarder cette loi : considérez la composition de deux fonctions, `f` et `g`. Composer deux fonctions se fait ainsi :

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = (\x -> f (g x))
```

Si le type de `g` est `a -> b` et le type de `f` est `b -> c`, on peut les arranger en une nouvelle fonction ayant pour type `a -> c`, de façon à ce que le paramètre soit passé entre les deux fonctions. Et si ces deux fonctions étaient monadiques, et retournait des valeurs monadiques ? Si l'on avait une fonction de type `a -> m b`, on ne pourrait pas simplement passer son résultat à une fonction de type `b -> m c`, parce que la fonction attend un `b` normal, pas un monadique. On pourrait cependant utiliser `>>=` pour réaliser cela. Ainsi, en utilisant `>>=`, on peut composer deux fonctions monadiques :

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```

On peut à présent composer deux fonctions monadiques :

```
ghci> let f x = [x,-x]
```

```
ghci> let g x = [x*3,x*2]
ghci> let h = f <=< g
ghci> h 3
[9,-9,6,-6]
```

Cool. Qu'est-ce que cela a à voir avec la loi d'associativité ? Eh bien, quand on regarde la loi comme une loi de composition, elle dit que `f <=< (g <=< g)` doit être égal à `(f <=< g) <=< h`. C'est juste une autre façon de dire que l'imbrication des opérations n'importe pas pour les monades.

Si l'on traduit les deux premières lois avec **`<=<`**, la composition de **`return`** à gauche dit que pour toute fonction monadique **`f`**, `f <=< return` est égal à `f` et celle à droite dit que `return <=< f` est aussi égal à `f`.

Dans ce chapitre, on a vu quelques bases des monades et appris comment la monade **`Maybe`** et la monade des listes fonctionnent. Dans le prochain chapitre, nous regarderons tout un tas d'autre monades cool, et on apprendra à créer les nôtres.

⬅ Foncteurs, foncteurs applicatifs et monoïdes Table des matières Et pour quelques monades de plus ➡

Et pour quelques monades de plus

← Pour une poignée de monades | Table des matières | Zippeurs →

Nous avons vu comment les monades pouvaient être utilisées pour prendre des valeurs dans un contexte et appliquer des fonctions dessus, et comment utiliser `>>=` et la notation `do` nous permettait de nous concentrer sur les valeurs elles-même pendant que le contexte était géré pour nous.

Nous avons croisé la monade `Maybe` et avons vu comment elle ajoutait un contexte d'échec potentiel. Nous avons appris la monade des listes et vu comment elle nous permettait d'introduire aisément du non déterminisme dans nos programmes. Nous avons aussi appris comment travailler dans la monade `IO`, avant même de savoir ce qu'était une monade !

Dans ce chapitre, on va apprendre beaucoup d'autres monades. On verra comment elles peuvent rendre notre programme plus clair en nous laissant traiter toutes sortes de valeurs comme des valeurs monadiques. Explorer d'autres monades nous aidera également à solidifier notre intuition de ce qu'elles sont.

Les monades que nous verrons font toutes partie du paquet `mtl`. Un paquet Haskell est une collection de modules. Le paquet `mtl` vient avec la plate-forme Haskell, donc vous l'avez probablement. Pour vérifier si c'est le cas, tapez `ghc-pkg list` dans l'invite de commande. Cela montrera quels paquets vous avez installés, et l'un d'entre eux devrait être `mtl` suivi d'un numéro de version.

Lui écrire ? Je la connais à peine !

Nous avons chargé notre pistolet avec les monades `Maybe`, liste et `IO`. Plaçons maintenant la monade `Writer` dans la chambre, et voyons ce qui se passe quand on tire !

Alors que `Maybe` est pour les valeurs ayant un contexte additionnel d'échec, et que les listes sont pour les valeurs non déterministes, la monade `Writer` est faite pour les valeurs qui peuvent avoir une autre valeur attachée agissant comme une sorte de registre. `Writer` nous permet d'effectuer des calculs tout en étant sûrs que toutes les valeurs du registre sont bien combinées en un registre qui reste attaché au résultat.

Par exemple, on peut vouloir munir nos valeurs d'une chaîne de caractères décrivant ce qui se passe, probablement pour déboguer notre programme. Considérez une fonction qui prend un nombre de bandits d'un gang et nous dit si c'est un gros gang ou pas. C'est une fonction très simple :

```
isBigGang :: Int -> Bool
isBigGang x = x > 9
```

Maintenant, et si au lieu de nous répondre seulement `True` ou `False`, on voulait aussi retourner une chaîne de caractères indiquant ce qu'on a fait ? Eh bien, il suffit de créer une chaîne et la retourner avec le `Bool` :

```
isBigGang :: Int -> (Bool, String)
isBigGang x = (x > 9, "Compared gang size to 9.")
```

À présent, au lieu de retourner juste un `Bool`, on retourne un tuple dont la première composante est la vraie valeur de retour, et la seconde est la chaîne accompagnant la valeur. Il y a un contexte additionnel à présent. Testons :

```
ghci> isBigGang 3
(False,"Compared gang size to 9.")
ghci> isBigGang 30
(True,"Compared gang size to 9.")
```

Jusqu'ici, tout va bien. `isBigGang` prend une valeur normale et retourne une valeur dans un contexte. Comme on vient de le voir, lui donner une valeur normale n'est pas un problème. Et si l'on avait déjà une valeur avec un registre attaché, comme `(3, "Smallish gang.")`, et qu'on voulait la donner à `isBigGang` ? Il semblerait qu'on se retrouve à nouveau face à la question : si l'on a une fonction qui prend une valeur normale et retourne une valeur dans un contexte, comment lui passer une valeur dans un contexte ?

Quand on explorait la monade `Maybe`, on a créé une fonction `applyMaybe`, qui prenait un `Maybe a` et une fonction de type `a -> Maybe b` et on donnait la valeur `Maybe a` à la fonction, bien qu'elle attende un `a` normal et pas un `Maybe a`. Ceci était

fait en prenant en compte le contexte qui venait avec la valeur `Maybe a`, qui était celui de l'échec potentiel. Mais une fois dans la fonction `a -> Maybe b`, on pouvait traiter la valeur comme une valeur normale, parce que `applyMaybe` (qui devint ensuite `>>=`) s'occupait de vérifier si c'était un `Nothing` ou un `Just`.

Dans la même veine, créons une fonction qui prend une valeur avec un registre attaché, c'est-à-dire, de type `(a, String)`, et une fonction `a -> (b, String)`, et qui donne cette valeur à cette fonction. On va l'appeler `applyLog`. Mais puisqu'une valeur `(a, String)` ne contient pas le contexte d'échec potentiel, mais plutôt un contexte de valeur additionnelle, `applyLog` va s'assurer que le registre de la valeur originale n'est pas perdu, mais est accolé au registre de la valeur résultant de la fonction. Voici l'implémentation d'`applyLog` :

```
applyLog :: (a,String) -> (a -> (b,String)) -> (b,String)
applyLog (x,log) f = let (y,newLog) = f x in (y,log ++ newLog)
```

Quand on a une valeur dans un contexte et qu'on souhaite la donner à une fonction, on essaie généralement de séparer la vraie valeur du contexte, puis on applique la fonction à cette valeur, et on s'occupe enfin de la gestion du contexte. Dans la monade `Maybe`, on vérifiait si la valeur était un `Just x` et si c'était le cas, on prenait ce `x` et on appliquait la fonction. Dans ce cas, il est encore plus simple de trouver la vraie valeur, parce qu'on a une paire contenant la valeur et un registre. On prend simplement la première composante, qui est `x` et on applique `f` avec. On obtient une paire `(y, newLog)`, où `y` est le nouveau résultat, et `newLog` le nouveau registre. Mais si l'on retournait cela en résultat, on aurait oublié l'ancien registre, ainsi on retourne une paire `(y, log ++ newLog)`. On utilise `++` pour juxtaposer le nouveau registre et l'ancien.

Voici `applyLog` en action :

```
ghci> (3, "Smallish gang.") `applyLog` isBigGang
(False,"Smallish gang.Compared gang size to 9")
ghci> (30, "A freaking platoon.") `applyLog` isBigGang
(True,"A freaking platoon.Compared gang size to 9")
```

Les résultats sont similaires aux précédents, seulement le nombre de personne dans le gang avait un registre l'accompagnant, et ce registre a été inclus dans le registre résultant. Voici d'autres exemples d'utilisation d'`applyLog` :

```
ghci> ("Tobin","Got outlaw name.") `applyLog` (\x -> (length x, "Applied length."))
(5,"Got outlaw name.Applied length.")
ghci> ("Bathcat","Got outlaw name.") `applyLog` (\x -> (length x, "Applied length"))
(7,"Got outlaw name.Applied length")
```

Voyez comme, dans la lambda, `x` est simplement une chaîne de caractères normale et non pas un tuple, et comment `applyLog` s'occupe de la juxtaposition des registres.

Monoïdes à la rescousse

Soyez certain de savoir ce que sont les monoïdes avant de continuer ! Cordialement.

Pour l'instant, `applyLog` prend des valeurs de type `(a, String)`, mais y a-t-il une raison à ce que le registre soit une `String` ? On utilise `++` pour juxtaposer les registres, ne devrait-ce donc pas marcher pour n'importe quel type de liste, pas seulement des listes de caractères ? Bien sûr que oui. On peut commencer par changer son type en :

```
applyLog :: (a,[c]) -> (a -> (b,[c])) -> (b,[c])
```

À présent, le registre est une liste. Le type des valeurs dans la liste doit être le même dans la valeur originale que dans la valeur retournée par la fonction, autrement on ne saurait utiliser `++` pour les juxtaposer.

Est-ce que cela marcherait pour des chaînes d'octets ? Il n'y a pas de raison que ça ne marche pas. Cependant, le type qu'on a là ne marche que pour les listes. On dirait qu'il nous faut une autre fonction `applyLog` pour les chaînes d'octets. Mais attendez ! Les listes et les chaînes d'octets sont des monoïdes. En tant que tels, elles sont toutes deux des instances de la classe de types `Monoid`, ce qui signifie qu'elles implémentent la fonction `mappend`. Et pour les listes autant que les chaînes d'octets, `mappend` sert à concaténer. Regardez :

```
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]
ghci> B.pack [99,104,105] `mappend` B.pack [104,117,97,104,117,97]
Chunk "chi" (Chunk "huahua" Empty)
```

Cool ! Maintenant, `applyLog` peut fonctionner sur n'importe quel monoïde. On doit changer son type pour refléter cela, ainsi que son implémentation pour

remplacer `++` par `mappend` :

```
applyLog :: (Monoid m) => (a,m) -> (a -> (b,m)) -> (b,m)
applyLog (x,log) f = let (y,newLog) = f x in (y,log `mappend` newLog)
```

Puisque la valeur accompagnante peut être n'importe quelle valeur monoïdale, plus besoin de penser à un tuple valeur et registre, on peut désormais penser à un tuple valeur et valeur monoïdale. Par exemple, on peut avoir un tuple contenant un nom d'objet et un prix en tant que valeur monoïdale. On utilise le `newtype` `Sum` pour s'assurer que les prix sont bien additionnés lorsqu'on opère sur les objets. Voici une fonction qui ajoute des boissons à de la nourriture de cow-boy :

```
import Data.Monoid

type Food = String
type Price = Sum Int

addDrink :: Food -> (Food,Price)
addDrink "beans" = ("milk", Sum 25)
addDrink "jerky" = ("whiskey", Sum 99)
addDrink _ = ("beer", Sum 30)
```

On utilise des chaînes de caractères pour représenter la nourriture, et un `Int` dans un `newtype` `Sum` pour tracer le nombre de centimes que quelque chose coûte. Juste un rappel, faire `mappend` sur des `Sum` résulte en la somme des valeurs enveloppées :

```
ghci> Sum 3 `mappend` Sum 9
Sum {getSum = 12}
```

La fonction `addDrink` est plutôt simple. Si l'on mange des haricots, elle retourne `"milk"` ainsi que `Sum 25`, donc 25 centimes encapsulés dans un `Sum`. Si l'on mange du bœuf séché, on boit du whisky, et si l'on mange quoi que ce soit d'autre, on boit une bière. Appliquer normalement une fonction à de la nourriture ne serait pas très intéressant ici, mais utiliser `applyLog` pour donner une nourriture qui a un prix à cette fonction est intéressant :

```
ghci> ("beans", Sum 10) `applyLog` addDrink
("milk",Sum {getSum = 35})
ghci> ("jerky", Sum 25) `applyLog` addDrink
("whiskey",Sum {getSum = 124})
ghci> ("dogmeat", Sum 5) `applyLog` addDrink
("beer",Sum {getSum = 35})
```

Du lait coûte `25` centimes, mais si on le prend avec des haricots coûtant `10` centimes, on paie au final `35` centimes. Il est à présent clair que la valeur attachée n'a pas besoin d'être un registre, elle peut être n'importe quel valeur monoïdale, et la façon dont deux de ces valeurs sont combinées dépend du monoïde. Quand nous faisions des registres, elles étaient juxtaposées, mais à présent, les nombres sont sommés.

Puisque la valeur qu'`addDrink` retourne est un tuple `(Food, Price)`, on peut donner ce résultat à `addDrink` à nouveau, pour qu'elle nous dise ce qu'on devrait boire avec notre boisson et combien le tout nous coûterait. Essayons :

```
ghci> ("dogmeat", Sum 5) `applyLog` addDrink `applyLog` addDrink
("beer",Sum {getSum = 65})
```

Ajouter une boisson à de la nourriture pour chien retourne une bière et un prix additionnel de `30` centimes, donc `("beer", Sum 35)`. Et si l'on utilise `applyLog` pour donner cela à `addDrink`, on obtient une autre bière et le résultat est `("beer", Sum 65)`.

Le type Writer

Maintenant qu'on a vu qu'une valeur couplée à un monoïde agissait comme une valeur monadique, examinons l'instance de `Monad` pour de tels types. Le module `Control.Monad.Writer` exporte le type `Writer w a` ainsi que son instance de `Monad` et quelques fonctions utiles pour manipuler des valeurs de ce type.

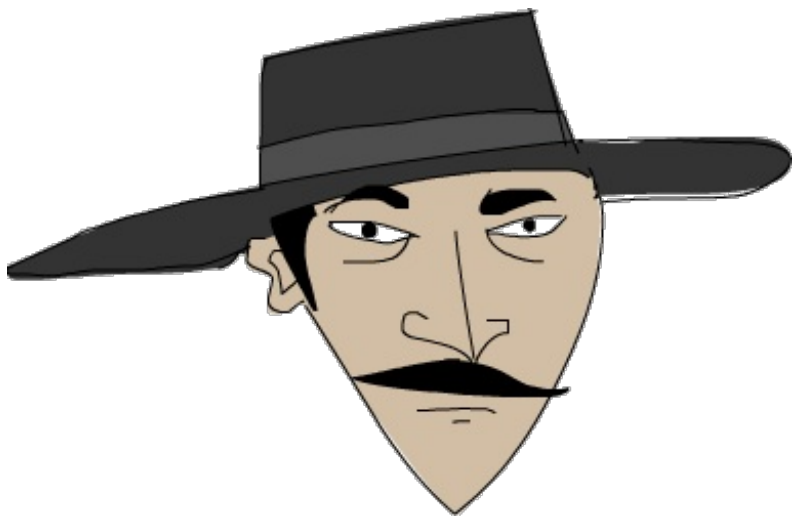
D'abord, examinons le type lui-même. Pour attacher un monoïde à une valeur, on doit simplement les placer ensemble dans un tuple. Le type `Writer w a` est juste un enrobage `newtype` de cela. Sa définition est très simple :

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

C'est enveloppé dans un `newtype` afin d'être fait instance de `Monad` et de séparer ce type des tuples ordinaires. Le paramètre de type `a` représente le type de la valeur, alors que le paramètre de type `w` est la valeur monoïdale attachée.

Son instance de `Monad` est définie de la sorte :

```haskell
instance (Monoid w) => Monad (Writer w) where
    return x = Writer (x, mempty)
    (Writer (x,v)) >>= f = let (Writer (y, v')) = f x in Writer (y, v `mappend` v')
```



Tout d'abord, examinons `>>=`. Son implémentation est essentiellement identique à `applyLog`, seulement à présent que notre tuple est enveloppé dans un `newtype` `Writer`, on doit l'en sortir en filtrant par motif. On prend la valeur `x` et applique la fonction `f`. Cela nous rend une valeur `Writer w a` et on utilise un filtrage par motif via une expression `let` dessus. On présente le `y` comme nouveau résultat et on utilise `mappend` pour combiner l'ancienne valeur monoïdale avec la nouvelle. On replace ceci et le résultat dans un constructeur `Writer` afin que notre résultat soit bien une valeur `Writer` et pas simplement un tuple non encapsulé.

Qu'en est-il de `return` ? Elle doit prendre une valeur et la placer dans un contexte minimal qui retourne ce résultat. Quel serait un tel contexte pour une valeur `Writer` ? Si l'on souhaite que notre valeur monoïdale affecte aussi faiblement que possible les autres valeurs monoïdales, il est logique d'utiliser `mempty`. `mempty` est l'élément neutre des valeurs monoïdales, comme `""` ou `Sum 0` ou une chaîne d'octets vide. Quand on utilise `mappend` avec `mempty` et une autre valeur monoïdale, le résultat est égal à cette autre valeur. Ainsi, si l'on utilise `return` pour créer une valeur `Writer` et qu'on utilise `>>=` pour donner cette valeur à une fonction, la valeur monoïdale résultante sera uniquement ce que la fonction retourne. Utilisons `return` sur le nombre `3` quelques fois, en lui attachant un monoïde différent à chaque fois :

```haskell
ghci> runWriter (return 3 :: Writer String Int)
(3,"")
ghci> runWriter (return 3 :: Writer (Sum Int) Int)
(3,Sum {getSum = 0})
ghci> runWriter (return 3 :: Writer (Product Int) Int)
(3,Product {getProduct = 1})
```

Puisque `Writer` n'a pas d'instance de `Show`, on a dû utiliser `runWriter` pour convertir nos valeurs `Writer` en tuples normaux qu'on peut alors afficher. Pour les `String`, la valeur monoïdale est la chaîne vide. Avec `Sum`, c'est `0`, parce que si l'on ajoute 0 à quelque chose, cette chose est inchangée. Pour `Product`, le neutre est `1`.

L'instance `Writer` n'a pas d'implémentation de `fail`, donc si un filtrage par motif échoue dans une notation `do`, `error` est appelée.

Utiliser la notation do avec Writer

À présent qu'on a une instance de `Monad`, on est libre d'utiliser la notation `do` pour les valeurs `Writer`. C'est pratique lorsqu'on a plusieurs valeurs `Writer` et qu'on veut faire quelque chose avec. Comme les autres monades, on peut les traiter comme des valeurs normales et les contextes sont pris en compte pour nous. Dans ce cas, les valeurs monoïdales sont attachées et `mappend` les unes aux autres et ceci se reflète dans le résultat final. Voici un exemple simple de l'utilisation de la notation `do` avec `Writer` pour multiplier des nombres.

```haskell
import Control.Monad.Writer

logNumber :: Int -> Writer [String] Int
logNumber x = Writer (x, ["Got number: " ++ show x])

multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    return (a*b)
```

`logNumber` prend un nombre et crée une valeur `Writer`. Pour le monoïde, on utilise une liste de chaînes de caractères et on donne au nombre une liste singleton qui dit simplement qu'on a ce nombre. `multWithLog` est une valeur `Writer` qui multiplie `3` et `5` et s'assure que leurs registres attachés sont inclus dans le registre final. On utilise `return` pour présenter `a*b` comme résultat. Puisque `return` prend simplement quelque chose et le place dans un contexte minimal, on peut être sûr de ne rien avoir ajouté au registre. Voici ce qu'on voit en évaluant ceci :

```haskell
ghci> runWriter multWithLog
(15,["Got number: 3","Got number: 5"])
```

Parfois, on veut seulement inclure une valeur monoïdale à partir d'un endroit donné. Pour cela, la fonction `tell` est utile. Elle fait partie de la classe de types `MonadWriter` et dans le cas de `Writer`, elle prend une valeur monoïdale, comme `["This is going on"]` et crée une valeur `Writer` qui présente la valeur factice `()` comme son résultat, mais avec notre valeur monoïdale attachée. Quand on a une valeur monoïdale qui a un `()` en résultat, on ne le lie pas à une variable. Voici `multWithLog` avec un message supplémentaire rapporté dans le registre :

```haskell
multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    tell ["Gonna multiply these two"]
    return (a*b)
```

Il est important que `return (a*b)` soit la dernière ligne, parce que le résultat de la dernière ligne d'une expression `do` est le résultat de l'expression entière. Si l'on avait placé `tell` à la dernière ligne, `()` serait le résultat de l'expression `do`. On aurait perdu le résultat de la multiplication. Cependant, le registre serait le même. Place à l'action :

```haskell
ghci> runWriter multWithLog
(15,["Got number: 3","Got number: 5","Gonna multiply these two"])
```

Ajouter de la tenue de registre à nos programmes

L'algorithme d'Euclide est un algorithme qui prend deux nombres et calcule leur plus grand commun diviseur. C'est-à-dire, le plus grand nombre qui divise à la fois ces deux nombres. Haskell contient déjà la fonction `gcd`, qui calcule exactement ceci, mais implémentons la nôtre avec des capacités de registre. Voici l'algorithme normal :

```haskell
gcd' :: Int -> Int -> Int
gcd' a b
    | b == 0    = a
    | otherwise = gcd' b (a `mod` b)
```

L'algorithme est très simple. D'abord, il vérifie si le second nombre est 0. Si c'est le cas, alors le premier est le résultat. Sinon, le résultat est le plus grand commun diviseur du second nombre et du reste de la division du premier par le second. Par exemple, si l'on veut connaître le plus grand commun diviseur de 8 et 3, on suit simplement cet algorithme. Parce que 3 est différent de 0, on doit trouver le plus grand commun diviseur de 3 et 2 (car si l'on divise 8 par 3, il reste 2). Ensuite, on cherche le plus grand commun diviseur de 3 et 2. 2 est différent de 0, on obtient donc 2 et 1. Le second nombre n'est toujours pas 0, alors on lance l'algorithme à nouveau sur 1 et 0, puisque diviser 2 par 1 nous donne un reste de 0. Finalement, puisque le second nombre est 0, alors le premier est le résultat final, c'est-à-dire 1. Voyons si le code est d'accord :

```haskell
ghci> gcd' 8 3
1
```

C'est le cas. Très bien ! Maintenant, on veut munir notre résultat d'un contexte, et ce contexte sera une valeur monoïdale agissant comme un registre. Comme auparavant, on utilisera une liste de chaînes de caractères pour notre monoïde. Le type de notre nouvelle fonction `gcd'` devrait donc être :

```haskell
gcd' :: Int -> Int -> Writer [String] Int
```

Il ne reste plus qu'à munir notre fonction de valeurs avec registres. Voici le code :

```haskell
import Control.Monad.Writer

gcd' :: Int -> Int -> Writer [String] Int
gcd' a b
    | b == 0 = do
        tell ["Finished with " ++ show a]
        return a
    | otherwise = do
        tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
        gcd' b (a `mod` b)
```

La fonction prend deux valeurs `Int` normales et retourne un `Writer [String] Int`, c'est-à-dire, un `Int` avec un registre en contexte. Dans le cas où `b` vaut `0`, plutôt que de donner simplement le `a` en résultat, on utilise une expression `do` pour le placer dans une valeur `Writer` en résultat. On utilise d'abord `tell` pour rapporter qu'on a terminé, puis `return` pour présenter `a` comme résultat de l'expression `do`. Au lieu de cette expression `do`, on aurait aussi pu écrire :

```haskell
Writer (a, ["Finished with " ++ show a])
```

Cependant, je pense que l'expression `do` est plus simple à lire. Ensuite, on a le cas où `b` est différent de `0`. Dans ce cas, on écrit qu'on utilise `mod` pour trouver le reste de la division de `a` par `b`. Puis, à la deuxième ligne de l'expression `do` on appelle récursivement `gcd'`. Souvenez-vous, `gcd'` retourne finalement une valeur `Writer`, il est donc parfaitement valide d'utiliser `gcd' b (a `mod` b)` dans une ligne d'une expression `do`.

Bien qu'il puisse être assez utile de tracer l'exécution de notre nouvelle `gcd'` à la main pour voir comment les registres sont juxtaposés, je pense qu'il sera plus

enrichissant de prendre de la perspective et voir ceux-ci comme des valeurs avec un contexte, et ainsi imaginer ce que notre résultat devrait être.

Essayons notre nouvelle fonction `gcd'`. Son résultat est une valeur `Writer [String] Int` et si on l'extrait de son `newtype`, on obtient un tuple. La première composante de cette paire est le résultat. Voyons si c'est le cas :

```
ghci> fst $ runWriter (gcd' 8 3)
1
```

Bien ! Qu'en est-il du registre ? Puisqu'il n'est qu'une liste de chaînes de caractères, utilisons `mapM_ putStrLn` pour afficher ces chaînes à l'écran :

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcd' 8 3)
8 mod 3 = 2
3 mod 2 = 1
2 mod 1 = 0
Finished with 1
```

Je trouve assez génial qu'on ait pu changer notre algorithme ordinaire en un algorithme reportant ce qu'il fait à la volée en changeant simplement les valeurs normales par des valeurs monadiques et en laissant l'implémentation de `>>=` pour `Writer` s'occuper des registres pour nous. On peut ajouter un mécanisme de tenue de registre à n'importe quelle fonction. On remplace simplement les valeurs normales par des valeurs `Writer`, et on remplace l'application de fonction usuelle par `>>=` (ou par des expressions `do` si cela augmente la lisibilité).

Construction inefficace de listes

Quand vous utilisez la monade `Writer`, il faut être extrêmement prudent avec le choix du monoïde à utiliser, parce qu'utiliser des listes peut s'avérer très lent. C'est parce que les listes utilisent `++` pour `mappend`, et utiliser `++` pour ajouter quelque chose à la fin d'une liste peut s'avérer très lent si la liste est très longue.

Dans notre fonction `gcd'`, la construction du registre est rapide parce que la concaténation se déroule ainsi :

```
a ++ (b ++ (c ++ (d ++ (e ++ f))))
```

Les listes sont des structures de données construites de la gauche vers la droite, et ceci est efficace parce que l'on construit d'abord entièrement la partie de gauche d'une liste, et ensuite on ajoute une liste plus longue à droite. Mais si l'on ne fait pas attention, utiliser la monade `Writer` peut produire une concaténation comme celle-ci :

```
((((a ++ b) ++ c) ++ d) ++ e) ++ f
```

Celle-ci est associée à gauche plutôt qu'à droite. C'est inefficace parce que chaque fois que l'on veut ajouter une partie droite à une partie gauche, elle doit construire la partie gauche en entier du début !

La fonction suivante fonctionne comme `gcd'`, mais enregistre les choses dans le sens inverse. Elle produit d'abord le registre du reste de la procédure, puis ajoute l'étape courante à la fin du registre.

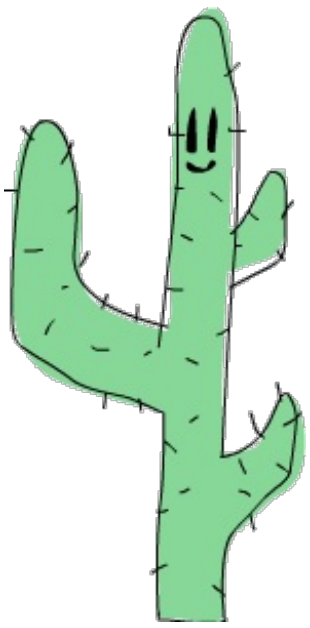
```
import Control.Monad.Writer

gcdReverse :: Int -> Int -> Writer [String] Int
gcdReverse a b
    | b == 0 = do
        tell ["Finished with " ++ show a]
        return a
    | otherwise = do
        result <- gcdReverse b (a `mod` b)
        tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
        return result
```

Elle effectue l'appel récursif d'abord, et lie le résultat au nom `result`. Puis, elle ajoute l'étape courante au registre, mais celle-ci vient donc s'ajouter à la fin du registre produit par l'appel récursif. Finalement, elle présente le résultat de l'appel récursif comme résultat final. La voici en action :

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcdReverse 8 3)
Finished with 1
2 mod 1 = 0
3 mod 2 = 1
8 mod 3 = 2
```

C'est inefficace parce qu'elle finit par associer les utilisations de `++` à gauche plutôt qu'à droite.

Listes différentielles



Puisque les listes peuvent parfois être inefficaces quand on concatène de façon répétée, il vaut mieux utiliser une structure de données qui supporte une concaténation toujours efficace. Une liste différentielle est une telle structure de données. Une liste différentielle est similaire à une liste, mais au lieu d'être une liste normale, c'est une fonction qui prend une liste et lui prépose une autre liste. La liste différentielle équivalente à la liste `[1, 2, 3]` serait la fonction `\xs -> [1, 2, 3] ++ xs`. Une liste vide normale est `[]`, alors qu'une liste différentielle vide est la fonction `\xs -> [] ++ xs`.

Le truc cool avec les listes différentielles, c'est qu'elles supportent une concaténation efficace. Quand on concatène deux listes normales avec `++`, elle doit traverser la liste de la gauche jusqu'à sa fin pour coller la liste de droite à cet endroit. Mais qu'en est-il avec l'approche des listes différentielles ? Eh bien, concaténer deux listes différentielles peut être fait ainsi :

```
f `append` g = \xs -> f (g xs)
```

Souvenez-vous, `f` et `g` sont des fonctions qui prennent une liste, et leur prépose quelque chose. Par exemple, si `f` est la fonction `("dog"++)` (qui est juste une autre façon d'écrire `\xs -> "dog" ++ xs`) et `g` est la fonction `("meat"++)`, alors `f `append` g` crée une nouvelle fonction équivalente à :

```
\xs -> "dog" ++ ("meat" ++ xs)
```

Nous avons concaténé deux listes différentielles simplement en en créant une nouvelle fonction qui applique d'abord une liste différentielle sur une liste, puis applique l'autre liste différentielle au résultat.

Créons un emballage `newtype` pour nos listes différentielles de façon à pouvoir les doter d'une instance de monoïde :

```
newtype DiffList a = DiffList { getDiffList :: [a] -> [a] }
```

Le type que l'on enveloppe est `[a] -> [a]` parce qu'une liste différentielle est simplement une fonction qui prend une liste et en retourne une autre. Convertir des listes normales en listes différentielles et vice versa est très facile :

```
toDiffList :: [a] -> DiffList a
toDiffList xs = DiffList (xs++)

fromDiffList :: DiffList a -> [a]
fromDiffList (DiffList f) = f []
```

Pour créer une liste normale à partir d'une liste différentielle, on fait comme on faisait avant en créant une fonction qui prépose une autre liste. Puisqu'une liste différentielle est une fonction qui prépose une autre liste à la liste qu'elle représente, pour obtenir cette liste, il suffit de l'appliquer à une liste vide !

Voici l'instance de `Monoid` :

```
instance Monoid (DiffList a) where
    mempty = DiffList (\xs -> [] ++ xs)
    (DiffList f) `mappend` (DiffList g) = DiffList (\xs -> f (g xs))
```

Remarquez comme pour ces listes, `mempty` est juste la fonction `id` et `mappend` est simplement la composition de fonctions. Voyons si cela marche :

```
ghci> fromDiffList (toDiffList [1,2,3,4] `mappend` toDiffList [1,2,3])
[1,2,3,4,1,2,3]
```

Tip top ! On peut maintenant améliorer l'efficacité de `gcdReverse` en lui faisant utiliser des listes différentielles plutôt que des listes normales :

```
import Control.Monad.Writer

gcd' :: Int -> Int -> Writer (DiffList String) Int
gcd' a b
    | b == 0 = do
        tell (toDiffList ["Finished with " ++ show a])
        return a
    | otherwise = do
        result <- gcd' b (a `mod` b)
        tell (toDiffList [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)])
        return result
```

On a simplement dû changer le type du monoïde de `[String]` en `DiffList String`, et changer nos listes normales en listes différentielles avec `toDiffList`

quand on utilisait `tell`. Voyons si le registre est assemblé correctement :

```
ghci> mapM_ putStrLn . fromDiffList . snd . runWriter $ gcdReverse 110 34
Finished with 2
8 mod 2 = 0
34 mod 8 = 2
110 mod 34 = 8
```

On fait `gcdReverse 110 34`, puis on utilise `runWriter` pour sortir le résultat du `newtype`, et on applique `snd` pour n'obtenir que le registre, puis on applique `fromDiffList` pour le convertir en une liste normale, et enfin on l'affiche entrée par entrée à l'écran.

Comparer les performances

Pour vous faire une idée de l'ordre de grandeur de l'amélioration des performances en utilisant les listes différentielles, considérez cette fonction qui décompte à partir d'un nombre jusqu'à zéro, et produit son registre dans le sens inverse, comme `gcdReverse`, de manière à ce que le registre compte les nombres dans l'ordre croissant :

```
finalCountDown :: Int -> Writer (DiffList String) ()
finalCountDown 0 = do
    tell (toDiffList ["0"])
finalCountDown x = do
    finalCountDown (x-1)
    tell (toDiffList [show x])
```

Si on lui donne `0`, elle l'écrit simplement dans le registre. Pour tout autre nombre, elle commence d'abord par décompter depuis son prédécesseur, jusqu'à `0` et enfin, ajoute ce nombre au registre. Ainsi, si l'on applique `finalCountDown` à `100`, la chaîne de caractères `"100"` sera la dernière du registre.

Si vous chargez cete fonction dans GHCi et que vous l'appliquez à un nombre gros, comme `500000`, vous verrez qu'elle compte rapidement depuis `0` vers l'avant :

```
ghci> mapM_ putStrLn . fromDiffList . snd . runWriter $ finalCountDown 500000
0
1
2
...
```

Cependant, si on la change pour utiliser des listes normales plutôt que des listes différentielles, de cette manière :

```
finalCountDown :: Int -> Writer [String] ()
finalCountDown 0 = do
    tell ["0"]
finalCountDown x = do
    finalCountDown (x-1)
    tell [show x]
```

Et qu'on demande à GHCi de compter :

```
ghci> mapM_ putStrLn . snd . runWriter $ finalCountDown 500000
```

On voit que le décompte est très lent.

Bien sûr, ce n'est pas la façon rigoureuse et scientifique de tester la vitesse de nos programmes, mais on peut déjà voir que dans ce cas, utiliser des listes différentielles commence à fournir des résultats immédiatement alors que pour des listes normales, cela prend une éternité.

Oh, au fait, la chanson Final Countdown d'Europe est maintenant coincée dans votre tête. De rien !

La lire ? Pas cette blague encore.

Dans le chapitre sur les foncteurs applicatifs, on a vu que le type des fonctions, `(->) r`, était une instance de `Functor`. Mapper une fonction `f` sur une fonction `g` crée une fonction qui prend la même chose que `g`, applique `g` dessus puis applique `f` au résultat. En gros, on crée une nouvelle fonction qui est comme `g`, mais qui applique `f` avant de renvoyer son résultat. Par exemple :

```
ghci> let f = (*5)
ghci> let g = (+3)
ghci> (fmap f g) 8
55
```

Nous avons aussi vu que les fonctions étaient des foncteurs applicatifs. Elles nous permettaient d'opérer sur les résultats à terme de fonctions comme si l'on avait déjà ces résultats. Par exemple :

```
ghci> let f = (+) <$> (*2) <*> (+10)
ghci> f 3
19
```

L'expression `(+) <$> (*2) <*> (+10)` crée une fonction qui prend un nombre, donne ce nombre à `(*2)` et à `(+10)`, et somme les résultats. Par exemple, si l'on applique cette fonction à `3`, elle applique à la fois `(*2)` et `(+10)` sur `3`, résultant en `6` et `13`. Puis, elle appelle `(+)` avec `6` et `13` et le résultat est `19`.

Non seulement le type des fonctions `(->) r` est un foncteur et un foncteur applicatif, mais c'est aussi une monade. Tout comme les autres valeurs monadiques que l'on a croisées jusqu'ici, une fonction peut être considérée comme une valeur dans un contexte. Le contexte dans le cas des fonctions est que la valeur n'est pas encore là, et qu'il faudra donc appliquer à la fonction sur quelque chose pour obtenir la valeur résultante.

Puisqu'on a déjà vu comment les fonctions fonctionnent comme des foncteurs et des foncteurs applicatifs, plongeons immédiatement dans le grand bassin et voyons l'instance de `Monad`. Elle est située dans `Control.Monad.Instances` et ressemble à ça :

```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w
```

Nous avons déjà vu comment `pure` était implémentée pour les fonctions, et `return` est la même chose que `pure`. Elle prend une valeur, et la place dans un contexte minimal contenant cette valeur comme résultat. Et le seul moyen de créer une fonction qui renvoie toujours le même résultat consiste à lui faire ignorer complètement son paramètre.

L'implémentation de `>>=` semble un peu plus cryptique, mais elle ne l'est pas tant que ça. Quand on utilisait `>>=` pour donner des valeurs monadiques à une fonction, le résultat était toujours une valeur monadique. Dans ce cas, si l'on donne une fonction à une autre fonction, le résultat est toujours une fonction. C'est pourquoi le résultat commence comme une lambda. Toutes les implémentations de `>>=` qu'on a vues jusqu'ici isolaient d'une certaine manière le résultat de la valeur monadique et lui appliquaient la fonction `f`. C'est la même chose ici. Pour obtenir le résultat d'une fonction, il faut l'appliquer à quelque chose, ce qu'on fait avec `(h w)`, puis on applique `f` au résultat. `f` retourne une valeur monadique, qui est une fonction dans notre cas, donc on l'applique également à `w`.

Si vous ne comprenez pas comment `>>=` marche ici, ne vous inquiétez pas, avec les exemples on va voir que c'est simplement une monade comme une autre. Voici une expression `do` qui utilise cette monade :

```
import Control.Monad.Instances

addStuff :: Int -> Int
addStuff = do
    a <- (*2)
    b <- (+10)
    return (a+b)
```

C'est la même chose que l'expression applicative qu'on a écrite plus haut, seulement maintenant elle se base sur le fait que les fonctions soient des monades. Une expression `do` résulte toujours en une valeur monadique, et celle-ci ne fait pas exception. Le résultat de cette valeur monadique est une fonction. Ce qui se passe ici, c'est qu'elle prend un nombre, puis fait `(*2)` sur ce nombre, ce qui résulte en `a`. `(+10)` est également appliquée au même nombre que celui donné à `(*2)`, et le résultat devient `b`. `return`, comme dans les autres monades, n'a pas d'autre effet que de créer une valeur monadique présentée en résultat. Elle présente ici `a + b` comme le résultat de cette fonction. Si on essaie, on obtient le même résultat qu'avant :

```
ghci> addStuff 3
19
```

`(*2)` et `(+3)` sont appliquées au nombre `3` dans ce cas. `return (a+b)` est également appliquée au nombre `3`, mais elle ignore ce paramètre et retourne toujours `a+b` en résultat. Pour cette raison, la monade des fonctions est aussi appelée la monade de lecture. Toutes les fonctions lisent en effet la même source. Pour illustrer cela encore mieux, on peut réécrire `addStuff` ainsi :

```
addStuff :: Int -> Int
addStuff x = let
    a = (*2) x
    b = (+10) x
    in a+b
```

On voit que la monade de lecture nous permet de traiter les fonctions comme des valeurs dans un contexte. On peut agir comme si l'on savait déjà ce que les fonctions retournaient. Ceci est réussi en collant toutes les fonctions ensemble et en donnant le paramètre de la fonction ainsi créée à toutes les fonctions qui la

composent. Ainsi, si l'on a plein de fonctions qui attendent toutes un même paramètre, on peut utiliser la monade de lecture pour extraire en quelque sorte leur résultat, et l'implémentation de `>>=` s'assurera que tout se passe comme prévu.

Calculs à états dans tous leurs états

Haskell est un langage pur, et grâce à cela, nos programmes sont faits de fonctions qui ne peuvent pas altérer un état global ou des variables, elles ne peuvent que faire des calculs et retourner des résultats. Cette restriction rend en fait plus simple la réflexion sur nos programmes, puisqu'elle nous libère de l'inquiétude de savoir quelle est la valeur de chaque variable à un instant donné. Cependant, certains problèmes sont intrinsèquement composés d'états en ce qu'ils se basent sur des états pouvant évoluer dans le temps. Bien que de tels problèmes ne soient pas un problème pour Haskell, ils peuvent parfois être fastidieux à modéliser. C'est pourquoi Haskell offre la monade d'états, qui rend la gestion de problèmes à états simple comme bonjour tout en restant propre et pur.

Lorsqu'on travaillait avec des nombres aléatoires, on manipulait des fonctions qui prenaient un générateur aléatoire en paramètre et retournaient un nombre aléatoire et un nouveau générateur aléatoire. Si l'on voulait générer plusieurs nombres aléatoires, nous avions toujours un générateur obtenu en résultat en même temps que le nombre aléatoire précédent. Quand on avait écrit une fonction prenant un `StdGen` et jetant trois fois une pièce en se basant sur un générateur, on avait fait :

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in  (firstCoin, secondCoin, thirdCoin)
```

Elle prenait un générateur `gen` et `random gen` retournait alors une valeur `Bool` ainsi qu'un nouveau générateur. Pour lancer la deuxième pièce, on utilisait le nouveau générateur, et ainsi de suite. Dans la plupart des autres langages, on n'aurait pas retourné un nouveau générateur avec le nombre aléatoire. On aurait simplement modifié le générateur existant ! Mais puisqu'Haskell est pur, on ne peut pas faire cela, donc nous devons prendre un état, créer à partir de celui-ci un résultat ainsi qu'un nouvel état, puis utiliser ce nouvel état pour générer d'autres résultats.

On pourrait se dire que pour éviter d'avoir à gérer manuellement ce genre de calculs à états, on devrait se débarrasser de la pureté d'Haskell. Eh bien, cela n'est pas nécessaire, puisqu'il existe une petite monade spéciale appelée la monade d'états qui s'occupe de toute cette manipulation d'états pour nous, et sans abandonner la pureté qui fait que programmer en Haskell est tellement cool.

Donc, pour nous aider à mieux comprendre le concept de calculs à états, donnons leur un type. On dira qu'un calcul à états est une fonction qui prend un état et retourne une valeur et un nouvel état. Le type de la fonction serait :

```
s -> (a,s)
```

`s` est le type de l'état et `a` le resultat du calcul à états.

L'affectation dans la plupart des autres langages pourrait être vue comme un calcul à états. Par exemple, quand on fait `x = 5` dans un langage impératif, cela assignera généralement la valeur `5` à la variable `x`, et l'expression elle-même aura aussi pour valeur `5`. Si vous imaginez cela fonctionnellement, vous pouvez le voir comme une fonction qui prend un état (ici, toutes les variables qui ont été affectées précédemment) et retourne un résultat (ici `5`) et un nouvel état, qui contiendrait toutes les valeurs précédentes des variables, ainsi que la variable fraîchement affectée.

Ce calcul à états, une fonction prenant un état et retournant un résultat et un état, peut aussi être vu comme une valeur dans un contexte. La valeur est le résultat, alors que le contexte est qu'on doit fournir un état initial pour pouvoir extraire la valeur, et qu'on retourne en plus du résultat un nouvel état.

Piles et rochers

Disons qu'on souhaite modéliser la manipulation d'une pile. Vous avez une pile de choses l'une sur l'autre, et vous pouvez soit ajouter quelque chose au sommet de la pile, soit enlever quelque chose du sommet de la pile. Quand on place quelque chose sur la pile, on dit qu'on l'empile, et quand on enlève quelque chose on dit qu'on le dépile. Si vous voulez quelque chose qui est tout en bas de la pile, il faut d'abord dépiler tout ce qui est au dessus.

Nous utiliserons une liste pour notre pile et la tête de la liste sera le sommet de la pile. Pour nous aider dans notre tâche, nous créerons deux fonctions : `pop` et `push`. `pop` prend une pile, dépile un élément, et retourne l'élément en résultat ainsi que la nouvelle pile sans cet élément. `push` prend un élément et une pile et empile l'élément sur la pile. Elle retourne `()` en résultat, ainsi qu'une nouvelle pile. Voici :

```
type Stack = [Int]

pop :: Stack -> (Int,Stack)
```

```
pop (x:xs) = (x,xs)

push :: Int -> Stack -> ((),Stack)
push a xs = ((),a:xs)
```

On utilise `()` comme résultat lorsque l'on empile parce qu'empiler un élément sur la pile n'a pas de résultat intéressant, son travail est simplement de changer la pile. Remarquez que si l'on applique que le premier paramètre de **`push`**, on obtient un calcul à états. **`pop`** est déjà un calcul à états de par son type.

Écrivons un petit bout de code qui simule une pile en utilisant ces fonctions. On va prendre une pile, empiler `3` et dépiler deux éléments, juste pour voir. Voici :

```
stackManip :: Stack -> (Int, Stack)
stackManip stack = let
    ((),newStack1) = push 3 stack
    (a ,newStack2) = pop newStack1
    in pop newStack2
```

On prend une pile **`stack`** et on fait **`push 3 stack`**, ce qui retourne un tuple. La première composante de cette paire est `()` et la seconde est la nouvelle pile, qu'on appelle **`newStack1`**. Ensuite, on dépile un nombre de **`newStack1`**, ce qui retourne un nombre **`a`** (qui est `3`) et une nouvelle pile qu'on appelle **`newStack2`**. Puis, on dépile un nombre de **`newStack2`** et obtient un nombre **`b`** et une nouvelle pile **`newStack3`**. Le tuple de ce nombre et cette pile est retourné. Essayons :

```
ghci> stackManip [5,8,2,1]
(5,[8,2,1])
```

Cool, le résultat est `5` et la nouvelle pile est `[8, 2, 1]`. Remarquez comme **`stackManip`** est elle-même un calcul à états. On a pris plusieurs calculs à états et on les a collés ensemble. Hmm, ça sonne familier.

Le code ci-dessus de **`stackManip`** est un peu fastidieux puisqu'on donne manuellement l'état à chaque calcul à états, puis on récupère un nouvel état qu'on donne à nouveau au prochain calcul. Ce serait mieux si, au lieu de donner les piles manuellement à chaque fonction, on pouvait écrire :

```
stackManip = do
    push 3
    a <- pop
    pop
```

Eh bien, avec la monade d'états c'est exactement ce qu'on écrira. Avec elle, on peut prendre des calculs à états comme ceux-ci et les utiliser sans se préoccuper de la gestion de l'état manuellement.

La monade State

Le module **`Control.Monad.State`** fournit un **`newtype`** qui enveloppe des calculs à états. Voici sa définition :

```
newtype State s a = State { runState :: s -> (a,s) }
```

Un **`State s a`** est un calcul à états qui manipule un état de type **`s`** et retourne un résultat de type **`a`**.

Maintenant qu'on a vu ce que sont des calculs à états et comment ils pouvaient être vus comme des valeurs avec des contextes, regardons leur instance de **`Monad`** :

```
instance Monad (State s) where
    return x = State $ \s -> (x,s)
    (State h) >>= f = State $ \s -> let (a, newState) = h s
                                        (State g) = f a
                                    in  g newState
```

Regardons d'abord **`return`**. Le but de **`return`** est de prendre une valeur et d'en faire un calcul à états retournant toujours cette valeur. C'est pourquoi on crée une lambda **`\s -> (x, s)`**. On présente toujours **`x`** en résultat du calcul à états et l'état est inchangé, parce que **`return`** place la valeur dans un contexte minimal. Donc **`return`** crée un calcul à états qui retourne une certaine valeur sans changer l'état.

Qu'en est-il de **`>>=`** ? Eh bien, le résultat obtenu en donnant une fonction à un calcul à états par **`>>=`** doit être un calcul à états, n'est-ce pas ? On commence donc à écrire le **`newtype State`** et une lambda. Cette lambda doit être notre nouveau calcul à états. Mais que doit-elle faire ? Eh bien, on doit extraire le résultat du premier calcul à états d'une manière ou d'une autre. Puisqu'on se trouve dans un calcul à états, on peut donner au calcul à états **`h`** notre état actuel **`s`**, ce qui retourne une paire d'un résultat et d'un nouvel état : **`(a, newState)`**. À chaque fois qu'on a implémenté **`>>=`**, après avoir extrait le résultat de la valeur monadique, on appliquait la fonction **`f`** dessus pour obtenir une nouvelle valeur monadique. Dans **`Writer`**, après avoir fait cela et

obtenu la nouvelle valeur monadique, on devait ne pas oublier de tenir compte du contexte en faisant **`mappend`** entre l'ancienne valeur monoïdale et la nouvelle. Ici, on fait **`f a`** et on obtient un nouveau calcul à états **`g`**. Maintenant qu'on a un calcul à états et un état (qui s'appelle **`newState`**) on applique simplement le calcul à états **`g`** à l'état **`newState`**. Le résultat est un tuple contenant le résultat final et l'état final !

Ainsi, avec **`>>=`**, on colle ensemble deux calculs à états, seulement le second est caché dans une fonction qui prend le résultat du premier. Puisque **`pop`** et **`push`** sont déjà des calculs à états, il est facile de les envelopper dans un **`State`**. Regardez :

```
import Control.Monad.State

pop :: State Stack Int
pop = State $ \(x:xs) -> (x,xs)

push :: Int -> State Stack ()
push a = State $ \xs -> ((),a:xs)
```

`pop` est déjà un calcul à états et **`push`** prend un **`Int`** et retourne un calcul à états. Maintenant, on peut réécrire l'exemple précédent où l'on empilait **`3`** sur la pile avant de dépiler deux nombres ainsi :

```
import Control.Monad.State

stackManip :: State Stack Int
stackManip = do
    push 3
    a <- pop
    pop
```

Voyez-vous comme on a collé ensemble un empilement et deux dépilements en un calcul à états ? Quand on sort ce calcul de son **`newtype`**, on obtient une fonction à laquelle on peut fournir un état initial :

```
ghci> runState stackManip [5,8,2,1]
(5,[8,2,1])
```

On n'avait pas eu besoin de lier le premier **`pop`** à **`a`** vu qu'on n'utilise pas ce **`a`**. On aurait pu écrire :

```
stackManip :: State Stack Int
stackManip = do
    push 3
    pop
    pop
```

Plutôt cool. Mais et si l'on voulait faire ceci : dépiler un nombre de la pile, puis si ce nombre est **`5`**, l'empiler à nouveau et sinon, empiler **`3`** et **`8`** plutôt ? Voici le code :

```
stackStuff :: State Stack ()
stackStuff = do
    a <- pop
    if a == 5
        then push 5
        else do
            push 3
            push 8
```

C'est plutôt simple. Lançons-la sur une pile vide.

```
ghci> runState stackStuff [9,0,2,1,0]
((),[8,3,0,2,1,0])
```

Souvenez-vous, les expressions **`do`** résultent en des valeurs monadiques, et avec la monade **`State`**, une expression **`do`** est donc une fonction à états. Puisque **`stackManip`** et **`stackStuff`** sont des calculs à états ordinaires, on peut les coller ensemble pour faire des calculs plus compliqués.

```
moreStack :: State Stack ()
moreStack = do
    a <- stackManip
    if a == 100
        then stackStuff
        else return ()
```

Si le résultat de `stackManip` sur la pile actuelle est `100`, on fait `stackStuff`, sinon on ne fait rien. `return ()` conserve l'état comme il est et ne fait rien.

Le module `Control.Monad.State` fournit une classe de types appelée `MonadState` qui contient deux fonctions assez utiles, j'ai nommé `get` et `put`. Pour `State`, la fonction `get` est implémentée ainsi :

```
get = State $ \s -> (s,s)
```

Elle prend simplement l'état courant et le présente en résultat. La fonction `put` prend un état et crée une fonction à états qui remplace l'état courant par celui-ci :

```
put newState = State $ \s -> ((),newState)
```

Avec ces deux fonctions, on peut voir la pile courante ou la remplacer par une toute nouvelle pile. Comme ça :

```
stackyStack :: State Stack ()
stackyStack = do
    stackNow <- get
    if stackNow == [1,2,3]
        then put [8,3,1]
        else put [9,2,1]
```

Il est intéressant d'examiner le type qu'aurait `>>=` si elle était restreinte aux valeurs `State` :

```
(>>=) :: State s a -> (a -> State s b) -> State s b
```

Remarquez que le type de l'état `s` reste le même, mais le type du résultat peut changer de `a` en `b`. Cela signifie que l'on peut coller ensemble des calculs à états dont les résultats sont de différents types, mais le type des états doit être le même. Pourquoi cela ? Eh bien, par exemple, pour `Maybe`, `>>=` a ce type :

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Il est logique que la monade elle-même, `Maybe`, ne change pas. Ça n'aurait aucun sens d'utiliser `>>=` entre deux monades différentes. Eh bien, pour la monade d'états, la monade est en fait `State s`, donc si ce `s` était différent, on utiliserait `>>=` entre deux monades différentes.

Aléatoire et monade d'états

Au début de la section, on a vu que générer des nombres aléatoires pouvait parfois sembler bizarre puisque chaque fonction aléatoire prenait un générateur et retourne un nombre aléatoire ainsi qu'un nouveau générateur, qui devait ensuite être utilisé à la place du précédent pour générer de nouveaux nombres aléatoires. La monade d'états rend cela bien plus facile.

La fonction `random` de `System.Random` a pour type :

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

Signifiant qu'elle prend un générateur aléatoire et produit un nombre aléatoire et un nouveau générateur. On peut voir que c'est un calcul à états, on peut donc l'envelopper dans le constructeur de `newtype State` et l'utiliser comme une valeur monadique afin que la gestion de l'état soit faite pour nous :

```
import System.Random
import Control.Monad.State

randomSt :: (RandomGen g, Random a) => State g a
randomSt = State random
```

À présent, si l'on souhaite lancer trois pièces (`True` pour pile, `False` pour face), on peut faire :

```
import System.Random
import Control.Monad.State

threeCoins :: State StdGen (Bool,Bool,Bool)
threeCoins = do
    a <- randomSt
    b <- randomSt
    c <- randomSt
    return (a,b,c)
```

`threeCoins` est un calcul à états et après avoir pris un générateur aléatoire initial, il le passe au premier `randomSt`, qui produit un nombre et un nouveau

générateur, qui est passé au prochain et ainsi de suite. On utilise `return (a, b, c)` pour présenter `(a, b, c)` comme le résultat sans changer le générateur le plus récent. Essayons :

```
ghci> runState threeCoins (mkStdGen 33)
((True,False,True),680029187 2103410263)
```

Erreur, erreur, ma belle erreur

On sait à présent que `Maybe` est utilisé pour ajouter un contexte d'échec possible à des valeurs. Une valeur peut être `Just something` ou `Nothing`. Aussi utile que cela puisse être, quand on a un `Nothing`, tout ce qu'on sait c'est qu'il y a eu une sorte d'erreur, mais il n'y a pas de moyen d'en savoir plus sur le type de l'erreur et sa raison.

Le type `Either e a` au contraire, nous permet d'incorporer un contexte d'échec éventuel à nos valeurs tout en étant capable d'attacher des valeurs aux échecs, afin de décrire ce qui s'est mal passé et de fournir d'autres informations intéressantes concernant l'échec. Une valeur `Either e a` peut être ou bien une valeur `Right`, indiquant la bonne réponse et un succès, ou une valeur `Left`, indiquant l'échec. Par exemple :

```
ghci> :t Right 4
Right 4 :: (Num t) => Either a t
ghci> :t Left "out of cheese error"
Left "out of cheese error" :: Either [Char] b
```

C'est simplement un `Maybe` avancé, il est donc logique que ce soit une monade, parce qu'elle peut aussi être vue comme une valeur avec un contexte additionnel d'échec éventuel, seulement maintenant il y a une valeur attachée à cette erreur.

Son instance de `Monad` est similaire à celle de `Maybe` et peut être trouvée dans `Control.Monad.Error` :

```
instance (Error e) => Monad (Either e) where
    return x = Right x
    Right x >>= f = f x
    Left err >>= f = Left err
    fail msg = Left (strMsg msg)
```

Comme toujours, `return` prend une valeur et la place dans un contexte par défaut minimal. Elle enveloppe notre valeur dans le constructeur `Right` parce qu'on utilise `Right` pour représenter un calcul réussi pour lequel un résultat est présent. C'est presque comme le `return` de `Maybe`.

`>>=` examine deux cas possibles : un `Left` ou un `Right`. Dans le cas d'un `Right`, la fonction `f` est appliquée à la valeur à l'intérieur, comme pour `Just`. Dans le cas d'une erreur, la valeur `Left` est conservée ainsi que son contenu qui décrit l'erreur.

L'instance de `Monad` d'`Either e a` a un pré-requis additionnel, qui est que le type de la valeur contenue dans `Left`, celui indexé par le paramètre de type `e`, doit être une instance de la classe de types `Error`. La classe de types `Error` est pour les types dont les valeurs peuvent être vues comme des messages d'erreur. Elle définit une fonction `strMsg`, qui prend une erreur sous la forme d'une chaîne de caractères et retourne une telle valeur. Un bon exemple d'instance d'`Error` est, eh bien, le type `String` ! Dans le cas de `String`, la fonction `strMsg` retourne simplement ce qu'elle a reçu :

```
ghci> :t strMsg
strMsg :: (Error a) => String -> a
ghci> strMsg "boom!" :: String
"boom!"
```

Puisqu'on utilise généralement des `String` pour décrire nos erreurs quand on utilise `Either`, on n'a pas trop à s'en soucier. Quand un filtrage par motif échoue dans la notation `do`, une valeur `Left` est utilisée pour indiquer cet échec.

Voici quelques exemples d'utilisation :

```
ghci> Left "boom" >>= \x -> return (x+1)
Left "boom"
ghci> Right 100 >>= \x -> Left "no way!"
Left "no way!"
```

Quand on utilise `>>=` pour donner une valeur `Left` à une fonction, la fonction est ignorée et une valeur `Left` identique est retournée. Quand on donne une valeur `Right` à une fonction, la fonction est appliquée sur ce qui est à l'intérieur du `Right`, mais dans ce cas, la fonction produit quand même une valeur `Left` !

Quand on donne une valeur `Right` à une fonction qui réussit également, on obtient une erreur de type curieuse ! Hmmm.

```
ghci> Right 3 >>= \x -> return (x + 100)
```

```
<interactive>:1:0:
    Ambiguous type variable `a' in the constraints:
      `Error a' arising from a use of `it' at <interactive>:1:0-33
      `Show a' arising from a use of `print' at <interactive>:1:0-33
    Probable fix: add a type signature that fixes these type variable(s)
```

Haskell dit qu'il ne sait pas quel type choisir pour la partie `e` de notre valeur `Either e a`, bien qu'on n'ait seulement affiché la partie `Right`. Ceci est dû à la contrainte `Error e` de l'instance de `Monad`. Ainsi, si vous obtenez une telle erreur de type en utilisant la monade `Either`, ajoutez une signature de type explicite :

```
ghci> Right 3 >>= \x -> return (x + 100) :: Either String Int
Right 103
```

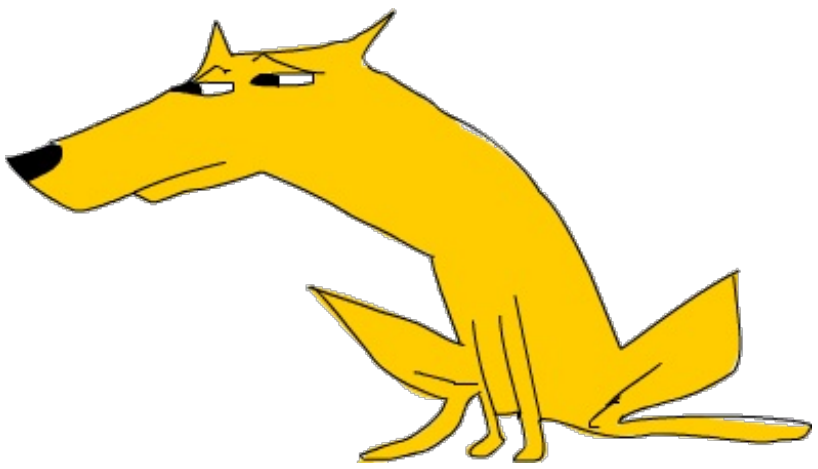
Parfait, cela fonctionne à présent !

À part ce petit écueil, utiliser cette monade est très similaire à l'utilisation de la monade `Maybe`. Dans le chapitre précédent, on utilisait les aspect monadiques de `Maybe` pour simuler l'atterrissage d'oiseaux sur la perche d'un funambule. En tant qu'exercice, vous pouvez réécrire cela avec la monade d'erreur afin que lorsque le funambule glisse et tombe, on sache combien il y avait d'oiseaux de chaque côté de la perche à l'instant de la chute.

Quelques fonctions monadiques utiles

Dans cette section, on va explorer quelques fonctions qui opèrent sur des valeurs monadiques ou retournent des valeurs monadiques (ou les deux à la fois !). De telles fonctions sont dites monadiques. Bien que certaines d'entre elles seront toutes nouvelles, d'autres ne seront que les équivalents monadiques de fonctions que l'on connaissait déjà, comme `filter` ou `foldl`. Voyons donc de qui il s'agit !

liftM et ses amies



Alors que nous débutions notre périple vers le sommet du Mont Monade, on a d'abord vu des foncteurs, qui étaient pour les choses sur lesquelles on pouvait mapper. Puis, on a appris qu'il existait des foncteurs améliorés appelés foncteurs applicatifs, qui nous permettaient d'appliquer des fonctions normales sur plusieurs valeurs applicatives ainsi que de prendre une valeur normale et de la placer dans un contexte par défaut. Finalement, on a introduit les monades comme des foncteurs applicatifs améliorés, qui ajoutaient la possibilité de donner ces valeurs avec des contextes à des fonctions normales.

Ainsi, chaque monade est un foncteur applicatif, et chaque foncteur applicatif est un foncteur. La classe de types `Applicative` a une contrainte de classe telle que notre type doit être une instance de `Functor` avant de pouvoir devenir une instance d'`Applicative`. Mais bien que `Monad` devrait avoir la même contrainte de classe pour `Applicative`, puisque chaque monade est un foncteur applicatif, elle ne l'a pas, parce que la classe de types `Monad` a été introduite en Haskell longtemps avant `Applicative`.

Mais bien que chaque monade soit un foncteur, on n'a pas besoin que son type soit une instance de `Functor` grâce à la fonction `liftM`. `liftM` prend une fonction et une valeur monadique et mappe la fonction sur la valeur. C'est donc comme `fmap` ! Voici le type de `liftM` :

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
```

Et le type de `fmap` :

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

Si un type est à la fois instance de `Functor` et de `Monad` et obéit leurs lois respectives, alors ces deux fonctions doivent être identiques (c'est le cas pour toutes les monades qu'on a vues jusqu'ici). Un peu comme `pure` et `return` font la même chose, seulement que la première a une contrainte de classe `Applicative` alors que l'autre a une contrainte `Monad`. Testons `liftM` :

```
ghci> liftM (*3) (Just 8)
Just 24
ghci> fmap (*3) (Just 8)
Just 24
ghci> runWriter $ liftM not $ Writer (True, "chickpeas")
(False,"chickpeas")
ghci> runWriter $ fmap not $ Writer (True, "chickpeas")
(False,"chickpeas")
ghci> runState (liftM (+100) pop) [1,2,3,4]
(101,[2,3,4])
ghci> runState (fmap (+100) pop) [1,2,3,4]
```

```
(101,[2,3,4])
```

On sait déjà comment `fmap` fonctionne avec les valeurs `Maybe`. Et `liftM` est identique. Pour les valeurs `Writer`, la fonction est mappée sur la première composante du tuple, qui est le résultat. Faire `fmap` ou `liftM` sur un calcul à états résulte en un autre calcul à états, mais son résultat final sera modifié par la fonction passée en argument. Si l'on n'avait pas mappé `(+100)` sur `pop`, elle aurait retourné `(1,[2,3,4])`.

Voici comment `liftM` est implémentée :

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = m >>= (\x -> return (f x))
```

Ou, en notation `do` :

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = do
    x <- m
    return (f x)
```

On donne la valeur monadique `m` à la fonction, et on applique `f` à son résultat avant de le retourner dans un contexte par défaut. Grâce aux lois des monades, on a la garantie que le contexte est inchangé, seule la valeur résultante est altérée. On voit que `liftM` est implémentée sans faire référence à la classe de types `Functor`. Cela signifie qu'on a pu implémenter `fmap` (ou `liftM`, peu importe son nom) en utilisant simplement ce que les monades nous offraient. Ainsi, on peut conclure que les monades sont plus fortes que les foncteurs normaux.

La classe de types `Applicative` nous permet d'appliquer des fonctions entre des valeurs dans des contextes comme si elles étaient des valeurs normales. Comme cela :

```
ghci> (+) <$> Just 3 <*> Just 5
Just 8
ghci> (+) <$> Just 3 <*> Nothing
Nothing
```

Utiliser ce style applicatif rend les choses plutôt faciles. `<$>` est juste `fmap` et `<*>` est une fonction de la classe de types `Applicative` qui a pour type :

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

C'est donc un peu comme `fmap`, seulement la fonction elle-même est dans un contexte. Il faut d'une certaine façon l'extraire de ce contexte et la mapper sur la valeur `f a`, puis restaurer le contexte. Grâce à la curryfication par défaut en Haskell, on peut utiliser la combinaison de `<$>` et `<*>` pour appliquer des fonctions qui prennent plusieurs paramètres entre plusieurs valeurs applicatives.

Il s'avère que tout comme `fmap`, `<*>` peut aussi être implémentée uniquement avec ce que la classe de types `Monad` nous donne. La fonction `ap` est simplement `<*>`, mais avec une contrainte de classe `Monad` plutôt qu'`Applicative`. Voici sa définition :

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap mf m = do
    f <- mf
    x <- m
    return (f x)
```

`mf` est une valeur monadique dont le résultat est une fonction. Puisque la fonction est dans un contexte comme la valeur, on récupère la fonction de son contexte et on l'appelle `f`, puis on récupère la valeur qu'on appelle `x` et finalement on applique la fonction avec la valeur et on présente le résultat. Voici un exemple rapide :

```
ghci> Just (+3) <*> Just 4
Just 7
ghci> Just (+3) `ap` Just 4
Just 7
ghci> [(+1),(+2),(+3)] <*> [10,11]
[11,12,12,13,13,14]
ghci> [(+1),(+2),(+3)] `ap` [10,11]
[11,12,12,13,13,14]
```

On voit à présent que les monades sont plus fortes que les foncteurs applicatifs, puisqu'on peut utiliser les fonctions de `Monad` pour implémenter celles d'`Applicative`. En fait, très souvent, lorsqu'un type s'avère être une monade, les gens écrivent d'abord l'instance de `Monad`, puis créent une instance d'`Applicative` en disant que `pure` est `return` et `<*>` est `ap`. De façon similaire, si vous avec une instance de `Monad`, vous pouvez faire une instance de

`Functor` en disant que `fmap` est `liftM`.

La fonction `liftA2` est pratique pour appliquer une fonction entre deux valeurs applicatives. Elle est simplement définie comme :

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f x y = f <$> x <*> y
```

La fonction `liftM2` fait la même chose, mais avec une contrainte `Monad`. Il existe également `liftM3`, `liftM4` et `liftM5`.

Nous avons vu que les monades étaient plus puissantes que les foncteurs applicatifs et les foncteurs et que bien que toutes les monades soient des foncteurs applicatifs et des foncteurs, elles n'ont pas nécessairement d'instances de `Functor` et `Applicative`, et on a donc vu les fonctions équivalentes à celles qu'on utilisait sur nos foncteurs et nos foncteurs applicatifs.

La fonction `join`

Voici de quoi faire travailler vos neurones : si le résultat d'une valeur monadique est une autre valeur monadique, autrement dit si une valeur monadique est imbriquée dans une autre, peut-on les aplatir en une valeur monadique simple ? Par exemple, si l'on a `Just (Just 9)`, peut-on en faire un `Just 9` ? Il s'avère que toute valeur monadique imbriquée peut être aplatie et ceci est une propriété unique des monades. Pour cela, la fonction `join` existe. Son type est :

```
join :: (Monad m) => m (m a) -> m a
```

Ainsi, elle prend une valeur monadique dans une valeur monadique et retourne simplement une valeur monadique, donc en quelque sorte elle l'aplatit. La voici en action sur diverses valeurs `Maybe` :

```
ghci> join (Just (Just 9))
Just 9
ghci> join (Just Nothing)
Nothing
ghci> join Nothing
Nothing
```

La première ligne contient un calcul réussi en résultat d'un calcul réussi, donc ils sont joints en un gros calcul réussi. La deuxième ligne contient un `Nothing` comme résultat dans un `Just`. À chaque fois qu'on a eu affaire à des valeurs `Maybe` auparavant et qu'on voulait en combiner plusieurs, que ce soit avec `<*>` ou `>>=`, elles devaient toutes être `Just` pour que le résultat soit `Just`. S'il y avait un seul échec parmi elles, le résultat était un échec. Il en va de même ici. À la troisième ligne, on essaie d'aplatir ce qui est déjà un échec, le résultat reste un échec.

Aplatir les listes est intuitif :

```
ghci> join [[1,2,3],[4,5,6]]
[1,2,3,4,5,6]
```

Comme vous le voyez, `join` est juste `concat`. Pour aplatir une valeur `Writer` donc le résultat est une valeur `Writer`, on doit `mappend` les valeurs monoïdales.

```
ghci> runWriter $ join (Writer (Writer (1,"aaa"),"bbb"))
(1,"bbbaaa")
```

Le valeur monoïdale extérieure `"bbb"` vient d'abord, puis `"aaa"` est juxtaposée. Intuitivement, pour examiner la valeur d'un `Writer`, il faut que sa valeur monoïdale soit mise au registre d'abord, et seulement ensuite peut-on examiner ce qu'elle contient.

Aplatir des valeurs `Either` est similaire au cas des valeurs `Maybe` :

```
ghci> join (Right (Right 9)) :: Either String Int
Right 9
ghci> join (Right (Left "error")) :: Either String Int
Left "error"
ghci> join (Left "error") :: Either String Int
Left "error"
```

Si on applique `join` à un calcul à états dont le résultat est un calcul à états, le résultat est un calcul à états qui lance d'abord le calcul extérieur et ensuite le calcul intérieur. Regardez :

```
ghci> runState (join (State $ \s -> (push 10,1:2:s))) [0,0,0]
((),[10,1,2,0,0,0])
```

La lambda ici prend un état et place `2` et `1` dans la pile et présente `push 10` comme son résultat. Quand cette chose est aplatie avec `join` et évaluée, elle empile d'abord `2` et `1` puis `push 10` est exécuté, empilant un `10` en sommet de pile.

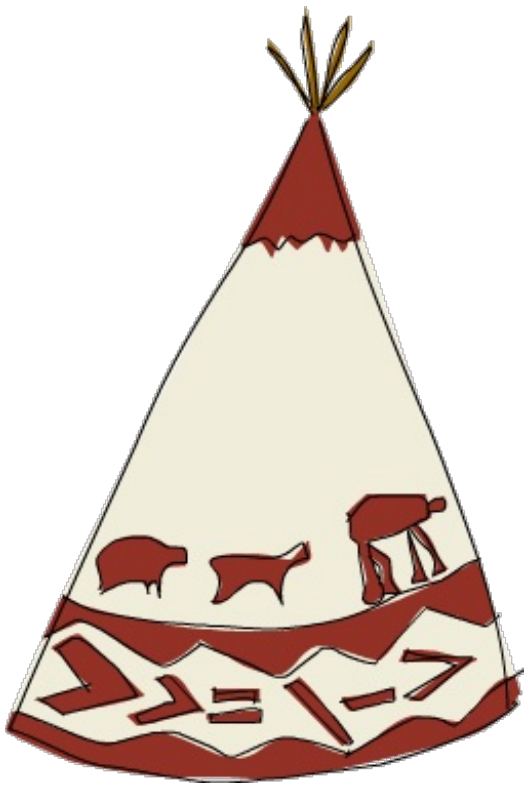
L'implémentation de `join` est comme suit :

```
join :: (Monad m) => m (m a) -> m a
join mm = do
    m <- mm
    m
```

Puisque le résultat de `mm` est une valeur monadique, on obtient ce résultat et on le place ensuite sur sa propre ligne, comme une valeur monadique. L'astuce ici est que lorsqu'on fait `m <- mm`, le contexte de la monade en question est pris en compte. C'est pourquoi, par exemple, des valeurs `Maybe` résultent en des `Just` seulement si les valeurs intérieures et extérieures sont toutes deux des valeurs `Just`. Voici à quoi cela ressemblait si `mm` était fixé à l'avance à la valeur `Just (Just 8)` :

```
joinedMaybes :: Maybe Int
joinedMaybes = do
    m <- Just (Just 8)
    m
```

Peut-être que la chose la plus intéressante à propos de `join` est que, pour chaque monade, donner une valeur monadique à une fonction avec `>>=` est équivalent à mapper cette fonction sur la valeur puis utiliser `join` pour aplatir la valeur monadique imbriquée résultante ! En d'autres termes, `m >>= f` est toujours égal à `join (fmap f m)` ! C'est logique quand on y réfléchit. Avec `>>=`, on donne une valeur monadique à une fonction qui attend une valeur normale et retourne une valeur monadique. Si l'on mappe cette fonction sur la valeur monadique, on a une valeur monadique à l'intérieur d'une valeur monadique. Par exemple, si l'on a `Just 9` et la fonction `\x -> Just (x + 1)`. En mappant la fonction sur `Just 9`, on obtient `Just (Just 10)`.



Le fait que `m >>= f` est toujours égal à `join (fmap f)` est très utile quand on crée nos propres instances de `Monad` pour certains types parce qu'il est souvent plus facile de voir comment on aplatirait une valeur monadique imbriquée plutôt que de trouver comment implémenter `>>=`.

filterM

La fonction `filter` est le pain quotidien de la programmation en Haskell (`map` étant le beurre sur la tartine). Elle prend un prédicat et une liste à filtrer et retourne une nouvelle liste dans laquelle tous les éléments satisfaisant le précidat ont été gardés. Son type est :

```
filter :: (a -> Bool) -> [a] -> [a]
```

Le prédicat prend un élément de la liste et retourne une valeur `Bool`. Et si la valeur `Bool` retournée était une valeur monadique ? Ouah ! C'est-à-dire, et si elle venait avec son contexte ? Est-ce que cela marcherait ? Par exemple, si chaque valeur `True` ou `False` que le prédicat produisait était accompagnée d'une valeur monoïdale, comme `["Accepted the number 5"]` ou `["3 is too small"]` ? On dirait que ça pourrait marcher. Si c'était le cas, on s'attendrait à ce que la liste résultante vienne également avec un registre combinant tous les registres obtenus en route. Donc, si le `Bool` retourné par le prédicat vient avec un contexte, on s'attendrait à ce que la liste résultante ait également un contexte attaché, autrement, le contexte de chaque `Bool` serait perdu.

La fonction `filterM` de `Control.Monad` fait exactement ce que l'on souhaite ! Son type est :

```
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

Le prédicat retourne une valeur monadique dont le résultat est un `Bool`, mais puisque c'est une valeur monadique, son contexte peut-être n'importe quoi, un possible échec, du non déterminisme, et encore plus ! Pour s'assurer que le contexte est reflété dans le résultat final, celui-ci est aussi une valeur monadique.

Prenons une liste et gardons seulement les valeurs inférieures à 4. Pour commencer, on va utiliser la fonction `filter` ordinaire :

```
ghci> filter (\x -> x < 4) [9,1,5,2,10,3]
[1,2,3]
```

C'était plutôt simple. Maintenant, créons un prédicat qui, en plus de retourner `True` ou `False`, fournisse également un registre de ce qu'il a fait. Bien sûr, on va utiliser la monade `Writer` à cet effet :

```
keepSmall :: Int -> Writer [String] Bool
```

```
keepSmall x
    | x < 4 = do
        tell ["Keeping " ++ show x]
        return True
    | otherwise = do
        tell [show x ++ " is too large, throwing it away"]
        return False
```

Plutôt que de seulement retourner un `Bool`, cette fonction retourne un `Writer [String] Bool`. C'est un prédicat monadique. Ça sonne classe, non ? Si le nombre est plus petit que `4`, on indique qu'on le garde et on `return True`.

À présent, passons-la à `filterM` avec une liste. Puisque le prédicat retourne une valeur `Writer`, la liste résultante sera également une valeur `Writer`.

```
ghci> fst $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
[1,2,3]
```

En examinant le résultat de la valeur `Writer` résultante, on voit que tout se déroule bien. Maintenant, affichons le registre pour voir ce qu'il s'est passé :

```
ghci> mapM_ putStrLn $ snd $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
9 is too large, throwing it away
Keeping 1
5 is too large, throwing it away
Keeping 2
10 is too large, throwing it away
Keeping 3
```

Génial. Donc simplement en fournissant un prédicat monadique à `filterM`, nous avons pu filtrer une liste en tirant profit du contexte monadique utilisé.

Une astuce Haskell très cool est d'utiliser `filterM` pour obtenir l'ensemble des parties d'une liste (en imaginant la liste comme un ensemble pour le moment). L'ensemble des parties d'un ensemble est l'ensemble des sous-ensembles de cet ensemble. Si l'on a un ensemble comme `[1, 2, 3]`, l'ensemble de ses parties contient les ensembles suivants :

```
[1,2,3]
[1,2]
[1,3]
[1]
[2,3]
[2]
[3]
[]
```

En d'autres termes, obtenir l'ensemble des parties d'un ensemble consiste à trouver toutes les combinaisons possibles de garder ou jeter les éléments de cet ensemble. `[2, 3]` est comme l'ensemble original, mais on a exclu le nombre `1`.

Pour créer une fonction renvoyant l'ensemble des parties d'une liste, on va s'appuyer sur le non déterminisme. On prend une liste `[1, 2, 3]` et on regarde son premier élément, qui est `1`, et on se demande : devrait-on le garder ou le jeter ? Eh bien, on aimerait faire les deux en réalité. On va donc filtrer une liste à l'aide d'un prédicat qui gardera et jettera chaque élément de façon non déterministe. Voici notre fonction des sous-parties d'un ensemble :

```
powerset :: [a] -> [[a]]
powerset xs = filterM (\x -> [True, False]) xs
```

Quoi, c'est tout ? Yup. On choisit de jeter et de garder chaque élément, peu importe lequel c'est. Nous avons un prédicat non déterministe, donc la liste résultante sera aussi une valeur non déterministe, et donc une liste de listes. Essayons :

```
ghci> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

Il faut un peu y réfléchir pour comprendre ce qui se passe, mais si vous considérez simplement les listes comme des valeurs non déterministes qui ne savent pas ce qu'elles valent et choisissent donc de tout valoir à la fois, c'est un peu plus simple.

foldM

L'équivalent monadique de `foldl` est `foldM`. Si vous vous souvenez de vos plis de la section sur les plis, vous savez que `foldl` prend une fonction binaire, un accumulateur initial et une liste à plier, et plie la liste en partant de la gauche avec la fonction binaire pour n'en faire plus qu'une valeur. `foldM` fait la même chose, mais prend une fonction qui produit une valeur monadique pour plier la liste. Sans surprise, la valeur résultante est également monadique. Le type de `foldl` est :

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Alors que `foldM` a pour type :

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
```

La valeur que la fonction binaire retourne est monadique, et donc le résultat du pli entier est aussi monadique. Sommons une liste de nombres à l'aide d'un pli :

```
ghci> foldl (\acc x -> acc + x) 0 [2,8,3,1]
14
```

L'accumulateur initial est `0`, puis `2` est ajouté à l'accumulateur, résultant en une valeur de `2`. `8` est ensuite ajouté, résultant en un accumulateur valant `10` et ainsi de suite jusqu'à atteindre la fin, l'accumulateur final étant le résultat.

Et si l'on voulait sommer une liste de nombres avec la condition supplémentaire que si l'un des nombres de la liste est plus grande que `9`, tout échoue ? Il semble logique d'utiliser une fonction binaire qui vérifie si le nombre à ajouter est plus grand que `9`, échoue le cas échéant, et continue son petit bonhomme de chemin si ce n'est pas le cas. À cause de cette possibilité d'échec additionnelle, faisons retourner à notre fonction un `Maybe` accumulateur plutôt qu'un accumulateur normal. Voici la fonction binaire :

```
binSmalls :: Int -> Int -> Maybe Int
binSmalls acc x
    | x > 9     = Nothing
    | otherwise = Just (acc + x)
```

Puisque notre fonction binaire est maintenant une fonction monadique, on ne peut plus utiliser le `foldl` normal, mais on doit utiliser `foldM`. Voici :

```
ghci> foldM binSmalls 0 [2,8,3,1]
Just 14
ghci> foldM binSmalls 0 [2,11,3,1]
Nothing
```

Excellent ! Puisqu'un des nombres de la liste est plus grand que `9`, le tout résulte en `Nothing`. Plier avec une fonction binaire qui retourne une valeur `Writer` est aussi cool parce que vous pouvez enregistrer ce que vous voulez pendant que le pli suit son chemin.

Créer une calculatrice NPI sécurisée

Quand nous résolvions le problème de l'implémentation d'une calculatrice NPI, nous avions remarqué qu'elle fonctionnait bien tant que l'entrée était bien formée. Mais s'il se passait quelque chose de travers, notre programme entier plantait. À présent que l'on sait prendre un code existant et le rendre monadique, reprenons notre calculatrice NPI et ajoutons-lui une gestion d'erreur en profitant de la monade `Maybe`.

Nous avions implémenté notre calculatrice NPI en prenant une chaîne de caractères comme `"1 3 + 2 *"`, en la découpant en mots pour obtenir quelque chose comme `["1","3","+","2","*"]`, puis en pliant cette liste en commençant avec une pile vide et en utilisant une fonction binaire de pli qui empile les nombres ou manipule ceux au sommet de la pile pour les ajouter ou les diviser, etc.

Ceci était le corps de notre fonction principale :

```
import Data.List

solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
```

On transformait l'expression en une liste de chaînes de caractères, qu'on pliait avec notre fonction de pli, et il ne nous restait plus qu'un élément dans la pile, qu'on retournait comme réponse. La fonction de pli était :

```
foldingFunction :: [Double] -> String -> [Double]
foldingFunction (x:y:ys) "*" = (x * y):ys
foldingFunction (x:y:ys) "+" = (x + y):ys
foldingFunction (x:y:ys) "-" = (y - x):ys
foldingFunction xs numberString = read numberString:xs
```

L'accumulateur du pli était une pile, qu'on représentait comme une liste de valeurs `Double`. Tandis que la fonction de pli traversait l'expression en NPI, si l'élément

en cours était un opérateur, elle prenait deux éléments en haut de la pile, appliquait l'opérateur sur ceux-ci et replaçait le résultat sur la pile. Si l'élément en cours était une chaîne représentant un nombre, elle convertissait cette chaîne en un vrai nombre et retournait une nouvelle pile comme l'ancienne, mais avec ce nombre empilé.

Rendons d'abord notre fonction de pli capable d'échouer gracieusement. Son type va changer de ce qu'il était en :

```
foldingFunction :: [Double] -> String -> Maybe [Double]
```

Ainsi, soit elle retournera `Just` une nouvelle pile, soit elle échouera avec la valeur `Nothing`.

La fonction `reads` est comme `read`, seulement elle retourne une liste avec un unique élément en cas de lecture réussie. Si elle échoue à lire quelque chose, alors elle retourne une liste vide. À part retourner la valeur qu'elle a lue, elle retourne également le morceau de chaîne de caractères qu'elle n'a pas consommé. On va dire qu'il faut qu'elle consomme l'entrée en entier pour que cela marche, et on va créer une fonction `readMaybe` pour notre convenance. La voici :

```
readMaybe :: (Read a) => String -> Maybe a
readMaybe st = case reads st of [(x,"")] -> Just x
                                _ -> Nothing
```

Testons :

```
ghci> readMaybe "1" :: Maybe Int
Just 1
ghci> readMaybe "GO TO HELL" :: Maybe Int
Nothing
```

Ok, ça a l'air de marcher. Donc, transformons notre fonction de pli en une fonction monadique pouvant échouer :

```
foldingFunction :: [Double] -> String -> Maybe [Double]
foldingFunction (x:y:ys) "*" = return ((x * y):ys)
foldingFunction (x:y:ys) "+" = return ((x + y):ys)
foldingFunction (x:y:ys) "-" = return ((y - x):ys)
foldingFunction xs numberString = liftM (:xs) (readMaybe numberString)
```

Les trois premiers cas sont comme les anciens, sauf que la nouvelle pile est enveloppée dans un `Just` (on a utilisé `return` pour cela, mais on aurait tout aussi bien pu écrire `Just`). Dans le dernier cas, on fait `readMaybe numberString` et on mappe ensuite `(:xs)` dessus. Donc, si la pile `xs` est `[1.0, 2.0]` et si `readMaybe numberString` résulte en `Just 3.0`, le résultat est `Just [3.0, 1.0, 2.0]`. Si `readMaybe numberString` résulte en `Nothing`, alors le résultat est `Nothing`. Essayons la fonction de pli seule :

```
ghci> foldingFunction [3,2] "*"
Just [6.0]
ghci> foldingFunction [3,2] "-"
Just [-1.0]
ghci> foldingFunction [] "*"
Nothing
ghci> foldingFunction [] "1"
Just [1.0]
ghci> foldingFunction [] "1 wawawawa"
Nothing
```

Ça a l'air de marcher ! Il est l'heure d'introduire notre nouvelle fonction `solveRPN` améliorée. Mesdames, mesdemoiselles et messieurs !

```
import Data.List

solveRPN :: String -> Maybe Double
solveRPN st = do
    [result] <- foldM foldingFunction [] (words st)
    return result
```

Tout comme avant, on prend une chaîne de caractères et on en fait une liste de mots. Puis, on fait un pli, démarrant avec une pile vide, seulement au lieu d'un `foldl` normal, on fait un `foldM`. Le résultat de ce `foldM` doit être une valeur `Maybe` contenant une liste (notre pile finale) et cette liste ne doit contenir qu'une valeur. On utilise une expression `do` pour obtenir cette valeur et on l'appelle `result`. Si `foldM` retourne `Nothing`, le tout vaudra `Nothing`, parce que c'est comme cela que `Maybe` fonctionne. Remarquez aussi qu'on filtre par motif dans l'expression `do`, donc si la liste a plus d'une valeur, ou bien aucune, le filtrage par motif échoue et `Nothing` est produit. À la dernière ligne, on fait simplement `return result` pour présenter le résultat du calcul NPI comme le résultat de la valeur `Maybe`.

Mettons-là à l'essai :

```
ghci> solveRPN "1 2 * 4 +"
Just 6.0
ghci> solveRPN "1 2 * 4 + 5 *"
Just 30.0
ghci> solveRPN "1 2 * 4"
Nothing
ghci> solveRPN "1 8 wharglbllargh"
Nothing
```

Le premier échec est dû au fait que la pile finale n'a pas un seul élément, donc le filtrage par motif échoue dans l'expression `do`. Le deuxième échec a lieu parce que `readMaybe` retourne `Nothing`.

Composer des fonctions monadiques

Lorsque nous étudiions les lois des monades, nous avions dit que la fonction `<=<` était comme la composition, mais qu'au lieu de travailler sur des fonctions ordinaires comme `a -> b`, elle travaillait sur des fonctions comme `a -> m b`. Par exemple :

```
ghci> let f = (+1) . (*100)
ghci> f 4
401
ghci> let g = (\x -> return (x+1)) <=< (\x -> return (x*100))
ghci> Just 4 >>= g
Just 401
```

Dans cet exemple, on a d'abord composé deux fonctions ordinaires, appliqué la fonction résultante à `4`, et ensuite on a composé deux fonctions monadiques, et donné `Just 4` à la fonction résultante à l'aide de `>>=`.

Si l'on a tout un tas de fonctions dans une liste, on peut les composer en une unique énorme fonction en utilisant `id` comme accumulateur initial et `.` comme fonction binaire. Voici un exemple :

```
ghci> let f = foldr (.) id [(+1),(*100),(+1)]
ghci> f 1
201
```

La fonction `f` prend un nombre et lui ajoute `1`, multiplie le résultat par `100` et ajoute `1` à ça. On peut composer des fonctions monadiques de la même façon, seulement au lieu de la composition normale on utilise `<=<` et au lieu d'`id` on utilise `return`. On n'a même pas à remplacer `foldr` par `foldM` puisque la fonction `<=<` s'occupe de gérer la composition de manière monadique.

Quand on s'habituait à la monade des listes dans le chapitre précédent, on l'utilisait pour trouver si un cavalier pouvait aller d'une position d'un échiquier à une autre en exactement trois mouvements. On avait une fonction nommée `moveKnight` qui prenait la position du cavalier sur l'échiquier et retournait tous les déplacements possibles au prochain tour. Puis, pour générer toutes les positions possibles après trois mouvements, on a créé la fonction suivante :

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

Et pour vérifier s'il pouvait aller de `start` à `end` en trois mouvements, on faisait :

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

En utilisant la composition de fonctions monadiques, on peut créer une fonction comme `in3`, seulement qu'au lieu de générer toutes les positions possibles du cavalier après trois mouvements, on peut le faire pour un nombre de mouvements arbitraires. Si vous regardez `in3`, vous voyez qu'on utilise `moveKnight` trois fois, et à chaque fois, on utilise `>>=` pour lui donner toutes les positions précédentes. Rendons cela plus général à présent. Voici comment procéder :

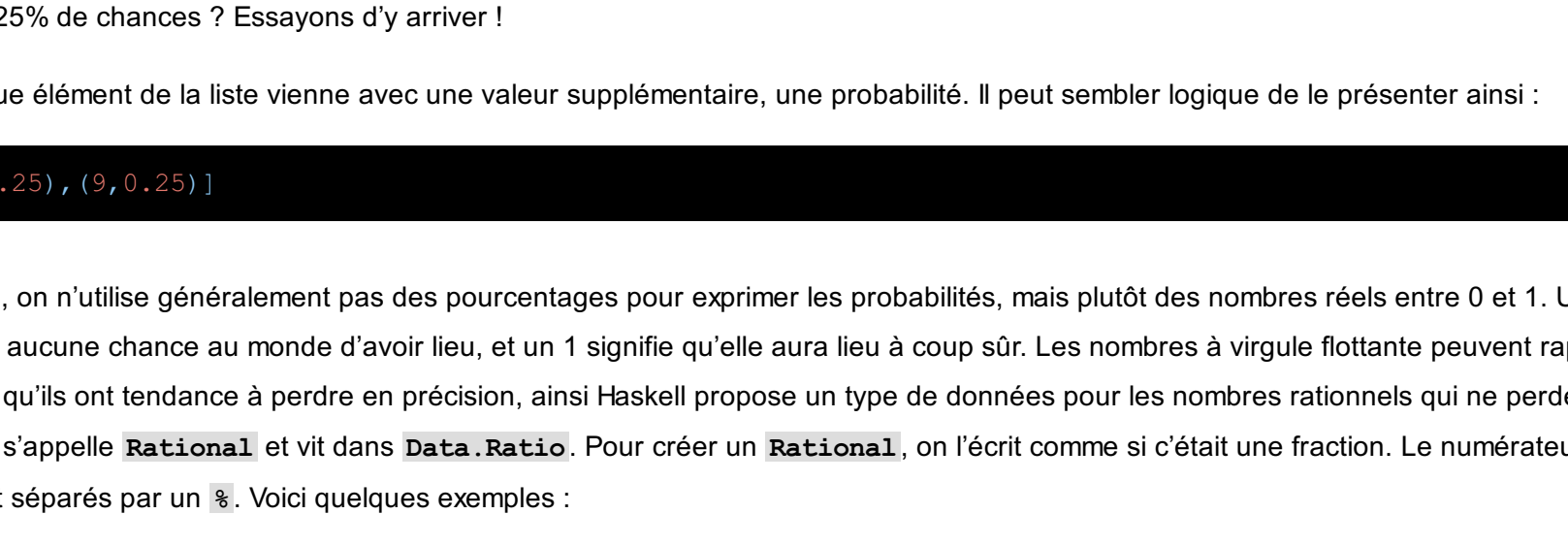
```
import Data.List

inMany :: Int -> KnightPos -> [KnightPos]
inMany x start = return start >>= foldr (<=<) return (replicate x moveKnight)
```

D'abord, on utilise `replicate` pour créer une liste contenant `x` copies de la fonction `moveKnight`. Puis, on compose monadiquement toutes ces fonctions en une seule, ce qui nous donne une fonction qui prend une position de départ, et déplace le cavalier `x` fois de façon non déterministe. Il suffit alors de placer la position initiale dans une liste singleton avec `return` et de la donner à notre fonction.

On peut maintenant changer la fonction `canReachIn3` pour être également plus générale :

```haskell
canReachIn :: Int -> KnightPos -> KnightPos -> Bool
canReachIn x start end = end `elem` inMany x start
```

Créer des monades



Dans cette section, on va regarder comment un type est créé, identifié comme une monade, et instancié en `Monad`. Généralement, on ne crée pas une monade pour le plaisir de créer une monade. Généralement, on crée plutôt un type dont le but est de modéliser un aspect du problème à résoudre, et plus tard si l'on s'aperçoit que ce type représente une valeur dans un contexte et peut agir comme une monade, on lui donne une instance de `Monad`.

Comme on l'a vu, les listes sont utilisées pour représenter des valeurs non déterministes. Une liste comme `[3, 5, 9]` peut être vue comme une unique valeur non déterministe qui ne peut tout simplement pas décider de ce qu'elle veut être. Quand on donne une liste à une fonction avec `>>=`, cela fait juste tous les choix possibles pour appliquer la fonction sur un élément de la liste, et le résultat est une liste présentant tous ces résultats.

Si l'on regarde la liste `[3, 5, 9]` comme les nombres `3`, `5` et `9` à la fois, on peut remarquer qu'il n'y a pas d'information quand à la probabilité de chacun d'eux. Et si l'on voulait modéliser une valeur non déterministe comme `[3, 5, 9]`, mais qu'on souhaitait exprimer que `3` a 50% de chances d'avoir lieu, alors que `5` et `9` n'ont chacun que 25% de chances ? Essayons d'y arriver !

Mettons que chaque élément de la liste vienne avec une valeur supplémentaire, une probabilité. Il peut sembler logique de le présenter ainsi :

```haskell
[(3,0.5),(5,0.25),(9,0.25)]
```

En mathématiques, on n'utilise généralement pas des pourcentages pour exprimer les probabilités, mais plutôt des nombres réels entre 0 et 1. Un 0 signifie que quelque chose n'a aucune chance au monde d'avoir lieu, et un 1 signifie qu'elle aura lieu à coup sûr. Les nombres à virgule flottante peuvent rapidement devenir bordéliques parce qu'ils ont tendance à perdre en précision, ainsi Haskell propose un type de données pour les nombres rationnels qui ne perdent pas en précision. Ce type s'appelle `Rational` et vit dans `Data.Ratio`. Pour créer un `Rational`, on l'écrit comme si c'était une fraction. Le numérateur et le dénominateur sont séparés par un `%`. Voici quelques exemples :

```haskell
ghci> 1%4
1 % 4
ghci> 1%2 + 1%2
1 % 1
ghci> 1%3 + 5%4
19 % 12
```

La première ligne est juste un quart. À la deuxième ligne, on additionne deux moitiés, ce qui nous donne un tout, et à la troisième ligne on additionne un tiers et cinq quarts et on obtient dix-neuf douzièmes. Jetons ces nombres à virgule flottante et utilisons plutôt des `Rational` pour nos probabilités :

```haskell
ghci> [(3,1%2),(5,1%4),(9,1%4)]
[(3,1 % 2),(5,1 % 4),(9,1 % 4)]
```

Ok, donc `3` a une chance sur deux d'avoir lieu, alors que `5` et `9` arrivent une fois sur quatre. Plutôt propre.

Nous avons pris des listes, et ajouter un contexte supplémentaire, afin qu'elles représentent des valeurs avec des contextes. Avant d'aller plus loin, enveloppons cela dans un `newtype` parce que mon petit doigt me dit qu'on va bientôt créer des instances.

```haskell
import Data.Ratio

newtype Prob a = Prob { getProb :: [(a,Rational)] } deriving Show
```

Bien. Est-ce un foncteur ? Eh bien, la liste étant un foncteur, ceci devrait probablement être également un foncteur, parce qu'on a juste rajouté quelque chose à la liste. Lorsqu'on mappe une fonction sur une liste, on l'applique à chaque élément. Ici, on va l'appliquer à chaque élément également, et on laissera les probabilités comme elles étaient. Créons une instance :

```haskell
instance Functor Prob where
    fmap f (Prob xs) = Prob $ map (\(x,p) -> (f x,p)) xs
```

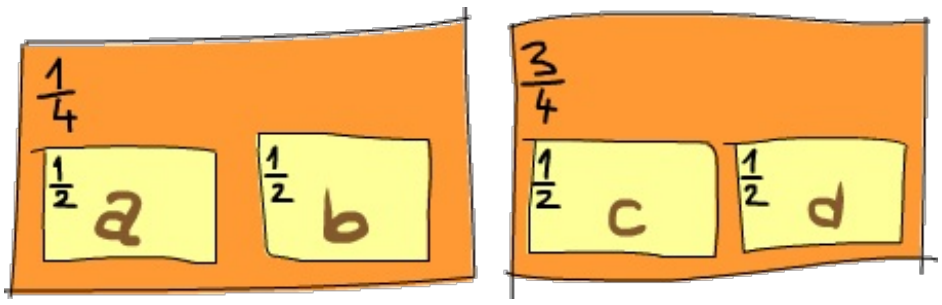
On sort la paire du `newtype` en filtrant par motif, on applique la fonction `f` aux valeurs en gardant les probabilités telles quelles, et on réencapsule le tout. Voyons si cela marche :

```haskell
ghci> fmap negate (Prob [(3,1%2),(5,1%4),(9,1%4)])
Prob {getProb = [(-3,1 % 2),(-5,1 % 4),(-9,1 % 4)]}
```

Autre chose à noter, les probabilités devraient toujours avoir pour somme `1`. Si ce sont bien toutes les choses qui peuvent avoir lieu, il serait illogique que la somme de leur probabilité ne fasse pas `1`. Une pièce qui a 75% de chances de faire pile et 50% de chances de faire face ne semble pouvoir marcher que dans un autre étrange univers.

Maintenant, la grande question, est-ce une monade ? Étant donné que la liste est une monade, on dirait que ça devrait aussi être une monade. Pensons d'abord à `return`. Comment marche-t-elle sur les listes ? Elle prend une valeur, et la place dans une liste singleton. Qu'en est-il ici ? Eh bien, puisque c'est censé être un contexte minimal par défaut, cela devrait aussi être une liste singleton. Qu'en est-il de la probabilité ? Eh bien, `return x` est supposée créer une valeur monadique qui présente toujours `x` en résultat, il serait donc illogique que la probabilité soit `0`. Si elle doit toujours la présenter en résultat, la probabilité devrait être `1` !

Qu'en est-il de `>>=` ? Ça semble un peu compliqué, profitons donc du fait que `m >>= f` est toujours égal à `join (fmap f m)` pour les monades, et pensons plutôt à la façon dont on aplatirait une liste de probabilités de listes de probabilités. Comme exemple, considérons une liste où il y a exactement 25% de chances que `'a'` ou `'b'` ait lieu. `a` et `b` ont la même probabilité. Également, il y a 75% de chances que `c` ou `d` ait lieu. `'c'` et `'d'` ont également la même probabilité. Voici une image de la liste de probabilités qui modélise ce scénario :



Quelles sont les chances que chacune de ces lettres ait lieu ? Si l'on devait redessiner ceci avec quatre boîtes, chacune avec une probabilité, quelles seraient ces probabilités ? Pour les trouver, il suffit de multiplier chaque probabilité par la probabilité qu'elle contient. `'a'` aurait lieu une fois sur huit, et de même pour `'b'`, parce que si l'on multiplie un demi par un quart, on obtient un huitième. `'c'` aurait lieu trois fois sur huit parce que trois quarts multipliés par un demi donnent trois huitièmes. `'d'` aurait aussi lieu trois fois sur huit. Si l'on somme toutes les probabilités, la somme vaut toujours un.

Voici cette situation exprimée comme une liste de probabilités :

```haskell
thisSituation :: Prob (Prob Char)
thisSituation = Prob
    [( Prob [('a',1%2),('b',1%2)] , 1%4 )
    ,( Prob [('c',1%2),('d',1%2)] , 3%4)
    ]
```

Remarquez que son type est `Prob (Prob Char)`. Maintenant qu'on a trouvé comment aplatir une liste de probabilités imbriquée, il nous suffit d'écrire le code correspondant, et on peut alors écrire `>>=` comme `join (fmap f m)` et avoir notre monade ! Voici `flatten`, qu'on nomme ainsi parce que le nom `join` est déjà pris :

```haskell
flatten :: Prob (Prob a) -> Prob a
flatten (Prob xs) = Prob $ concat $ map multAll xs
    where multAll (Prob innerxs,p) = map (\(x,r) -> (x,p*r)) innerxs
```

La fonction `multAll` prend un tuple formé d'une liste de probabilités et d'une probabilité `p` et multiplie toutes les probabilités à l'intérieur de cette première par `p`, retournant une liste de paires d'éléments et de probabilités. On mappe `multAll` sur chaque paire de notre liste de probabilités imbriquée et on aplatit simplement la liste imbriquée résultante.

On a à présent tout ce dont on a besoin pour écrire une instance de `Monad` !

```haskell
instance Monad Prob where
    return x = Prob [(x,1%1)]
    m >>= f = flatten (fmap f m)
    fail _ = Prob []
```

Puisqu'on a déjà fait tout le travail, l'instance est très simple. On a aussi défini la fonction `fail`, qui est la même que pour les listes, donc en cas d'échec d'un filtrage par motif dans une expression `do`, un échec a lieu dans le contexte d'une liste de

probabilités.

Il est également important de vérifier si les lois des monades tiennent pour l'instance qu'on vient de créer. La première dit que `return x >>= f` devrait être égal à `f x`. Une preuve rigoureuse serait fastidieuse, mais on peut voir que si l'on place une valeur dans un contexte par défaut avec `return` et qu'on `fmap` une fonction par dessus, puis qu'on aplatit la liste de probabilités résultante, chaque probabilité résultant de la fonction serait multipliée par la probabilité `1%1` créée par `return`, donc le contexte serait inaffecté. Le raisonnement montrant que `m >>= return` est égal à `m` est similaire. La troisième loi dit que `f <=< (g <=< h)` devrait être égal à `(f <=< g) <=< h`. Puisque cette loi tient pour la monade des listes, qui est la base de la monade des probabilités, et parce que la multiplication est associative, cette loi tient donc également pour la monade des probabilités. `1%2 * (1%3 * 1%5)` est égal à `(1%2 * 1%3) * 1%5`.

À présent qu'on a une monade, que peut-on en faire ? Eh bien, elle peut nous aider à faire des calculs avec des probabilités. On peut traiter des évènements probabilistes comme des valeurs dans des contextes, et la monade de probabilité s'assurera que les probabilités sont bien reflétées dans le résultat final.

Mettons qu'on ait deux pièces normales et une pièce pipée qui donne pile un nombre ahurissant de neuf fois sur dix, et face seulement une fois sur dix. Si l'on jette les trois pièces à la fois, quelles sont les chances qu'elles atterrissent toutes sur pile ? D'abord, créons des valeurs de probabilités pour un lancer de pièce normale et un lancer de pièce pipée :

```
data Coin = Heads | Tails deriving (Show, Eq)

coin :: Prob Coin
coin = Prob [(Heads,1%2),(Tails,1%2)]

loadedCoin :: Prob Coin
loadedCoin = Prob [(Heads,1%10),(Tails,9%10)]
```

Et finalement, le lancer de pièces en action :

```
import Data.List (all)

flipThree :: Prob Bool
flipThree = do
    a <- coin
    b <- coin
    c <- loadedCoin
    return (all (==Tails) [a,b,c])
```

En l'essayant, on voit que les chances que les trois pièces atterrissent sur pile ne sont pas très bonnes, en dépit d'avoir triché avec notre pièce pipée :

```
ghci> getProb flipThree
[(False,1 % 40),(False,9 % 40),(False,1 % 40),(False,9 % 40),
 (False,1 % 40),(False,9 % 40),(False,1 % 40),(True,9 % 40)]
```

Toutes les trois atterriront sur pile neuf fois sur quarante, ce qui fait moins de 25% de chances. On voit que notre monade ne sait pas joindre tous les résultats `False` où les pièces ne tombent pas toutes sur pile en un seul résultat. Ce n'est pas un gros problème, puisqu'écrire une fonction qui regroupe tous ses résultats en un seul est plutôt facile et est laissé comme exercice pour le lecteur (vous !).

Dans cette section, on est parti d'une question (et si les listes transportaient également une information de probabilité ?) pour créer un type, reconnaître une monade et finalement créer une instance et faire quelque chose avec elle. Je trouve ça plutôt attrayant ! À ce stade, on devrait avoir une assez bonne compréhension de ce que sont les monades.

← Pour une poignée de monades | Table des matières | Zippeurs →

Zippeurs

⬅ Et pour quelques monades de plus

Table des matières

Alors que la pureté d'Haskell amène tout un tas de bienfaits, elle nous oblige à attaquer certains problèmes sous un autre angle que celui qu'on aurait pris dans des langages impurs. À cause de la transparence référentielle, une valeur est aussi bonne qu'une autre en Haskell si elles représentent toutes les deux la même chose.

Donc si on a un arbre plein de cinq (tapez-m'en cinq ?) et qu'on veut en changer un en un six, il nous faut un moyen de savoir exactement quel cinq dans notre arbre on veut changer. Il faut savoir où il est dans l'arbre. Dans les langages impurs, on aurait pu noter l'adresse mémoire où est situé le cinq, et changer la valeur à cette adresse. Mais en Haskell, un cinq est un cinq comme les autres, et on ne peut donc pas le discriminer en fonction de sa position en mémoire. On ne peut pas non plus vraiment *changer* quoi que ce soit, et quand on dit qu'on change un arbre, on veut en fait dire qu'on prend un arbre et qu'on en retourne un autre, similaire à l'original mais un peu différent.

Une possibilité consiste à se rappeler du chemin de la racine de l'arbre à l'élément qu'on souhaite changer. On pourrait dire : prends cet arbre, va à gauche, va à droite, et encore à gauche, et change l'élément qui est à cet endroit. Bien que cela marche, ça peut être inefficace. Si l'on veut plus tard changer un élément qui est juste à côté de celui qu'on vient de changer, on doit traverser l'arbre à nouveau depuis la racine jusqu'à cet élément !

Dans ce chapitre, nous allons voir qu'on peut prendre une structure de données et se focaliser sur une partie de celle-ci de façon à ce que modifier ses éléments soit facile, et à ce que se balader autour soit efficace. Joli !

Une petite balade

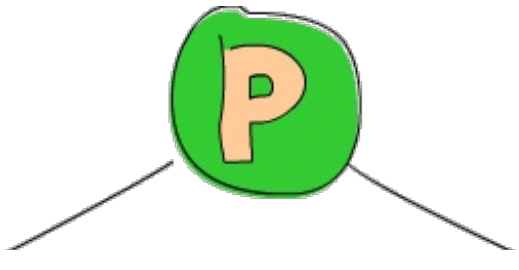
Comme on l'a appris en cours de biologie, il y a beaucoup de sortes différentes d'arbres, alors choisissons une graine qu'on utilisera pour planter le nôtre. La voici :

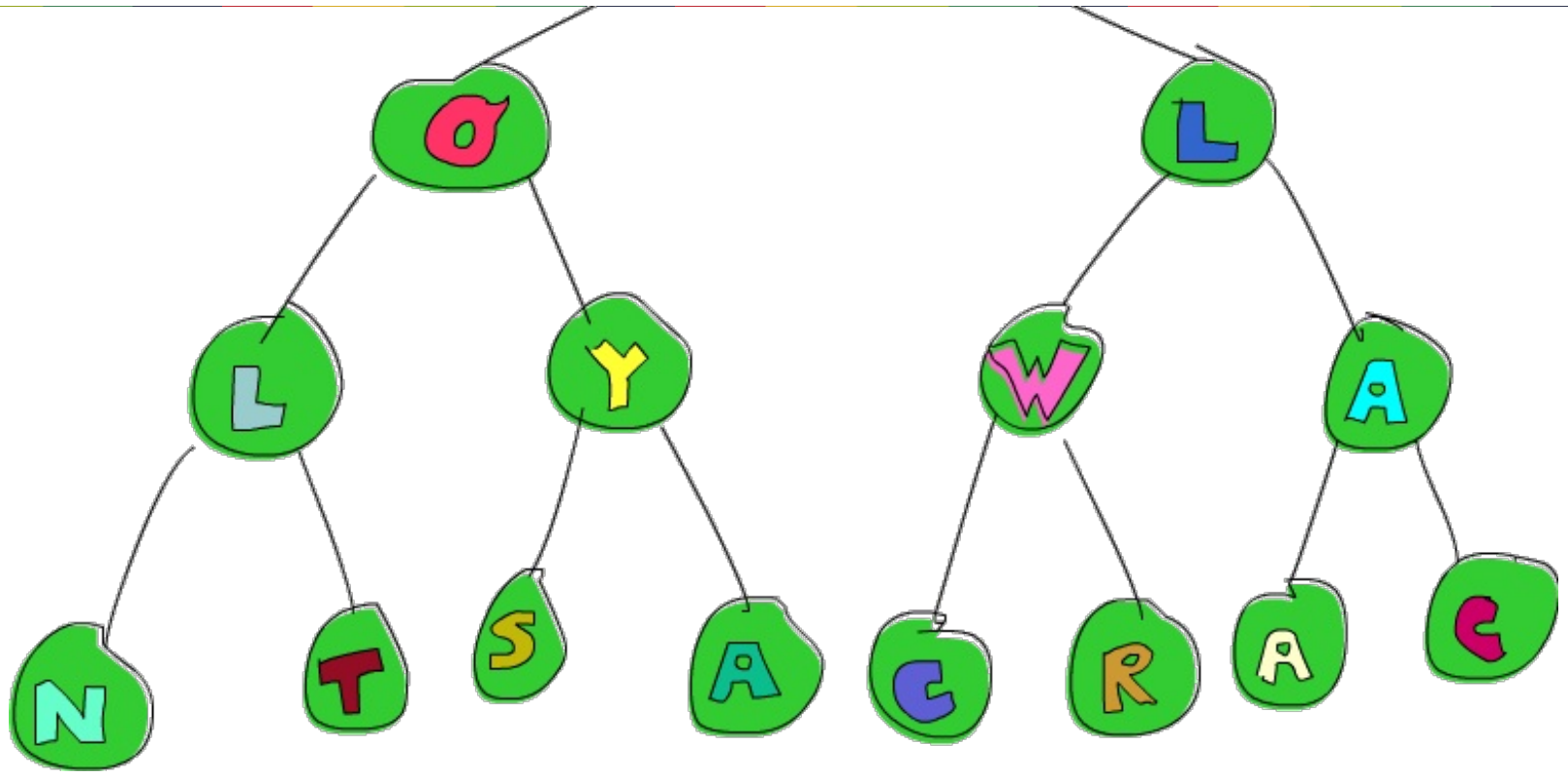
```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
```

Notre arbre est donc ou bien vide, ou bien un nœud qui a un élément et deux sous-arbres. Voici un bon exemple d'un tel arbre, que je donne, à vous lecteur, gratuitement !

```
freeTree :: Tree Char
freeTree =
    Node 'P'
        (Node 'O'
            (Node 'L'
                (Node 'N' Empty Empty)
                (Node 'T' Empty Empty)
            )
            (Node 'Y'
                (Node 'S' Empty Empty)
                (Node 'A' Empty Empty)
            )
        )
        (Node 'L'
            (Node 'W'
                (Node 'C' Empty Empty)
                (Node 'R' Empty Empty)
            )
            (Node 'A'
                (Node 'A' Empty Empty)
                (Node 'C' Empty Empty)
            )
        )
```

Et voici ce même arbre représenté graphiquement :





Remarquez-vous le `W` dans cet arbre ? Disons qu'on souhaite le changer en un `P`. Comment ferions-nous cela ? Eh bien, une façon de faire consisterait à filtrer par motif sur notre arbre jusqu'à ce qu'on trouve l'élément situé en allant d'abord à droite puis à gauche, et qu'on change cet élément. Voici le code faisant cela :

```haskell
changeToP :: Tree Char -> Tree Char
changeToP (Node x l (Node y (Node _ m n) r)) = Node x l (Node y (Node 'P' m n) r)
```

Beurk ! Non seulement c'est plutôt laid, mais c'est aussi un peu déroutant. Qu'est-ce qu'il se passe ici ? Eh bien, on filtre par motif sur notre arbre et on appelle sa racine `x` (il devient le `'P'` de la racine) et son sous-arbre gauche `l`. Plutôt que de donner un nom au sous-arbre droit, on le filtre à nouveau par motif. On continue ce filtrage jusqu'à atteindre le sous-arbre dont notre `'W'` est la racine. Une fois ceci fait, on recrée l'arbre, seulement le sous-arbre qui contenait le `'W'` a maintenant pour racine `'P'`.

Y a-t-il un meilleur moyen de faire ceci ? Pourquoi ne pas créer une fonction qui prenne un arbre ainsi qu'une liste de directions ? Les directions seront soit `L` soit `R`, représentant respectivement la gauche et la droite, et on changera l'élément sur lequel on tombe en suivant ces directions. Voici :

```haskell
data Direction = L | R deriving (Show)
type Directions = [Direction]

changeToP :: Directions -> Tree Char -> Tree Char
changeToP (L:ds) (Node x l r) = Node x (changeToP ds l) r
changeToP (R:ds) (Node x l r) = Node x l (changeToP ds r)
changeToP [] (Node _ l r) = Node 'P' l r
```

Si le premier élément de notre liste de directions est un `L`, on construit un nouvel arbre comme l'original, mais son sous-arbre gauche a un élément modifié en `'P'`. Quand on appelle récursivement `changeToP`, on ne donne que la queue de la liste des directions, puisqu'on est déjà allé une fois à gauche. On fait la même chose dans le cas de `R`. Si la liste des directions est vide, cela signifie qu'on est arrivé à destination, on retourne donc un arbre qui est comme celui en entrée, mais avec `'P'` à sa racine.

Pour éviter d'afficher l'arbre entier, créons une fonction qui prend une liste de directions et nous dit quel est l'élément à la destination :

```haskell
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Node _ l _) = elemAt ds l
elemAt (R:ds) (Node _ _ r) = elemAt ds r
elemAt [] (Node x _ _) = x
```

La fonction est en fait assez similaire à `changeToP`, seulement au lieu de se souvenir des choses en chemin et de reconstruire l'arbre, elle ignore simplement tout sauf sa destination. Ici on change le `'W'` en `'P'` et on vérifie si le changement a bien eu lieu dans le nouvel arbre :

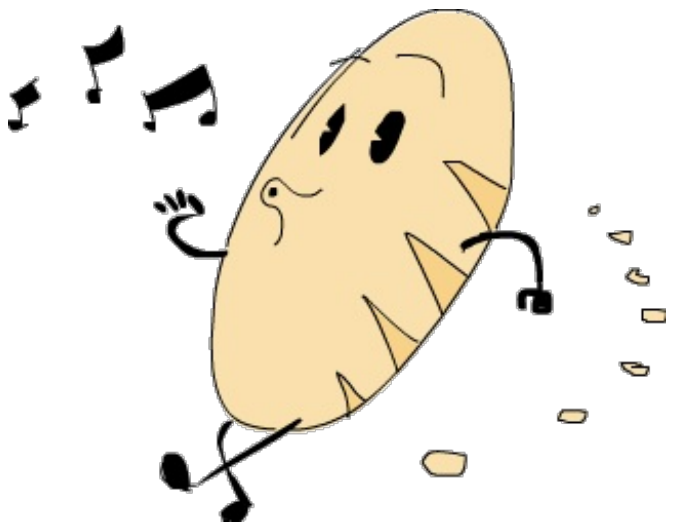
```haskell
ghci> let newTree = changeToP [R,L] freeTree
ghci> elemAt [R,L] newTree
'P'
```

Bien, ça a l'air de marcher. Dans ces fonctions, la liste des directions agit comme une sorte de *point focal*, parce qu'il désigne exactement un sous-arbre de notre arbre. Une liste de directions comme `[R]` se focalise sur le sous-arbre droit de la racine, par exemple. Une liste de directions vide se focalise sur l'arbre entier.

Bien que cette technique ait l'air cool, elle peut être assez inefficace, notamment si l'on veut changer des éléments à répétition. Disons qu'on ait un arbre énorme et une très longue liste de directions qui pointe vers un élément tout en bas de cet arbre. On utilise la liste de directions pour se balader dans l'arbre et changer

l'élément en bas. Si l'on souhaite changer un autre élément proche de cet élément, on doit repartir de la racine et retraverser tout l'arbre depuis le début à nouveau ! La barbe !

Une traînée de miettes



Ok, donc pour se concentrer sur un sous-arbre, on veut quelque chose de mieux qu'une liste de directions qu'on suit toujours depuis la racine de notre arbre. Est-ce que cela aiderait si l'on commençait à la racine de l'arbre, et qu'on se déplaçait vers la gauche ou la droite, une étape à la fois, en laissant d'une certaine façon des miettes de pain sur notre passage ? C'est-à-dire, quand on va à gauche, on se souvient qu'on est allé à gauche, et quand on va à droite, on se souvient qu'on est allé à droite. Bien, on peut essayer.

Pour représenter nos miettes de pain, on va aussi utiliser une liste de `Direction` (soit `L`, soit `R`), seulement au lieu de l'appeler `Directions`, on l'appellera `Breadcrumbs`, puisque nos directions seront à présent renversées puisqu'on jette les miettes au fur et à mesure qu'on avance dans l'arbre :

```
type Breadcrumbs = [Direction]
```

Voici une fonction qui prend un arbre et des miettes et se déplace dans le sous-arbre gauche tout en ajoutant `L` dans la liste qui représente nos miettes de pain :

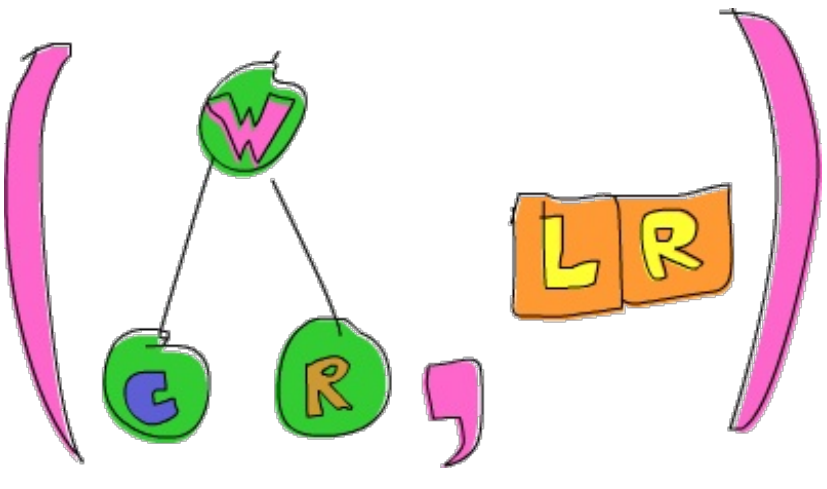
```
goLeft :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goLeft (Node _ l _, bs) = (l, L:bs)
```

On ignore l'élément à la racine ainsi que le sous-arbre droit, et on retourne juste le sous-arbre gauche ainsi que l'ancienne traînée de miettes, avec `L` placé à sa tête. Voici une fonction pour aller à droite :

```
goRight :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goRight (Node _ _ r, bs) = (r, R:bs)
```

Elle fonctionne de façon similaire. Utilisons ces fonctions pour prendre notre `freeTree` et aller à droite puis à gauche :

```
ghci> goLeft (goRight (freeTree, []))
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```



Ok, donc maintenant on a un arbre qui a `'W'` à sa racine, `'C'` à la racine de son sous-arbre gauche, et `'R'` à la racine de son sous-arbre droit. Les miettes de pain sont `[L, R]` parce qu'on est d'abord allé à droite, puis à gauche.

Pour rendre plus clair le déplacement dans notre arbre, on peut réutiliser la fonction `-:` qu'on avait définie ainsi :

```
x -: f = f x
```

Ce qui nous permet d'appliquer des fonctions à des valeurs en écrivant d'abord la valeur, puis `-:` et enfin la fonction. Ainsi, au lieu d'écrire `goRight (freeTree, [])`, on peut écrire `(freeTree, []) -: goRight`. En utilisant ceci, on peut réécrire le code ci-dessus de façon à faire mieux apparaître qu'on va en premier à droite, et ensuite à gauche :

```
ghci> (freeTree, []) -: goRight -: goLeft
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```

Retourner en haut

Et si l'on veut remonter dans l'arbre à présent ? Grâce à notre traînée de miettes, on sait que notre arbre actuel est le sous-arbre gauche de son parent, et que celui-ci est le sous-arbre droit de son parent, mais c'est tout. Elles ne nous en disent pas assez sur le parent du sous-arbre pour qu'on puisse remonter dans l'arbre. Il semblerait qu'en plus de la direction prise, notre miette de pain devrait aussi contenir toutes les données nécessaires pour remonter dans l'arbre. Dans ce cas, il nous faudrait l'élément de l'arbre parent ainsi que le sous-arbre droit.

En généal, une miette de pain doit contenir toutes les données nécessaires pour reconstruire son nœud parent. Elle doit donc contenir toute l'information sur les chemins que l'on n'a pas suivis, ainsi que la direction que l'on a prise, mais elle n'a pas besoin de contenir le sous-arbre sur lequel on se focalise, puisqu'on a déjà ce sous-arbre dans la première composante de notre tuple, donc s'il était aussi dans la miette, il y aurait une duplication de l'information.

Modifions notre miette de pain afin qu'elle contienne aussi l'information sur tout ce qu'on a ignoré précédemment quand on a choisi d'aller à gauche ou à droite. Au lieu de `Direction`, définissons un nouveau type de données :

```haskell
data Crumb a = LeftCrumb a (Tree a) | RightCrumb a (Tree a) deriving (Show)
```

Maintenant, au lieu d'avoir juste un `L`, on a un `LeftCrumb` qui contient également l'élément dans le nœud parent, ainsi que son sous-arbre droit qu'on a choisi de ne pas visiter. Au lieu de `R`, on a un `RightCrumb`, qui contient aussi l'élément du nœud parent et son sous-arbre gauche qu'on n'a pas visité.

Ces miettes de pain contiennent à présent toutes les données nécessaires pour recréer l'arbre qu'on a parcouru. Ainsi, au lieu d'être juste des miettes de pain normales, ce sont plutôt des petits disques durs qu'on parsème en chemin, parce qu'ils contiennent beaucoup plus d'informations que seulement la direction prise.

En gros, chaque miette de pain est maintenant comme un nœud d'arbre mais avec un trou d'un côté. Lorsqu'on se déplace en profondeur dans l'arbre, la miette transporte toute l'information du nœud duquel on est parti, *sauf* le sous-arbre sur lequel on se focalise. Elle doit aussi savoir de quel côté se trouve le trou. Dans le cas d'un `LeftCrumb`, on sait qu'on est allé à gauche, donc le sous-arbre manquant est le sous-arbre gauche.

Changeons également notre synonyme de type `Breadcrumbs` pour refléter le changement :

```haskell
type Breadcrumbs a = [Crumb a]
```

À présent, on doit modifier les fonctions `goLeft` et `goRight` afin qu'elles stockent l'information sur les chemins que l'on n'a pas pris dans nos miettes, au lieu de tout ignorer comme elles le faisaient auparavant. Voici `goLeft` :

```haskell
goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

Vous pouvez voir que ça ressemble beaucoup à notre ancienne `goLeft`, seulement au lieu d'ajouter seulement un `L` en tête de notre liste de miettes, on ajoute un `LeftCrumb` qui indique qu'on est allé à gauche, et on munit ce `LeftCrumb` de l'élément du nœud qu'on quitte (le `x`) et de son sous-arbre droit qu'on a choisi de ne pas visiter.

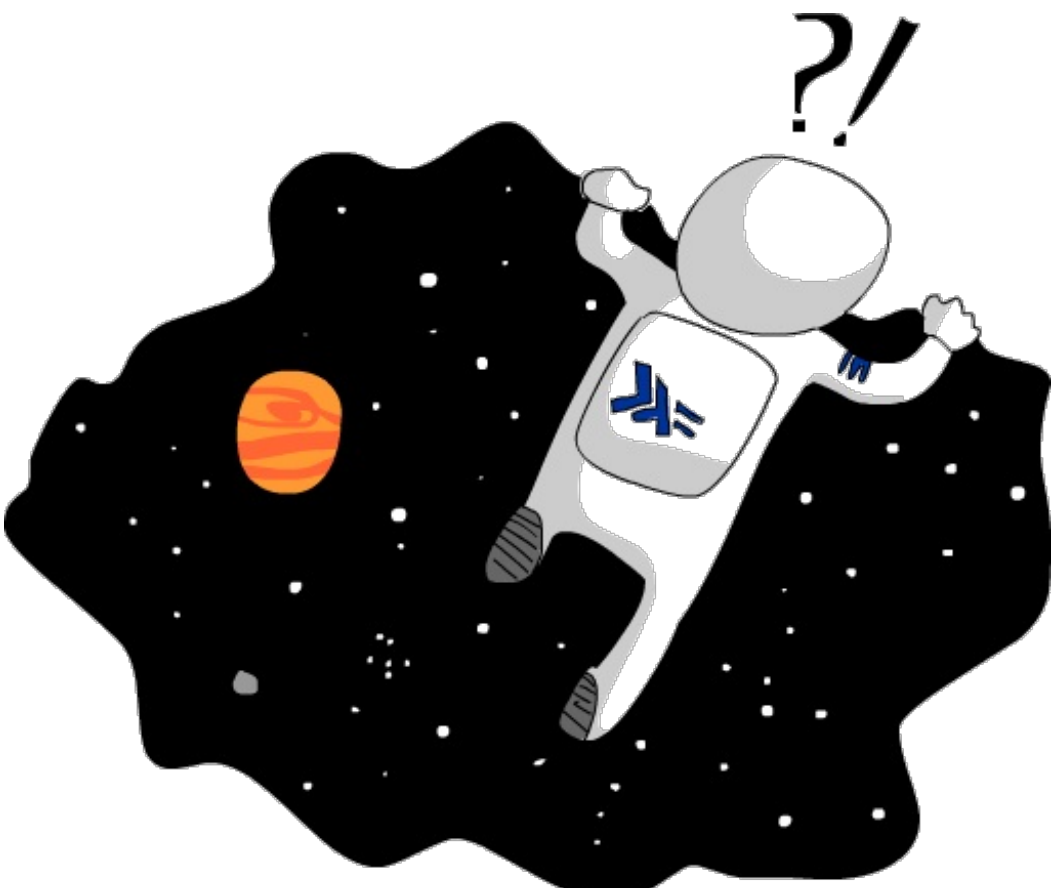
Remarquez que cette fonction suppose que l'arbre sur lequel on se focalise actuellement n'est pas `Empty`. Un arbre vide n'a pas de sous-arbres, donc si l'on essaie d'aller à gauche dans un arbre vide, une erreur aura lieu parce que le filtrage par motif sur `Node` échouera, et parce qu'on n'a pas mis de motif pour `Empty`.

`goRight` est similaire :

```haskell
goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goRight (Node x l r, bs) = (r, RightCrumb x l:bs)
```

Précédemment, nous étions capable d'aller à gauche ou à droite. Ce qu'on a gagné, c'est la possibilité de rebrousser chemin, en se souvenant de ce à quoi ressemblaient les nœuds parents et les chemins qu'on n'a pas visités. Voici la fonction `goUp` :

```haskell
goUp :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```



On se focalise sur l'arbre `t`, et on vérifie ce qu'était le dernier `Crumb`. Si c'était un `LeftCrumb`, alors on construit un nouvel arbre où notre arbre `t` est le sous-arbre gauche, et on utilise l'information à propos du sous-arbre droit qu'on n'a pas visité et de l'élément pour reconstruire le reste du `Node`. Puisqu'on a en quelque sorte rebroussé chemin et ramassé la dernière miette pour récréer le nœud parent, la nouvelle liste de miettes ne contient pas cette miette.

Remarquez que cette fonction cause une erreur si l'on est déjà à la racine d'un arbre et qu'on essaie de remonter. Plus tard, on utilisera la monade `Maybe` pour représenter un échec possible lorsqu'on change de point focal.

Avec une paire de `Tree a` et de `Breadcrumbs a`, on a toute l'information nécessaire pour reconstruire un arbre entier, tout en étant focalisé sur un de ses sous-arbres. Ce mécanisme nous permet de facilement nous déplacer à gauche, à droite, et vers le haut. Une telle paire contenant une partie focalisée d'une structure de données, et ses alentours, est appelée un zippeur, parce que changer de point focal vers le haut et vers le bas fait

penser à une braguette qui monte et descend. Il est donc pratique de créer un synonyme de types :

```
type Zipper a = (Tree a, Breadcrumbs a)
```

Je préférerais appeler le synonyme de types `Focus` pour indiquer clairement qu'on se focalise sur une partie de la structure de données, mais le terme zippeur est globalement accepté pour décrire un tel mécanisme, on s'en tiendra donc à `Zipper`.

Manipuler des arbres sous focalisation

À présent qu'on sait se déplacer de haut en bas, créons une fonction qui modifie l'élément à la racine de l'arbre sur lequel on est actuellement focalisé :

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, bs) = (Node (f x) l r, bs)
modify f (Empty, bs) = (Empty, bs)
```

Si l'on se focalise sur un nœud, on modifie son élément racine avec la fonction `f`. Si on se focalise sur un arbre vide, on ne fait rien. Désormais, on peut démarrer avec un arbre, se déplacer où l'on veut, et modifier l'élément à cet endroit, tout en restant focalisé sur cet élément de façon à pouvoir facilement se déplacer de haut en bas par la suite. Un exemple :

```
ghci> let newFocus = modify (\_ -> 'P') (goRight (goLeft (freeTree,[])))
```

On va à gauche, puis à droite, et on modifie l'élément racine en le remplaçant par un `'P'`. C'est encore plus lisible avec `-:` :

```
ghci> let newFocus = (freeTree,[]) -: goLeft -: goRight -: modify (\_ -> 'P')
```

On peut ensuite remonter si l'on veut, et remplacer un élément par un mystérieux `'X'` :

```
ghci> let newFocus2 = modify (\_ -> 'X') (goUp newFocus)
```

Ou avec `-:` :

```
ghci> let newFocus2 = newFocus -: goUp -: modify (\_ -> 'X')
```

Remonter est facile puisque les miettes qu'on a laissées forment la structure de données que l'on n'a pas visitée, seulement tout est renversé, un peu comme une chaussette à l'envers. C'est pourquoi, lorsqu'on veut remonter d'un cran, on n'a pas besoin de repartir de la racine et de redescendre presque tout en bas, il nous suffit de prendre le sommet de notre arbre inversé, en le remettant dans le bon sens et sous notre focalisation.

Chaque nœud a deux sous-arbres, même si ces sous-arbres sont vides. Ainsi, si l'on se focalise sur un arbre vide, on peut le remplacer par un sous-arbre non vide, en lui attachant un arbre comme feuille. Le code pour faire cela est simple :

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_, bs) = (t, bs)
```

On prend un arbre et un zippeur, et on retourne un nouveau zippeur, dont le point focal est remplacé par l'arbre en question. Non seulement peut-on étendre des arbres ainsi, en remplaçant des sous-arbres vides par d'autres arbres, mais on peut aussi complètement remplacer des sous-arbres existants. Attachons un arbre tout à gauche de notre `freeTree` :

```
ghci> let farLeft = (freeTree,[]) -: goLeft -: goLeft -: goLeft -: goLeft
ghci> let newFocus = farLeft -: attach (Node 'Z' Empty Empty)
```

`newFocus` est à présent focalisé sur l'arbre qu'on vient juste d'attacher, et le reste de l'arbre est stocké renversé dans les miettes de pain. Si l'on utilisait `goUp` pour remonter jusqu'à la racine de l'arbre, il serait exactement comme `freeTree`, mais avec un `'Z'` en plus tout à gauche.

I'm going straight to the top, oh yeah, up where the air is fresh and clean!

Créer une fonction qui remonte tout en haut de notre arbre, peu importe l'endroit sur lequel on s'est focalisé, est assez facile. La voilà :

```
topMost :: Zipper a -> Zipper a
topMost (t,[]) = (t,[])
topMost z = topMost (goUp z)
```

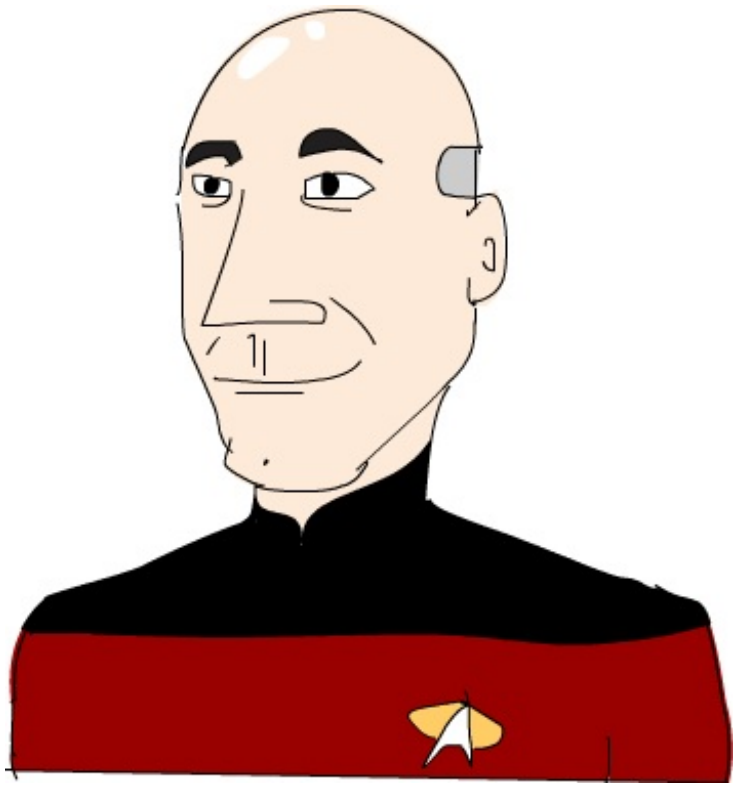
Si notre traînée de miettes est vide, cela signifie qu'on est arrivé à la racine de l'arbre, on retourne donc le point focal actuel. Sinon, on remonte d'un cran pour obtenir le point focal du nœud parent, et on appelle récursivement `topMost` sur ce point focal. À présent, on peut se balader dans notre arbre, aller à gauche, à droite, en haut, appliquer `modify` et `attach` tout en se déplaçant, et quand on a fini de faire nos modifications, on peut utiliser `topMost` pour nous focaliser à nouveau sur la racine de l'arbre, et voir tous les changements effectués depuis cette meilleure perspective.

Se focaliser sur des listes

Les zippeurs peuvent être utilisés avec quasiment toutes les structures de données, il n'est donc pas surprenant qu'on puisse les utiliser pour se focaliser sur les sous-listes d'une liste. Après tout, les listes sont un peu comme des arbres, seulement là où le nœud d'un arbre a un élément (ou pas) et plusieurs sous-arbres, un nœud d'une liste n'a qu'un élément et qu'une sous-liste. Quand on avait implémenté nos propres listes, on avait défini le type de données ainsi :

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

Contrastez ceci avec la définition de notre arbre binaire, et vous verrez facilement que les listes peuvent être considérées comme des arbres pour lesquelles chaque nœud n'a qu'un sous-arbre.



Une liste comme `[1, 2, 3]` peut être écrite `1:2:3:[]`. Elle consiste en une tête de liste, qui est `1`, et une queue, qui est `2:3:[]`. À son tour, `2:3:[]` a aussi une tête, qui est `2`, et une queue, qui est `3:[]`. Avec `3:[]`, la tête est `3` et la queue est la liste vide `[]`.

Créons un zippeur de listes. Pour changer le point focal d'une sous-liste d'une liste, on peut aller soit en avant, soit en arrière (alors que pour les arbres, on pouvait aller en haut, à gauche ou à droite). La partie focalisée sera une sous-liste, avec une traînée de miettes qu'on aura laissée en avançant. En quoi consisterait donc une miette de liste ? Quand on travaillait avec des arbres binaires, on disait qu'une miette devait contenir l'élément racine du nœud parent, ainsi que tous les sous-arbres qu'on n'avait pas visités. Elle devait aussi se souvenir si l'on était allé à gauche ou à droite. Ainsi, elle doit contenir toute l'information du nœud parent, à l'exception du sous-arbre qu'on choisit de visiter.

Les listes sont plus simples que des arbres, donc on n'a pas besoin de se souvenir si l'on est allé à gauche ou à droite, parce qu'il n'y a qu'un chemin pour aller plus profondément dans la liste. Et puisqu'il n'y a qu'un sous-arbre par nœud, on n'a pas besoin de se souvenir de chemins qu'on n'aurait pas pris. Il semblerait que tout ce dont on ait besoin de se souvenir est l'élément précédent. Si l'on a une liste comme `[3, 4, 5]`, et qu'on sait que l'élément précédent était `2`, on peut rebrousser chemin en plaçant simplement cet élément en tête de liste, obtenant `[2, 3, 4, 5]`.

Puisqu'une miette de pain est juste un élément dans ce cas, on n'a pas vraiment besoin de la placer dans un type de données, comme on l'avait fait avec `Crumb` pour notre zippeur d'arbres :

```
type ListZipper a = ([a],[a])
```

La première liste représente la liste sur laquelle on se focalise, et la seconde est la liste des miettes. Créons des fonctions pour avancer et reculer dans des listes :

```
goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs, x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

Quand on avance, on se focalise sur la queue de la liste actuelle et on laisse la tête comme miette. Quand on rebrousse chemin, on prend la dernière miette, et on la replace en tête de liste.

Voici les deux fonctions en action :

```
ghci> let xs = [1,2,3,4]
ghci> goForward (xs,[])
([2,3,4],[1])
ghci> goForward ([2,3,4],[1])
([3,4],[2,1])
ghci> goForward ([3,4],[2,1])
([4],[3,2,1])
ghci> goBack ([4],[3,2,1])
([3,4],[2,1])
```

On voit que les miettes de pain dans le cas des listes ne sont rien de plus qu'une partie de la liste renversée. L'élément dont on part devient la tête des miettes, il

est donc facile de revenir en arrière en reprenant la tête des miettes et en en faisant la tête de notre point focal.

On voit également mieux pourquoi on appelle ça un zippeur, parce que cela ressemble vraiment à une braguette montant et descendant.

Si vous étiez en train de créer un éditeur de texte, vous pourriez utiliser une liste de chaînes de caractères pour représenter les lignes du fichier de texte ouvert, et utiliser un zippeur pour savoir à quelle ligne le curseur se trouve actuellement. En utilisant un zippeur, il serait également facile d'insérer de nouvelles lignes n'importe où dans le texte ou d'effacer des lignes existantes.

Un système de fichiers élémentaire

Maintenant qu'on sait comment les zippeurs fonctionnent, utilisons des arbres pour représenter un système de fichiers très simple, et créons un zippeur pour ce système de fichiers, qui nous permettra de nous déplacer à travers les dossiers, comme on le fait généralement quand on parcourt un système de fichiers.

Une vue simpliste du système de fichiers usuel consiste à le voir comme composé principalement de fichiers et de dossiers. Les fichiers sont des unités de données qui ont un nom, alors que les dossiers sont utilisés pour organiser ces fichiers et peuvent contenir d'autres dossiers. Disons donc qu'un élément d'un système de fichiers est soit un fichier, avec son nom et ses données, soit un dossier, qui a un nom et tout un tas d'autres éléments pouvant être eux-mêmes des fichiers ou des dossiers. Voici le type de données qu'on utilisera et quelques synonymes de types pour savoir qui est quoi :

```
type Name = String
type Data = String
data FSItem = File Name Data | Folder Name [FSItem] deriving (Show)
```

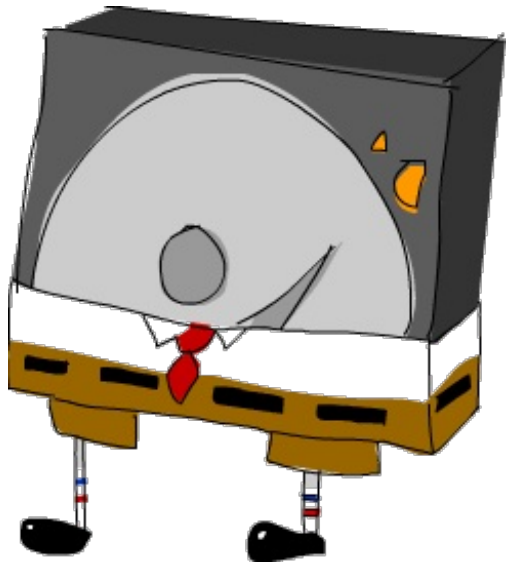
Un fichier vient avec deux chaînes de caractères, qui représentent son nom et les données qu'il contient. Un dossier vient avec une chaîne de caractères, qui est son nom, et une liste d'éléments. Si cette liste est vide, on a un dossier vide.

Voici un dossier avec des fichiers et des sous-dossiers :

```
myDisk :: FSItem
myDisk =
    Folder "root"
        [ File "goat_yelling_like_man.wmv" "baaaaaa"
        , File "pope_time.avi" "god bless"
        , Folder "pics"
            [ File "ape_throwing_up.jpg" "bleargh"
            , File "watermelon_smash.gif" "smash!!"
            , File "skull_man(scary).bmp" "Yikes!"
            ]
        , File "dijon_poupon.doc" "best mustard"
        , Folder "programs"
            [ File "fartwizard.exe" "10gotofart"
            , File "owl_bandit.dmg" "mov eax, h00t"
            , File "not_a_virus.exe" "really not a virus"
            , Folder "source code"
                [ File "best_hs_prog.hs" "main = print (fix error)"
                , File "random.hs" "main = print 4"
                ]
            ]
        ]
```

C'est exactement ce que contient mon disque dur à l'instant.

Un zippeur pour notre système de fichiers



Maintenant qu'on a un système de fichiers, tout ce dont on a besoin est d'un zippeur afin de zippeur et zoomer dans celui-ci, et d'ajouter, modifier ou supprimer des fichiers ou des dossiers. Tout comme avec les arbres binaires et les listes, on va semer des miettes qui contiennent l'information nécessaire concernant les choses qu'on n'a pas choisi de visiter. Comme on l'a dit, une miette doit contenir tout, à l'exception du sous-arbre sur lequel on se focalise. Elle doit aussi savoir où se situe le trou, afin de pouvoir replacer notre point focal au bon endroit quand on décidera de rebrousser chemin.

Dans ce cas, une miette de pain doit être un dossier dans lequel il manque le dossier qu'on a choisi d'explorer. Et pourquoi pas un fichier plutôt ? Eh bien, parce qu'une fois qu'on se focalise sur un fichier, on ne peut pas descendre plus profond dans le système de fichiers, donc une miette qui dirait qu'on vient d'un fichier ne serait pas logique. Un fichier est en quelque sorte un arbre vide.

Si l'on se focalise sur le dossier `"root"`, puis qu'on choisit de se focaliser sur le fichier `"dijon_poupon.doc"`, à quoi devrait ressembler la miette qu'on laissera ? Eh bien, elle devrait contenir le nom du dossier parent ainsi que les éléments qui venaient avant le fichier sur lequel on se focalise et les éléments qui viennent après. Tout ce dont on a besoin est donc d'un `Name` et de deux listes d'éléments. En gardant séparée la liste des éléments venant avant et celle des éléments venant après, on sait exactement où placer l'élément sur lequel on était focalisé quand on retourne en arrière. Ainsi,

on sait où se trouve le trou.

Voici notre type miette pour le système de fichiers :

```
data FSCrumb = FSCrumb Name [FSItem] [FSItem] deriving (Show)
```

Et voici un synonyme de type pour notre zippeur :

```
type FSZipper = (FSItem, [FSCrumb])
```

Remonter d'un cran dans la hiérarchie est très simple. On prend simplement la dernière miette et on assemble le nouveau point focal à partir du précédent et de la miette. Ainsi :

```
fsUp :: FSZipper -> FSZipper
fsUp (item, FSCrumb name ls rs:bs) = (Folder name (ls ++ [item] ++ rs), bs)
```

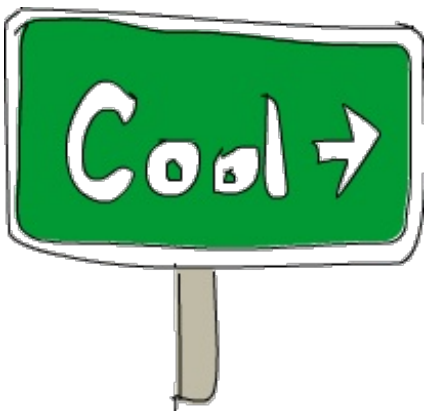
Puisque notre miette savait le nom du dossier parent, ainsi que les éléments qui venaient avant l'élément sur lequel on s'était focalisé (c'est le **`ls`**) ainsi que ceux venant après (c'est le **`rs`**), remonter d'un cran était simple.

Qu'en est-il d'avancer plus profondément dans le système de fichiers ? Si nous sommes dans **`"root"`** et qu'on veut se focaliser sur **`"dijon_poupon.doc"`**, la miette qu'on laissera devra inclure le nom **`"root"`** ainsi que les éléments précédant **`"dijon_poupon.doc"`** et ceux qui viennent après.

Voici une fonction qui, étant donné un nom, se focalise sur un fichier ou un dossier situé dans le dossier courant :

```
import Data.List (break)

fsTo :: Name -> FSZipper -> FSZipper
fsTo name (Folder folderName items, bs) =
    let (ls, item:rs) = break (nameIs name) items
    in  (item, FSCrumb folderName ls rs:bs)

nameIs :: Name -> FSItem -> Bool
nameIs name (Folder folderName _) = name == folderName
nameIs name (File fileName _) = name == fileName
```

`fsTo` prend un **`Name`** et un **`FSZipper`** et retourne un nouveau **`FSZipper`** qui se focalise sur le fichier ou dossier portant ce nom. Ce fichier doit être dans le dossier courant. Cette fonction ne va pas le chercher dans tous les endroits possibles, elle regarde seulement dans le dossier courant.



Tout d'abord, on utilise **`break`** pour détruire la liste des éléments d'un dossier en ceux qui précèdent le fichier qu'on cherche et ceux qui le suivent. Si vous vous en souvenez, **`break`** prend un prédicat et une liste et retourne une paire de listes. La première liste de la paire contient les premiers éléments pour lesquels le prédicat a retourné **`False`**. Puis, dès que le prédicat retourne **`True`** pour un élement, elle place cet élément et tout le reste de la liste dans la deuxième composante de la paire. On a créé une fonction auxiliaire nommée **`nameIs`** qui prend un nom et un élément d'un système de fichiers et retourne **`True`** si le nom correspond.

À présent, **`ls`** est une liste contenant les éléments qui précèdent l'élément qu'on cherchait, **`item`** est l'élément en question, et **`rs`** est la liste des éléments venant après lui dans ce dossier. Muni de ceci, il ne nous reste plus qu'à présenter l'élément trouvé par **`break`** comme le point focal et à construire la miette appropriée.

Remarquez que si le nom qu'on cherche ne se trouve pas dans le dossier, le motif **`item:rs`** essaiera de filtrer une liste vide, et on aura une erreur. Si notre point focal n'est pas un dossier mais un fichier, on aura une erreur et le programme plantera également.

On peut maintenant se déplacer de haut en bas dans notre système de fichiers. Partons de la racine et dirigeons-nous vers le fichier **`"skull_man(scary).bmp"`** :

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsTo "skull_man(scary).bmp"
```

`newFocus` est à présent un zippeur qui est focalisé sur le fichier **`"skull_man(scary).bmp"`**. Récupérons le premier élément du zippeur (le point focal lui-même) et vérifions si c'est bien vrai :

```
ghci> fst newFocus
File "skull_man(scary).bmp" "Yikes!"
```

Remontons d'un cran pour nous focaliser sur son voisin **`"watermelon_smash.gif"`** :

```
ghci> let newFocus2 = newFocus -: fsUp -: fsTo "watermelon_smash.gif"
ghci> fst newFocus2
File "watermelon_smash.gif" "smash!!"
```

Manipuler notre système de fichiers

À présent qu'on sait naviguer dans notre système de fichiers, le manipuler est très simple. Voici une fonction qui renomme le fichier ou dossier courant :

```
fsRename :: Name -> FSZipper -> FSZipper
fsRename newName (Folder name items, bs) = (Folder newName items, bs)
fsRename newName (File name dat, bs) = (File newName dat, bs)
```

On peut maintenant renommer `"pics"` en `"cspi"` :

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsRename "cspi" -: fsUp
```

On est descendu jusqu'au dossier `"pics"`, on l'a renommé, et on est remonté.

Pourquoi pas une fonction qui prend un nouvel élément, et l'ajoute au dossier courant ? Admirez :

```
fsNewFile :: FSItem -> FSZipper -> FSZipper
fsNewFile item (Folder folderName items, bs) =
    (Folder folderName (item:items), bs)
```

C'était du gâteau. Remarquez que ceci planterait si l'on essayait d'ajouter un élément alors que notre point focal était un fichier plutôt qu'un dossier.

Ajoutons un fichier au dossier `"pics"` et retournons ensuite à la racine :

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsNewFile (File "heh.jpg" "lol") -: fsUp
```

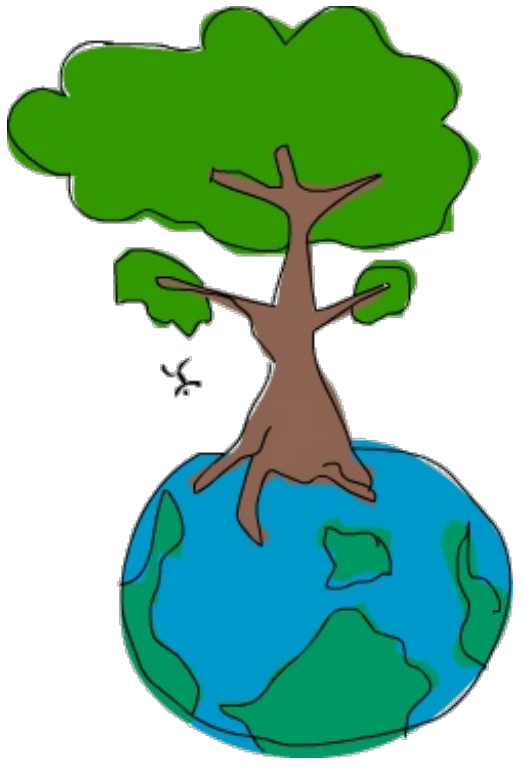
Ce qui est vraiment cool, c'est que lorsqu'on modifie notre système de fichiers, on ne le modifie pas vraiment en lieu et place, mais on retourne plutôt un nouveau système de fichiers. Ainsi, on a à la fois accès à l'ancien système de fichiers (ici, `myDisk`) et au nouveau (la première composante de `newFocus`). Ainsi, en utilisant des zippeurs, on obtient du versionnage sans effort, ce qui signifie qu'on peut toujours se référer aux anciennes versions de nos structures de données même après les avoir modifiées, pour ainsi dire. Ce n'est pas unique aux zippeurs, c'est simplement une propriété d'Haskell dûe au fait que ses structures de données sont immuables. Cependant, avec les zippeurs, on a la capacité de se déplacer efficacement dans des structures de données, et c'est là que la persistance des structures de données d'Haskell commence à dévoiler son potentiel.

Attention à la marche

Jusqu'ici, lorsqu'on traversait nos structures de données, qu'elles soient des arbres binaires, des listes ou des systèmes de fichiers, on ne se préoccupait pas trop d'avancer un pas trop loin et de tomber dans le vide. Par exemple, notre fonction `goLeft` prend un zippeur et un arbre binaire et déplace le point focal vers son sous-arbre gauche :

```
goLeft :: Zipper a -> Zipper a
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

Et si l'arbre duquel on avançait était vide ? C'est-à-dire, et si ce n'était pas un `Node`, mais un `Empty` ? Dans ce cas, on obtiendrait une erreur à l'exécution parce que le filtrage par motif échouerait et parce qu'on n'avait pas mis de motif gérant le cas d'un arbre vide, qui ne contient pas de sous-arbres. Jusqu'ici, on a seulement supposé que l'on n'essaierait jamais de nous focaliser sur le sous-arbre gauche d'un arbre vide, puisque celui-ci n'existe pas. Mais, aller dans le sous-arbre gauche d'un arbre vide est illogique, et jusqu'alors, on a ignoré ce fait paresseusement.



Et si l'on était à la racine d'un arbre, sans miettes, et qu'on essayait de rebrousser chemin ? La même chose aurait lieu. Il semblerait que quand on utilise les zippeurs, chaque pas pourrait être notre dernier pas (une musique menaçante se fait entendre). En d'autres termes, tout mouvement peut réussir, mais peut aussi échouer. Cela ne vous rappelle rien ? Bien sûr, les monades ! Plus spécifiquement, la monade `Maybe` qui ajoute un contexte d'échec potentiel à nos valeurs.

Utilisons donc la monade `Maybe` pour ajouter un contexte possible d'échec à nos mouvements. On va prendre les fonctions qui marchent sur notre zippeur d'arbres binaires et les rendre monadiques. D'abord, occupons-nous des échecs possibles de `goLeft` et `goRight`. Jusqu'ici, l'échec de fonctions pouvant échouer était toujours reflété dans leur résultat, il en va de même ici. Voici `goLeft` et `goRight` pouvant échouer :

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (Node x l r, bs) = Just (l, LeftCrumb x r:bs)
goLeft (Empty, _) = Nothing

goRight :: Zipper a -> Maybe (Zipper a)
goRight (Node x l r, bs) = Just (r, RightCrumb x l:bs)
goRight (Empty, _) = Nothing
```

Cool, maintenant, si l'on essaie d'avancer vers la gauche d'un arbre vide, on obtient **`Nothing`** !

```
ghci> goLeft (Empty, [])
Nothing
ghci> goLeft (Node 'A' Empty Empty, [])
Just (Empty,[LeftCrumb 'A' Empty])
```

Ça a l'air bon ! Et pour rebrousser chemin ? Le problème avait lieu lorsqu'on essayait de remonter alors qu'on n'avait plus de miettes, ce qui signifiait qu'on était déjà à la racine de notre arbre. Voici la fonction **`goUp`** qui lance une erreur si l'on ne reste pas dans les limites de notre arbre :

```
goUp :: Zipper a -> Zipper a
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

Modifions-là pour échouer gracieusement :

```
goUp :: Zipper a -> Maybe (Zipper a)
goUp (t, LeftCrumb x r:bs) = Just (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = Just (Node x l t, bs)
goUp (_, []) = Nothing
```

Si l'on a des miettes, tout va bien et on retourne un nouveau point focal réussi, mais si ce n'est pas le cas, on retourne un échec.

Auparavant, ces fonctions prenaient des zippeurs et retournaient des zippeurs, ce qui signifiait qu'on pouvait les chaîner ainsi pour se balader :

```
gchi> let newFocus = (freeTree,[]) -: goLeft -: goRight
```

Mais à présent, au lieu de retourner un **`Zipper a`**, elles retournent un **`Maybe (Zipper a)`**, et chaîner les fonctions ainsi ne fonctionnera plus. On avait eu un problème similaire quand on s'occupait de notre funambule dans le chapitre sur les monades. Lui aussi avançait un pas à la fois et chaque pas pouvait échouer parce qu'un tas d'oiseau pouvait atterrir sur un côté de sa perche d'équilibre et le faire tomber.

Maintenant, nous sommes l'arroseur arrosé, parce que c'est nous qui essayons de nous déplacer, et on traverse le labyrinthe qu'on a nous-même créé. Heureusement, on peut apprendre de l'expérience du funambule et faire ce qu'il a fait, c'est-à-dire échanger l'application de fonctions normale contre **`>>=`**, qui prend une valeur dans un contexte (dans notre cas, un **`Maybe (Zipper a)`**, qui a un contexte d'échec potentiel) et la donne à une fonction en s'assurant que le contexte est bien géré. Ainsi, tout comme avec notre funambule, on va échanger nos **`-:`** contre des **`>>=`**. Très bien, on peut à nouveau chaîner nos fonctions ! Observez :

```
ghci> let coolTree = Node 1 Empty (Node 3 Empty Empty)
ghci> return (coolTree,[]) >>= goRight
Just (Node 3 Empty Empty,[RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight
Just (Empty,[RightCrumb 3 Empty,RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight >>= goRight
Nothing
```

On a utilisé **`return`** pour placer un zippeur dans un **`Just`**, puis utilisé **`>>=`** pour donner cela à notre fonction **`goRight`**. D'abord, on crée un arbre qui a un sous-arbre gauche vide, et un sous-arbre droit avec deux sous-arbres. Quand on essaie d'aller à droite une fois, c'est un succès, parce que l'opération est logique. Aller deux fois à droite est aussi correct, on se retrouve avec un sous-arbre vide en point focal. Mais aller à droite trois fois n'a pas de sens, parce qu'on ne peut pas aller à droite dans un sous-arbre vide, c'est pourquoi le résultat est **`Nothing`**.

On a désormais muni nos arbres d'un filet de sécurité qui nous attraperait si l'on devait tomber. Ouah, cette métaphore était parfaite.

Notre système de fichiers a aussi beaucoup de cas pouvant échouer, comme lorsque l'on essaie de se focaliser sur un fichier inexistant. Comme exercice, vous pouvez munir notre système de fichiers de fonctions qui échouent gracieusement en utilisant la monade **`Maybe`**.

⬅ Et pour quelques monades de plus Table des matières