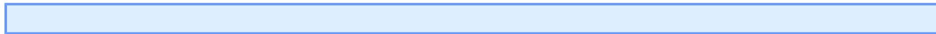
Developpez.com
Club des développeurs

Tutorial Haskell pour le développeur C

par Eric Etheridge (Auteur) Corentin Dupont (traducteur) (Home Page)

Date de publication : 05/03/2008

Dernière mise à jour : 06/05/2008



I - Introduction
- I-A - Résumé
- I-B - Téléchargements
- I-C - Licence
- I-D - Préface et notes sur le style

II - Un langage surprenant !
- II-A - Les Bizarreries d'Haskell
- II-B - Les Entrées et Sorties
- II-C - Une petite introduction aux types
- II-D - La compréhension de liste
- II-E - La lumière sur 'fibs'

III - Introduction aux fonctions
- III-A - L'ordre des choses
- III-B - Les types et les fonctions
- III-C - Les types polymorphes
- III-D - Les fonctions, enfin !

IV - C'est parti pour les fonctions !
- IV-A - Les motifs
- IV-B - Après les motifs, les gardes
- IV-C - Le 'If'
- IV-D - L' Indentation
- IV-E - Les lambdas fonctions
- IV-F - Les types polymorphes et les constructeurs de types
- IV-G - La Monade IO
- IV-H - Le 'main' à la loupe

V - Haskell et vous
- V-A - Où sont passées les boucles " for "?
- V-B - Types avancés
- V-C - Pour conclure

VI - Le mot de la fin
- VI-A - La 'Transparence référentielle' est-elle vraiment utile ?
- VI-B - Contacts et références
- VI-C - Digression finale

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

I - Introduction

I-A - Résumé

La plupart des gens sont habitués aux langages impératifs comme C, C++, Java, Python et Pascal.

Pour les étudiants en informatique, Haskell est étrange et incompréhensible.

Ce tutorial suppose que le lecteur connaît le C/C++, le Python, le Java ou le Pascal. Je l'ai écrit parce qu'il m'a semblé qu'aucun autre tutorial n'a été fait pour aider les étudiants à passer d'un langage comme C/C++ ou Java à Haskell.

Peut-être avez-vous regardé du coté des manuels de références comme " A Gentle Introduction to Haskell ", sans toutefois arriver à tout déchiffrer ?...

Haskell n'est pas " un peu différent " et ne vous prendra pas " un peu de temps " pour arriver à le comprendre. Il est très différent des autres langages et vous ne pourrez pas le maîtriser du jour au lendemain. Néanmoins, j'espère que ce tutorial pourra vous aider.

Je ferais pas mal de pauses tout au long de ce tutorial, parce que Haskell, ça fait mal. En ce qui me concerne, j'ai eu besoin de beaucoup de pauses et j'ai eu beaucoup de migraines en essayant de comprendre ce langage !

Haskell possède à la fois plus de " flexibilité " et plus de " contrôle " que la plupart des langages. Je ne connais rien qui puisse battre la capacité de contrôle du C, mais Haskell peut faire aussi bien que le C dans tous les domaines à moins que vous vouliez contrôler des octets particuliers en mémoire. C'est pour cette raison que Haskell n'est pas seulement " bon ", il est " puissant ".

J'ai écrit ce tutorial parce que j'ai eu beaucoup de mal à apprendre l'Haskell, mais maintenant je l'adore. " Lorsque j'étais étudiant, je pensais : "Haskell, c'est dur !" ; " On ne peut pas coder comme on veux !" ; " J'ai mal à la tête ! " " !" ; "Ca manque de bonnes références ! ". En fait, de bonnes références, il y en a, mais elles ne répondent pas au vrai problème : Les développeurs ne connaissent que les langages impératifs comme le C.

I-B - Téléchargements

Ndt : ADU

I-C - Licence

J'ai décidé d'être précis dans le choix de la licence de ce tutorial. Pour faire simple, vous pouvez faire tout ce que vous voulez avec ce tutorial tant que mon nom figure dessus : vous pouvez le modifier, le redistribuer ou vendre des travaux qui s'en inspirent. En particulier, vous pouvez l'utiliser dans le cadre d'une formation professionnelle. Je serais enchanté d'apprendre qu'une entreprise intéressée par Haskell a utilisé ce tutorial comme support de formation. Donc n'hésitez pas ! De toute manière, tout travail dérivé restera sous cette licence. " Share alike ". Merci, Creative Commons. Ce lien vous mènera aux mentions légales :

Creative Commons Attribution-ShareAlike 2.5 License.

Note du traducteur : Cette version traduite du tutorial reste effectivement sous la licence Creative Commons " Share Alike ". Si vous utilisez ou modifiez cette version du tutorial en Français, merci de mentionner le nom de l'auteur ainsi que du traducteur !

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

I-D - Préface et notes sur le style

Ceci n'est pas un ouvrage de référence du langage Haskell. C'est un tutorial fait pour aider les gens qui ont des difficultés à comprendre Haskell. Ce tutorial est destiné aux gens qui, comme moi, ont besoin de comprendre les concepts avant de s'attaquer à la compréhension du code. Haskell permet de faire des choses facilement et simplement, mais il n'est ni facile ni simple et peut être très ardue pour un débutant. Vous ne pourrez pas comprendre du code Haskell d'emblée au premier coup d'#il. En écrivant ce tutorial j'ai essayé de couvrir tous les aspects les plus courants d'Haskell qui posent problème.

Au fil du tutorial, une chose devrait devenir de plus en plus claire : Haskell révèle sa vraie puissance lorsque vous vous attaquez aux problèmes difficiles. C'est pour cela que j'utiliserai quelques exemples non triviaux dans ce tutorial. Ne vous inquiétez pas si vous ne comprenez pas les solutions à la première lecture# Haskell n'est pas un langage gadget et même un nombre réduit de fonctions pourront mettre en #uvre des fonctionnalités complexes d'Haskell.

Voilà un dilemme pour le formateur : dois-je utiliser des exemples ridiculement simples pour couvrir un seul sujet à la fois, ou dois-je décrire quelque chose d'effectivement utile et essayer d'expliquer chaque élément et leurs articulations ?

Beaucoup de tutoriaux et de cours ont choisi la première solution, je préfère la seconde. Cela implique souvent des explications plus longues pour chaque exemple.

La plupart du temps, un concept doit d'abord être exprimé en termes extrêmement simples, et ensuite être ré- expliqué plus tard, après avoir abordé d'autres sujets connexes. Au fur et à mesure de votre lecture de ce tutoriel, rappelez-vous ceci : La vraie puissance d'Haskell réside dans le fait que tous les éléments s'emboîtent parfaitement les uns dans les autres, et pas seulement parce que ce sont des bons éléments, pris séparément.

La syntaxe et les conventions de nommages utilisés dans ce tutoriel sont celles utilisées dans le source d'Haskell et ses librairies, ainsi que celles que j'ai appris à l'école. Les programmes en Haskell ont tendance à être courts, mais avec une grande portée. Je recommande d'utiliser des noms de variables descriptifs, même pour les indices.

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

II - Un langage surprenant !

II-A - Les Bizarreries d'Haskell

Première chose, Haskell n'a pas d'opérateur d'affectation. Si cette phrase vous parait absurde, ne vous inquiétez pas et poursuivez votre lecture. Le code suivant est impossible en Haskell :

```
int a
a := 4
print a
a := 5
print a
```

```
> 4
> 5
```

Le mode de programmation ci-dessus, c'est-à-dire " faire une variable, y mettre une donnée, l'utiliser, puis remplacer la donnée, l'utiliser à nouveau " n'existe pas en Haskell classique. Ceux parmi vous qui ont utilisé LISP ou Scheme doivent être familiers avec ce concept, mais je suis sûr que les autres sont stupéfaits. Voici comment Haskell marche, à nouveau en pseudo code :

```
print a

int a
a = 5

> 5
```

Ou alors:

```
int a
a = 5

print a

>       5
```

L'ordre des instructions n'a pas d'importance. Il y a aussi une raison pour laquelle le premier exemple utilisait ' :=' et le second '='. Dans les langages impératifs, stocker une donnée est une opération, et cette opération est faite dans une séquence. Dans les langages fonctionnels comme Haskell, la signification du signe "égal" correspond exactement à ce que son nom indique. En d'autres termes, chaque variable est égale à sa valeur non seulement après l'affectation, mais aussi avant et à n'importe quel point de l'exécution.

A présent, vous allez me dire, " Super Eric, mais qu'est ce que je vais faire d'un langage ou tout est codé en dur ? Est-ce que je dois définir chaque variable avec sa valeur finale directement dans mon code ? S'il n'y a plus aucun calcul à faire, mon ordinateur risque d'être au chômage ! ". Et vous auriez raison, on ne programme pas en connaissant les résultats à l'avance. La magie d'Haskell est que vous n'avez pas besoin de stocker de données pour retourner un résultat.

Je vais faire autant de pauses que nécessaire dans ce tutoriel, car apprendre l'Haskell provoque des migraines, du moins chez moi. J'ai eu besoin de pauses, et ma tête me faisait mal lorsque j'essayais de déchiffrer certains concepts.

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

Enfin bref, revenons sur cette affirmation : vous n'avez pas besoin de stocker de données pour retourner un résultat. Voici une fonction exemple en C :

```
int foo (int bar) {
        int result;
        result = bar * 10 + 4;
        return result;
}
```

On peut récrire la fonction ainsi :

```
int foo (int bar) {
        return bar * 10 + 4;
}
```

Ce sont les mêmes, mais la seconde est plus courte, et plus claire. Avec une telle fonction, vous pouvez dire : " La valeur de foo(x) est égale à (x * 10 + 4)." Ou, plus simplement, "foo(x) = x * 10 + 4". Bien sûr, vous allez me dire " La plupart des fonctions ne sont pas si simples ".

C'est vrai, mais croyez moi : Haskell a beaucoup plus d'outils permettant d'écrire des fonctions q ue les autres langages, et beaucoup de choses complexes paraissent simples en Haskell. La clef pour utiliser ces outils sera de changer de mode de pensée : il faut passer de l'optique 'faire une donnée, puis la modifier' à 'définir une fonction qui retourne le résultat, puis lui appliquer les entrées'.

II-B - Les Entrées et Sorties

Nous reviendrons sur le mode de pensée plus tard. Haskell est si différent du C/C++ que pas mal de ses concepts ont une signification uniquement si on les met en relations avec d'autres. C'est pour cette raison que je dois d'abord balayer un certain nombre de concepts en surface avant de pouvoir les aborder en profondeur.

Allons-y! La question à laquelle tout le monde pense est : " Comment fonctionne les E/S ? " ou " Quels sont ces outils qu'on nous promet ? ". Les E/S sont l'une des parties les plus compliquées d'Haskell, je décrirai plus tard comment elles fonctionnent et comment les implémenter dans GHC.

En attendant, n'hésitez pas à utiliser GHCi ou Hugs pour les exemples. Ils disposent d'une invite interactive où vous pouvez taper vos expressions, par exemple une fonction avec ses paramètres. Ils évalueront le résultat immédiatement à la suite. De plus, les assignations de variables comme 'a=4' restant actives après les avoir évaluées, les exemples devraient fonctionner.

Si vous voulez écrire vos propres fonctions, il est préférable de les écrire dans un fichier source pour ne pas les perdre et de charger celui-ci sous GHCi. Utiliser GHC lui-même suppose de connaître quelques outils d'Haskell plus complexes, nous aborderons ce point plus tard.

Pour utiliser Hugs et GHCi avec vos propres fonctions, vous devrez écrire vos propres sources Haskell et les charger dans l'interpréteur. Généralement, cela fonctionne comme ceci :

1. Ouvrir un éditeur de texte et écrire le code Haskell.
2. Sauver le code dans un fichier avec l'extension '.hs', par exemple, 'test.hs'.
3. Lancer Hugs ou GHCi dans le même répertoire que le fichier.
4. Tapez ' :l test.hs' dans Hugs ou GHCi.

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

5 Les sources qui nécessitent des modules comme Data.Maybe doivent avoir 'import Data.Maybe' au début du fichier.

Notez que le module 'Prelude' est automatiquement importé. Ce module contient tous les éléments de base du langage.

II-C - Une petite introduction aux types

Pour continuer, parlons de ces fameux outils. La puissance d'Haskell réside dans la possibilité de définir des fonctions facilement et clairement.

```
int foo (int bar) {
        return bar * 10 + 4;
}
```

En Haskell, pour écrire cette fonction foo, vous écrirez:

```
foo bar = bar * 10 + 4
```

C'est tout, à l'exception du type:

```
foo :: Int -> Int
```

Le type se lit: "foo est de type Int vers Int", ce qui signifie qu'il prend un Int et retourne un Int. Ecrit ensemble, cela donne :

```
foo :: Int -> Int
foo bar = bar * 10 + 4
```

Définir des fonctions et des types est l'essentiel du boulot en Haskell, et nécessite à peu près le même temps. Haskell propose différentes techniques pour définir des fonctions, et les tutoriaux mentionnés plus haut ne les présentent pas toutes correctement. Nous y reviendrons plus tard, une fois que nous serons suffisamment outillé.

II-D - La compréhension de liste

Ceux d'entre vous qui sont familiers avec le C savent que les pointeurs sont de première importance dans ce langage. De même dans la plupart des langages impératifs la structure la plus importante est le tableau, une séquence de valeurs stockées (habituellement) dans l'ordre en mémoire.

Haskell propose des tableaux, mais l'objet le plus utilisé est la liste. En Haskell la liste n'est pas forcément rangée dans l'ordre en mémoire et seule la tête est accessible. Cela peut paraître atroce, mais Haskell a des capacités tellement particulières qu'il est plus naturel d'utiliser des listes, c'est même plus rapide. Commençons avec le code C de la suite de Fibonacci commençant par zéro :

```
 int fib (int n) {
 int a = 0, b = 1, i, temp;
 for (i = 0; i < n; i++) {
  temp = a + b;
```


Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

```
    a = b;
    b = temp;
   }
   return a;
  }
```

Ce code est parfait pour calculer une valeur donnée, mais les ennuis commencent lorsque vous voulez créer la séquence complète:

```
int * fibArray(int n) {
 int * fibs;
 fibs = (int *)malloc((sizeof int) * n);
 for (i = 0; i < n; i++) {
  fibs[i] = a;
  temp = a + b;
  a = b;
  b = temp;
 }
 return fibs;
}
```

Quand je dis que les ennuis commencent, je veux dire qu'il y a quelque chose qui est inclus dans la fonction et qui ne devrait pas l'être: la taille de la liste. La suite de Fibonacci est infinie, et le code ci-dessus n'en représente qu'une partie. Cela n'a pas l'air trop grave, à moins que vous ne sachiez pas au départ de combien de valeurs vous avez besoin.

En Haskell, 'fib', la fonction pour calculer une valeur de la suite de Fibonacci peut être écrite comme suit :

```
fib :: Int -> Int
fib n = fibGen 0 1 n

fibGen :: Int -> Int -> Int -> Int
fibGen a b n = case n of
 0 -> a
 n -> fibGen b (a + b) (n - 1)
```

C'est un peu mieux qu'en C, mais guère plus. Notez que le type de fibGen est " Int vers Int vers Int vers Int ", signifiant qu'elle prend trois Int et retourne un Int. Nous en dirons plus là-dessus plus tard. Notez aussi qu'on utilise une fonction récursive. La récursivité est omniprésente en Haskell. La plupart de vos " boucles " se transformeront en récursivité, à moins qu'elles n'utilisent des outils plus sophistiqués comme ci-dessus.

Le vrai plus par rapport au C vient en écrivant ceci:

```
fibs :: [Int]
fibs = 0 : 1 : [ a + b | (a, b) <- zip fibs (tail fibs)]
```

N'ayez pas peur. Une fois que vous aurez compris cette fonction, vous aurez compris au moins la moitié des subtilités d'Haskell. Commençons par le début. En Haskell, les listes s'écrivent comme suit :

```
[ 4, 2, 6, 7, 2 ]
```

Cette liste contient 4, puis 2, puis 6, etc. L'opérateur ':' s'utilise pour ajouter une valeur en tête de liste (à gauche). Donc :

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

```
temp = 1 : [ 4, 2, 5 ]
```

est la liste [1, 4, 2, 5].

Cela signifie que dans le code ci-dessus, fibs est une liste d'Int, et que ses deux premières valeurs sont zéro et un. Jusqu'ici tout va bien, au moins les deux premières valeurs seront justes, si la chose compile. Mais l'autre partie a l'air franchement bizarre. C'est un outil très pratique appelé 'compréhension de liste'. Au lieu de créer des cases et d'y mettre les bonnes valeurs, vous pouvez définir les bonnes valeurs. La compréhension de liste fonctionne comme ça :

```
[ func x | x <- list, boolFunc x ]
```

Au milieu, il y a une 'liste', et elle crache des valeurs appelées x. Ce sont les valeurs de la liste dans l'ordre. Si 'boolFunc x' est vrai, alors x sera utilisé dans la nouvelle liste. boolFunc n'apparaît pas dans 'fibs', mais je l'ai inclut ici car elle peut être très utile. 'func x' applique une certaine fonction à la valeur de x, et le résultat est inséré dans le résultat final, en supposant que 'boolFunc x' était vrai. Voici un exemple de liste et son utilisation de la compréhension de liste :

```
nums :: [Int]
nums = [ 4, 2, 6, 8, 5, 11 ]

[ x + 1 | x <- nums ]
= [ 5, 3, 7, 9, 6, 12 ]

[ x * x | x <- nums, x < 7 ]
= [ 16, 4, 36, 25 ]

[ 2 * x | x <- 9 : 1 : nums ]
= [ 18, 2, 8, 4, 12, 16, 10, 22 ]

[ "String" | x <- nums, x < 5 ]
= [ "String", "String" ]
```

Notez que l'ordre est conservé. C'est très important dans notre exemple. Notez également que le type de la compréhension de liste n'est pas nécessairement celui de nums, pas plus que x ne doivent être utilisé dans la fonction func. Retournons à 'fibs'.

II-E - La lumière sur 'fibs'

Nous travaillions sur une définition de la suite de Fibonacci. Voici l'exemple à nouveau :

```
fibs :: [Int]
fibs = 0 : 1 : [ a + b | (a, b) <- zip fibs (tail fibs)]
```

Alors, qu'est ce que ce drôle de '(a, b)' et 'zip fibs (tail fibs)' et tout ça ? Eh bien, Haskell a un système de typage plus expressif que la plupart des autres langages. Comme en Python, '(a, b)' est un tuple, signifiant que deux valeurs sont scotchés ensemble. C'est une manière pratique de stocker et de passer plusieurs valeurs, à la manière des structures. Pour transmettre un groupe de valeurs, ajoutez simplement des parenthèses et autant de virgules que nécessaires. Vous devrez seulement être cohérent, donc utiliser le bon type. Par exemple, '(a, b)' est du type '(Int, Int)', soit :

```
(a, b) :: (Int, Int)
```


Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

'zip fibs (tail fibs)' sera donc du type '[(Int, Int)]', une liste de 2-tuples : un Int et un Int. Ou encore :

```
zip fibs (tail fibs) :: [(Int, Int)]
```

GHCi et Hugs vous donne le type à l'aide de la commande ':t' suivie d'une variable ou d'une fonction. C'est très pratique.

Que signifie 'zip'? Sont type et sa signification sont décrits ici:

Prelude, Section: Zipping and Unzipping Lists

'zip' comme son nom l'indique prend deux listes et les 'zippent' ensemble, retournant une liste de tuples. Le membre de gauche de chaque tuple est un élément de la première liste, de même pour le membre de droite.

```
zip [ 1, 3, 6, 2 ] [ "duck", "duck", "duck", "goose" ]
= [ (1, "duck"), (3, "duck"), (6, "duck"), (2, "goose") ]
```

Et pour '(tail fibs)'? 'tail' est plutôt simple : il élimine le premier élément de la liste et retourne le reste. Cette affirmation peut être trompeuse. 'fibs' n'est pas modifiée par l'utilisation de 'tail' : comme je l'ai dit plus tôt, Haskell n'a pas d'opérateur d'affectation. 'tail' se contente de calculer le résultat et de le retourner, sans modifier 'fibs'.

```
tail [ 10, 20, 30, 40, 50 ]
= [ 20, 30, 40, 50 ]
```

Bon, apparemment 'zip fibs (tail fibs)' est correctement typé, mais quel est ce type ?

```
fibs :: [Int]
fibs = 0 : 1 : [ a + b | (a, b) <- zip fibs (tail fibs)]
```

Le premier paramètre de zip est 'fibs', qui est aussi ce qui est définit par l'expression! Hé, on peut faire ça ? Oui, on peut. Voyez vous, 'fibs' est la liste entière, avec le 0 et le 1 au début. Donc les deux premiers tuples créés par la fonction zip auront un 0 et un 1 sur le membre de gauche. Alors, qu'est ce que 'zip fibs (tail fibs)'? Eh bien la première valeur est bien sûr (0, 1). Pourquoi? Parce que le premier élément dans fibs est 0, et le premier élément dans (tail fibs) est 1, le second de fibs.

Et le deuxième élément de fibs (tail fibs) ? C'est (1,1). Et d'où viens le 1 de droite ? C'est la troisième valeur de fibs, que l'on vient de calculer. La première valeur de zip fibs (tail fibs) est (0, 1), qui est '(a, b)' dans la compréhension de liste, donc la première valeur dans cette compréhension est 0 + 1, soit 1. C'est donc la troisième valeur de fibs, etc.

Vous avez bien tout capté? La définition de fibs s'évalue elle-même tout en se calculant. Alors pourquoi n'y a-t-il pas d'erreurs dues à des valeurs indéfinies par exemple ? C'est possible grâce à la paresse d'Haskell. De plus, l'évaluation est toujours un cran en avance sur le calcul, donc l'évaluation peut toujours aller aussi loin que nécessaire pour le prochain calcul.

Dernier point, la liste sera infinie. Bien sûr, aucun ordinateur ne peut manipuler une quantité de données infinie. Alors, quelle quantité est réellement présente en mémoire? La réponse est simple : avant que vous ne lisiez effectivement la liste, il y a seulement le 0 et le 1, ainsi que la fonction permettant d'en générer plus. Après en avoir lu jusqu'à un certain point, fibs sera générée jusqu'à ce point et pas plus. Puisque fibs est définie globalement, les valeurs

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

calculées resteront en mémoire, rendant les prochains accès très rapides. Essayez ceci dans GHCi ou Hugs et vous verrez ce que je veux dire.

```
fibs !! 2
fibs !! 4
fibs !! 30
fibs !! 30
fibs !! 6
fibs !! 20
fibs !! 30
take 10 fibs
```

'!!' est l'opérateur d'index. Il parcours la liste et retourne le nième élément, en commençant par zéro comme en C/C++ et Python. 'take 10 fibs' retournera les 10 premières valeurs de fibs. Soyez prudent, fibs est de longueur infinie. Si vous tapez juste 'fibs', le programme se s'arrêtera pas.

Et comment se fait-il que cette liste ne soit pas évaluée en entier mais seulement aussi loin que ce qu'on affiche? Haskell est 'paresseux', ce qui signifie qu'il n'effectue pas une tâche s'il n'en a pas besoin. Les programmeurs C ont les opérateurs '&&' et '||' qui fonctionnent comme des 'courts circuits' : la partie de droite n'est pas évaluée si ce n'est pas nécessaire. Cela permet pas mal d'astuces, comme de ne pas dé-référencer un pointeur null.

L'intégralité du langage applique ce mécanisme de court-circuit, même dans les fonctions que vous écrivez vous-même. Cela peut paraître étrange. Mais ça deviendra de plus en plus important, en particulier lorsque nous aborderons les outils d'Haskell.

Cela nous amène à l'une des autres bizarreries d'Haskell : **Il est souvent plus facile de coder la solution d'un problème dans le cas général, plutôt que d'écrire la fonction qui produit une valeur particulière**. Il faudra vous y habituer, et vous aurez sans doute à y revenir encore. Et encore.

Bien, voici venu le moment de faire une pause. Si vous avez compris tout ça, ou que du moins vous allez bientôt le comprendre après quelques essais sous Hugs, eh bien vous avez fait la moitié du chemin. Prêt pour l'autre moitié ?

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

III - Introduction aux fonctions

III-A - L'ordre des choses

Je voudrais tout d'abord faire une petite remarque sur l'écriture des fonctions récursives en Haskell.

Reprenons l'exemple fib / fibGen:

```
fib :: Int -> Int
fib n = fibGen 0 1 n

fibGen :: Int -> Int -> Int -> Int
fibGen a b n = case n of
 0 -> a
 n -> fibGen b (a + b) (n - 1)
```

J'ai écrit le type de fib en premier, puis le type et la définition de fibGen, et enfin la définition de fib.

Programmer en Haskell nécessite souvent d'écrire des fonctions récursives. Ces fonctions récursives ont souvent besoin de sous-fonctions, qui font soit l'opération principale de la récursion, soit des tâche annexes de nettoyage de la fonction principale. Dans les deux cas, la sous-fonction peut être écrite plus tard, après que l'opération de récursion principale ainsi que les conditions de fin soient clairement définies.

C'est généralement une bonne idée de se concentrer sur les aspects les plus importants d'un code lorsque l'on programme. Le design du langage vous y pousse. L'écriture de sous-fonctions, comme ici pour le paramétrage (fib appel fibGen avec '0 1 n'), peut attendre que la fonction principale elle-même soit finie. Haskell permet d'écrire des fonctions triviales si rapidement qu'elles peuvent être ignorées pour se concentrer sur le problème principal. Cela va peut-être changer la manière dont vous codez, et sans doutes dans le bon sens.

III-B - Les types et les fonctions

A ce stade, vous devriez deviner quel est le deuxième gros morceau de la syntaxe d'Haskell. Il s'agit des fonctions.

Alors, qu'est ce qu'une fonction? Comme indiqué dans les préliminaires de ce tutoriel, nous ferons la comparaison avec le C/C++. En C, une fonction est une séquence de commandes, elles sont appelées à l'exécution et on leur passe des paramètres, elles héritent de la portée de l'espace de noms où elles sont écrites, et retournent une valeur à l'appelant. En Haskell l'essentiel de ceci est vrai, excepté bien sûr que les fonctions en Haskell ne sont pas des séquences d'événements mais des expressions et des définitions. C'est une grosse différence entre C et Haskell, et ça a un gros impact sur la flexibilité des fonctions. En C, les fonctions prennent des paramètres et retournent une unique valeur. Nous avons déjà vu qu'il y a plein de manière de regrouper des valeurs en Haskell, comme dans d'autres langages.

Les deux principales manières de regrouper les valeurs sont les listes et les tuples, puisqu'ils peuvent servirent de types de retour pour une fonction. En deux mots, en Haskell les listes sont de taille variable et contiennent des éléments du même type, et les tuples sont de longueur fixe et contiennent des valeurs de types différents. Voici un exemple de type de fonction retournant un tuple :

```
splitAt :: Int -> [a] -> ([a], [a])
```


Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

'splitAt' prend un Int et une liste et retourne un tuple. La liste de gauche dans le tuple contient les n premières valeurs dans la liste, et la liste de droite contient le reste. Cette fonction appartient au Prelude, sa description peut être trouvée ici :

Prelude, Section: Sublists

Nous avons déjà vu les listes lorsque j'ai donné le type de fibs:

```
fibs :: [Int]
```

Les nombres de Fibonacci croissant rapidement, j'aurais pu utiliser des 'Integer' qui sont en précision infini plutôt que des 'Int' qui sont en 32 bits.

```
fibs :: [Integer]
```

C'est bien le type d'une fonction. Elle prend zéro arguments et retourne une liste d'Integers. Ce n'est pas une astuce de syntaxe, 'fibs' est vraiment une fonction qui retourne la liste des nombres de Fibonacci *lorsqu'elle est évaluée*. C'est ce genre de logique qui permet au compilateur d'Haskell de faire du code très rapide, et qui permet au programmeur d'écrire du code très efficacement et rapidement.

III-C - Les types polymorphes

C'est l'heure d'une petite digression à propos des types. Comme vous avez pu le remarquer, la mode est au " tout fonction ". Et c'est vrai. Par exemple, prenez '4'. Lorsque vous codez '4' en dur dans votre code, cela n'a ni plus ni moins l'air du nombre '4'. Et pour Haskell ? Tapez '4' dans Hugs ou GHCi. Vous obtenez ce genre de truc:

```
4 :: Num a => a
```

On dirait une fonction qui prend un paramètre. Ce n'est pas le cas, tout est dans la double flèche '=>' plutôt que la simple flèche '->'. Ce type se lit " quatre est du type 'a', où 'a' est dans la classe 'Num'. " Qu'est ce que la classe Num ? C'est la classe à laquelle tous les nombres appartiennent. La vérité, c'est qu'Haskell a quelque chose que le C n'a pas : du vrai polymorphisme. La plupart des programmeurs C++ connaissent bien le terme 'surcharge', qui signifie qu'une fonction est définie avec plus d'un jeu de paramètres. Par exemple, l'addition et la multiplication sont surchargées, ce qui permet la combinaison suivante :

```
int a = 4, b = 5;
float x = 2.5, y = 7.0;

cout << a + b; //9
cout << a + y; //11
cout << y + a; //11.0
cout << x + y; //9.5
cout << b * a; //20
cout << b * x; //12.5
cout << y * b; //35.0
cout << x * y; //17.5
```

En C/C++, ceci est permis grâce à la définition des fonctions surchargées suivantes :

```
operator+ (int, int);
```


Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

```
operator+ (int, float);
operator+ (float, int);
operator+ (float, float);
operator* (int, int);
operator* (int, float);
operator* (float, int);
operator* (float, float);
```

Le compilateur sélectionne le type approprié à la compilation. Le problème, c'est qu'en C/C++, chaque combinaison doit être écrite séparément. De plus, en C/C++ n'importe quelle autre fonction qui souhaite utiliser un int ou un float doit préciser lequel il utilise, ou doit elle-même être surchargée. Cela nous amène à l'idée de classes. Pour quels types '+' est-il défini ? En C/C++ il est possible de surcharger l'opérateur pour de nouveau types, mais ces nouveaux types ne seront pas interchangeables avec int, float et les autres types numériques. Par exemple, les fonctions de tri comme mergeSort et quickSort auront besoin d'être récrites pour trier des tableaux de nouveaux types. A l'inverse, voici le type de mergeSort en Haskell :

```
mergeSort :: Ord a => [a] -> [a]
```

Qu'en est-il? Dans la signature, il y a bien sûr deux paramètres et non trois. Le premier élément qui a l'apparence d'un paramètre est en fait une restriction de classe. Comme on peut s'en douter, 'mergeSort' prend une liste d'objets d'un type donné (type 'a'), et retourne une liste d'objets du même type. Alors pourquoi le type suivant est-il insuffisant ?

```
mergeSortBadType :: [a] -> [a]
```

La réponse est qu'à un certain point dans mergeSort les éléments auront besoin d'être comparés les uns aux autres avec des opérateurs comme '>', '<', '>=', ou '<='. En Haskell ces opérateurs font partis de la définition d'une classe. L'opérateur '>' et les autres sont définis uniquement pour les membres de la classe 'Ord', nommée aussi car on peut ordonner leurs éléments. Beaucoup de types numériques font parties de la classe Ord, comme les caractères et les chaînes. Ainsi, mergeSort doit clarifier le type attendu en précisant que son argument doit être une liste d'objets qui supportent les opérateurs tel que '<'. On pourrait aussi préciser encore plus le type attendu, mais ce serait superflu.

Et le '4'? Pourquoi est-il de type 'a', où 'a' est un membre de la classe 'Num'? Il ne peut pas être juste un Num? Ou un Int? Il peut être un Int si on le précise, comme ceci:

```
a = (4 :: Int) + 2
```

Dans cette définition '4' est un Int. C'est la technique pour spécifier le type de quelque chose dans une expression. Mais sans ça, 4 est de type 'a', où 'a' est dans la classe 'Num', ou plus simplement 4 est de type 'a' dans la classe 'Num'. Et c'est ce qui est important, puisque '+' est définis pour tous les membres de la classe Num, ce qui signifie que '4' est parfaitement légal comme paramètre de cette fonction :

```
doubleIt :: Num a => a -> a
doubleIt n = n + n
```

'-' et '*' sont aussi définis pour tous les types membres de Num, donc 4 est aussi valable pour cette fonction:

```
fib :: (Num a, Num b) => a -> b
fib n = fibGen 0 1 n

fibGen :: (Num a, Num b) => b -> b -> a -> b
fibGen a b n = case n of
```


Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

```
    0 -> a
    n -> fibGen b (a + b) (n - 1)
```

C'est la fonction fib de tout à l'heure, mais les types ont changés. Le premier type se lit, " fib est du type 'a' vers 'b', où 'a' est un membre de la classe Num et 'b' est un membre de la classe Num. ". Il y a une seule flèche '=>' puisque qu'il y a toujours une seule section qui décrit les restrictions de classes. Pourquoi faire ça ? On ne devrait pas plutôt choisir un type? Eh bien, comment feriez vous si vous travailliez sur un projet de groupe, et que deux personnes aient besoins de calculer des nombres de Fibonacci ? Et que pour une raison donnée, l'un ai besoin d'un Int en retour et l'autre d'un Integer ? Ou d'un Double ? Vous écririez le code en double ? En C, vous seriez obligé. Utiliser le type le plus général possible permet la réutilisabilité. Les classes de types permettent la réutilisabilité.

Notez également que dans le premier appel à 'fibGen' le troisième paramètre est 'n', comme le premier paramètre de 'fib', et que les types de 'fib' et 'fibGen' font de même. Enfin la raison pour laquelle j'ai écrit 'fib' avec un type de retour différent est la suivante :

```
fib :: Int -> Integer
```

On a seulement besoin de la taille d'un Int pour notre compteur, mais peut-être qu'un Integer sera nécessaire pour le résultat. Remarquez aussi comment les types circulent dans 'fibGen'.

L'écriture ne mélange pas les paramètres de type 'a' et 'b', et un paramètre de type 'b' est aussi utilisé comme valeur de retour. Les types correspondent aussi bien extérieurement qu'intérieurement. Faire suivre les types comme ça sera important pour le débogage.

Continuons, dans l'exemple de fib nous avions utilisé 'tail'. Voici son type :

```
tail :: [a] -> [a]
```

En C, tail doit être réimplémenté pour chaque type de liste. Cela semble plutôt excessif ! Et en ce qui concerne l'opérateur d'index '!!' ? Eh bien dans la plupart des autres langages, indexer une liste est fait en dur par le langage lui-même, pour que cela fonctionne pour tous les types. En C, tout est soit surchargé, soit en dur, ou ne concerne qu'un seul type. Bon, il y a quelques exceptions, qui utilisent généralement des transtypages vers et depuis '(void *)'.

Naturellement, vous verrez souvent 'Num a =>' au début d'une signature de type, avec 'a' et 'b' dans la signature elle-même. Dans ce cas, 'a' et 'b' sont des variables de types, utilisées par le compilateur pour déterminer le type correcte à la compilation. Vous obtiendrez de temps en temps des messages comme 'can't determine type', ou 'type mismatch'. Le second signifie que vous avez fait une bêtise sur les types, alors que le premier signifie généralement qu'une variable de type ne peut pas être réduite à un seul type pour une de vos fonctions. Cela peut arriver pour une raison très simple :

```
main = putStrLn (show 4)
```

'putStrLn' prend une chaîne et l'affiche. 4 a un type polymorphe : il est membre d'une classe de type et n'est pas réduit à un seul type. 'show' prend basiquement tout ce qui peut être convertit en chaîne, ce qui fait qu'il ne précise pas le type de '4' non plus. Cela laisse le compilateur dans l'embarras : aucun type particulier n'est indiqué nulle part, et il s'en plaint. Pour résoudre ce problème, ajoutez une définition de type comme ceci :

```
main = putStrLn (show (4 :: Int))
```


Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

Vous pouvez remplacer Int par Integer, ou Double, or quoi que ce soit. Voila qui sera pratique pour le moment où vous aurez à tester une fonction généralisée.

Une dernière remarque, vous pouvez définir le type de plusieurs fonctions simultanément:

```
addOne, subtractOne :: Int -> Int
```

III-D - Les fonctions, enfin !

Retournons à nos fonctions. Comme vous avez pu le remarquer, apparemment tout peut servir de paramètre ou de valeur de retour pour une fonction. Et c'est absolument vrai, tant que les types correspondent. Regardons par exemple la fonction extrêmement utile 'map' :

```
map :: (a -> b) -> [a] -> [b]
```

A présent vous devriez être capable de lire cette ligne, aussi étrange qu'elle paraisse. " map est du type fonction de a vers b que suit une liste de type a et retourne une liste de type b ". map prend une fonction comme paramètre. Cette fonction n'a pas de restriction de type, elle est polymorphe. Regardez les deux autres éléments : la première fonction est de a vers b, ensuite on a une liste de type a et comme retour une liste de type b. Avec un nom comme 'map' ('associer', ndt), son utilité devrait être assez clair :

```
fooList :: [Int]
fooList = [3, 1, 5, 4]

bar :: Int -> Int
bar n = n - 2

map bar fooList
= [1, -1, 3, 2]
```

Sans commentaires. Notez seulement que j'ai dû donner à 'fooList' et 'bar' un type précis, sinon Hugs et GHC se plaignent que les types ne sont pas totalement déterminés.

Vous pouvez écrire des fonctions qui prennent d'autres fonctions en arguments. Ca peut être rigolo, et également très utile. Essayons à présent quelque chose de plus subtil :

```
subEachFromTen :: [Int] -> [Int]
subEachFromTen = map (10 -)
```

De quoi s'agit-il? Tout d'abord, les parenthèses autour du '-' et du '10' sont absolument nécessaires. Et à quoi ça sert? Une étape à la fois ! '(- 10)' est une fonction. Elle prend un nombre et retourne 10 moins ce nombre. Tapez ':t' sous Hugs ou GHCi pour vous en convaincre:

```
(10 -) :: Int -> Int
```

Ou plutôt:

```
(10 -) :: Num a => a -> a
```


Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

Ensuite, 'map' prend une fonction comme premier argument. C'est la raison pour laquelle Haskell utilise des flèches pour définir les types, plutôt qu'une liste avec des parenthèses. 'map' appliqué à '(10 -)' a le type suivant (n'hésitez pas à vérifier sous Hugs ou GHCi):

```
map (10 -) :: Num a => [a] -> [a]
```

Cette fonction prend une liste de membres de Num (qui doivent tous être du même type de Num, bien sûr) et retourne une liste du même type. C'est ce qu'on appel 'l'évaluation partielle'. Vous prenez une fonction, vous lui fournissez une partie de ses paramètres et vous obtenez une fonction du reste des paramètres. Voila donc 'subEachFromTen' en action :

```
subEachFromTen [4, 6, 7, 11]
= [6, 4, 3, -1]
```

C'est le résultat qu'on attendait. Rappelons qu'appliquer subEachFromTen à une liste, même nommée, ne change pas cette liste mais retourne seulement le résultat.

Prenez un peu de temps pour jouer avec l'évaluation partielle, les fonctions sur les listes et la compréhension de listes. Souvenez-vous qu'une fonction 'met la main' tout de suite sur ses paramètres, donc vous aurez à mettre des parenthèses autour d'un paramètre composé d'une fonction avec ses propres paramètres.

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

IV - C'est parti pour les fonctions !

IV-A - Les motifs

Vous devriez maintenant être à l'aise avec la définition et l'utilisation de fonctions en Haskell, avec au choix Hugs, GHC ou GHCi. Si ce n'est pas le cas, continuez à vous entraîner. Il est maintenant temps de nous intéresser à toutes les manières dont une fonction peut être définie.

Toutes les fonctions ont un type, même celles sans paramètres (les 'variables globales' et les 'variables locales'). On peut parfois se passer d'écrire ce type puisque le compilateur peut le déterminer, mais c'est une bonne pratique et c'est parfois une nécessitée. Après avoir lu ce chapitre, vous devriez lire le document suivant sur la syntaxe :

http://www.cs.uu.nl/~afie/haskell/tourofsyntax.html

Nous pouvons détailler quelques exemples pour clarifier les choses. Pour commencer, une petite fonction qui prend une liste et fait la somme de ses éléments :

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

Cette fonction est récursive. Elle prend une liste de 'a' et retourne un 'a'. Cependant, il semble qu'il y ait deux définitions de sumAll. Et c'est le cas. C'est comme ça que fonctionne la reconnaissance de motifs ('pattern matching' en anglais, ndt). Les paramètres des deux définitions ont chacun leur spécification, et à chaque appel de sumAll l'évaluation se fera en fonction du motif reconnu.

Regardons les définitions. La seconde est la plus claire. '[]' Représente la liste vide, et sumAll d'une liste vide est défini comme égal à zéro. La ligne du milieu est plus complexe. '(x:xs)' est donné comme paramètre, comme si nous essayons de coller quelque chose en tête d'une liste. En réalité c'est le cas, puisque ce motif prend son entrée et la sépare en deux. Il y a quelques autres motifs qui font ça, cette fonctionnalité d'Haskell rend les listes très faciles à utiliser.

Pour résumer, quand on a '(x:xs)' donné comme paramètre dans la définition d'une fonction, il ne prendra que les listes qui ont un élément en tête. En d'autres mots, il ne prendra que les listes avec au moins un élément. Le choix des noms de variables 'x' et 'xs' est totalement arbitraire, mais comme 'x' va correspondre au premier élément de la lise et 'xs' au reste, il est naturel d'écrire un 'x' pour le premier, et le reste des 'xs' pour la liste résultante.

La première définition est appelée lorsque la donnée en entrée correspond à 'sumAll (x:xs)', c'est-à-dire lorsque sumAll est appelé avec au moins une valeur. En retour, vous aurez la somme de la première valeur de votre liste avec le résultat de l'appel à sumAll sur le reste de la liste. Les motifs sont essayés du haut vers le bas, donc lorsque 'sumAll (x:xs)' ne correspond pas, on essaie 'sumAll []'. Le seul motif qui ne pourrait pas correspondre à 'sumAll (x:xs)' est la liste vide, celle-ci sera prise par 'sumAll []' qui retournera zéro. C'est la condition terminale de la récursion.

On voit très souvent ce genre de fonctions en Haskell. Les motifs nous permettent de remplacer des instructions 'switch' compliqués par des définitions distinctes, traitant des entrées distinctes. Cela augmente la clarté et la concision du code. Nous pouvons alors récrire la fonction fib avec les motifs :

```
fib :: (Num a, Num b) => a -> b
fib n = fibGen 0 1 n

fibGen :: (Num a, Num b) => b -> b -> a -> b
```


Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

```
fibGen a _ 0 = a
fibGen a b n = fibGen b (a + b) (n - 1)
```

Dans cette définition on utilise un littéral ('0') pour comme motif. Remarquez aussi le souligné bas ('_') de la première définition. Ce caractère correspond à n'importe quoi, comme un nom de variable, mais ne subira pas d'affectation comme un paramètre. Cela rend le code plus clair si on ne souhaite pas utiliser pas un paramètre. Dans notre cas, le second paramètre n'est pas utilisé, seul le premier et le dernier le sont.

IV-B - Après les motifs, les gardes

Certaines fonctions nécessitent un traitement plus complexe pour choisir entre les définitions. On peut le faire avec des gardes, comme ceci :

```
showTime :: Int -> Int -> String
showTime hours minutes
  | hours == 0 = "12" ++ ":" ++ showMin ++ " am"
  | hours <= 11 = (show hours) ++ ":" ++ showMin ++ " am"
  | hours == 12 = (show hours) ++ ":" ++ showMin ++ " pm"
  | otherwise  = (show (hours - 12)) ++ ":" ++ showMin ++ " pm"
 where
 showMin
   | minutes < 10 = "0" ++ show minutes
   | otherwise  = show minutes
```

Ne vous préoccupez pas du paragraphe commençant par le 'where' pour l'instant. 'showTime' possède une seul définition, mais elle est éclatée en 4 gardes. Chaque garde a une expression booléenne. Ils sont testés dans l'ordre. Le premier qui vaudra Vrai verra son expression évaluée et retournée. La fonction est égale à l'expression correspondant au garde évalué à Vrai. 'otherwise' étant égal à Vrai, il sera toujours accepté s'il est atteint. Cependant 'otherwise' n'est pas forcément nécessaire. '++' est l'opérateur de concaténation de listes. Il est utile ici pour la raison suivante:

```
String :: [Char]
```

Tout est donc clair, sauf pour le 'where'. Je l'ai utilisé pour définir une fonction locale, 'showMin'. 'showMin' pourrait rappeler une variable dans un langage impératif. Mais plutôt qu'utiliser un 'if' ou un 'case', j'ai préféré utiliser les gardes à nouveau pour donner deux définitions de 'showMin'.

Au total, cette fonction prend une heure (entre 0 et 23) et des minutes (entre 0 et 59) et donne l'heure sous forme de chaine.

Notez bien que les fonctions définies dans les clauses 'where', et dans leurs cousins les clauses 'let', sont visibles uniquement par le motif dans lesquelles elles sont définies. Une fonction définie à l'aide de multiples motifs ne peut pas utiliser de clause 'where' pour définir une variable qui leur soit commune.

IV-C - Le 'If'

J'ai évoqué l'instruction 'if' tout à l'heure: elle existe en Haskell mais seulement sous la forme if-then-else. Toutes les fonctions devant retourner une valeur, 'if-then' ne suffit pas. Voici un exemple :

```
showMsg :: Int -> String
showMsg n = if n < 70 then "failing" else "passing"
```


Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

C'est simple, n'est-ce pas? 'showMsg' ayant String comme type de retour, il faut que ce soit aussi le type des valeurs dans le 'if' et le 'then'. On peut aussi utiliser le 'if' dans une sous-partie de la fonction :

```
showLen :: [a] -> String
showLen lst = (show (theLen)) ++ (if theLen == 1 then " item" else " items")
 where
 theLen = length lst
```

IV-D - L' Indentation

Vous avez sûrement remarqué que j'utilise l'indentation pour séparer les blocs de code source. Ce n'est pas seulement une affaire de style, cela fait partie de la syntaxe d'Haskell. L'indentation fait partie de la structure du langage. Plus exactement, changer le niveau d'indentation entre une ligne et la suivante indique le début ou la fin d'un bloc de code.

De la même manière, Haskell ne vous laissera pas mettre la définition de deux fonctions sur des lignes consécutives sans blanc. Il vous demandera au contraire de mettre une ligne blanche pour montrer que la première définition est terminée. Tout ceci oblige à avoir dans une certaine mesure un style correct, et réduit considérablement la pollution du code avec des caractères comme '{', '}', et ';'.

IV-E - Les lambdas fonctions

Haskell permet de définir des fonctions locales, ce qui est très pratique. Mais vous pourriez vous demander " Peut-on définir une fonction de manière encore plus brève ? "

```
(\x y -> x * 4 + y) :: (Num a) => a -> a -> a
```

A quoi sert ce '(\' au début? Le caractère '\', voyez vous, ressemble à la lettre grecque lambda, qui est aussi le symbole d'Haskell lui-même :

http://www.haskell.org/

Le 'Lambda calcul' est une branche des mathématiques qui utilise des fonctions créées " à la volée ". Je n'en sais pas plus à ce sujet, mais vous trouverez beaucoup de ressources sur le net. Une 'lambda expression' démarre généralement avec '(\' et ressemble souvent à celle donnée ci-dessus. Elle commence avec '(\', donne quelques noms de variables (courts), une flèche '->' et une expression qui utilise ces variables. Et bien sûr une ')'. Alors , que fait-elle ? Elle définit une fonction qui sera utilisée " sur place ". Par exemple :

```
map (\x -> "the " ++ show x) [1, 2, 3, 4, 5]
= ["the 1", "the 2", "the 3", "the 4", "the 5"]
```

Je parle des lambda fonctions car elles sont bien pratiques, mais aussi car elles interviennent dans des exemples plus complexes. Les lambda fonctions ne peuvent évidemment pas être récursives, puisqu'elles auraient besoin d'un nom pour s'appeler elles-mêmes. Elles sont donc utiles lorsqu'on a besoin d'une fonction seulement une fois, généralement pour définir une autre fonction.

IV-F - Les types polymorphes et les constructeurs de types

Le programme le plus simple que l'on puisse faire est le suivant:

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

```
main = return ()
```

La variable 'main' est un mot réservé. Quelque soit sa valeur, le programme s'exécute. Voici le fameux et inévitable "Hello World":

```
main = putStrLn "Hello World"
```

Voici le type de 'main':

```
main :: IO ()
```

Comment dire? Ce type est plutôt étonnant. On peut même voir deux choses étonnantes dans cet exemple. Restez calmes, cela arrive souvent lorsqu'on apprend Haskell, et c'est pour ça que ce tutoriel est si long. '()' est un type. La seule valeur de type '()' est notée '()'. On peut aussi nommer ce type et sa valeur 'null'. Ce type peut donc se lire " main est du type IO null ". Mais qu'est ce que c'est que ce truc ? Pourquoi met-on deux types l'un après l'autre ? IO n'est pas un type. 'IO a' est un type. 'IO' est un constructeur de type qui prend un paramètre. Respirez profondément, les secours arrivent.

'IO' n'est pas une classe. Lorsque nous parlions des classes, j'ai dit que les fonctions pouvaient être polymorphes, ce qui signifie qu'elles peuvent utiliser des valeurs de n'importe quel type pourvue que les bonnes fonctions soient définies pour ces types. Vous pouvez créer un type et en faire un membre de la classe Num, pourvus que vous lui définissiez aussi ses opérateurs '+', '-', et/ou '*' ainsi que l'égalité. Si vous faites tout ça, toute fonction qui a '(Num a) =>' au début de son type acceptera votre nouveau type et tout ira bien. Mais 'IO' n'est pas une classe, ni même une fonction polymorphe. C'est quelque chose de pire# C'est un type polymorphe.

Ce type prend donc un autre type en paramètre. Regardons un exemple tiré de la librairie standard :

```
data Maybe a = Nothing | Just a
```

On peut le trouver ici:

http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data.Maybe.html

On définit un type avec 'data'. Les valeurs de droite sont séparées par un '|', le " tuyau ", qui peut être vu ici comme un " ou ". Ce type se lit " une valeur de type 'Maybe a' peut être 'Nothing' ou 'Just a' ". Voici un exemple utilisant Maybe et des motifs :

```
showPet :: Maybe (String, Int, String) -> String
showPet Nothing  = "none"
showPet (Just (name, age, species)) = "a " ++ species ++ " named " ++ name ++ ", aged " ++ (show
 age)
```

'showPet' a deux motifs. Le premier correspond à une valeur de type 'Nothing', le premier 'contructeur de donnés' de Maybe. Il n'y a pas de variable après le 'Nothing' exactement comme dans la définition de 'Maybe a'. Le deuxième motif correspond à une valeur de 'Just', le second 'constucteur de donnés' de Maybe. 'Just' est écrit avec un tuple, comme dans la définition de type. Les parenthèses sont utilisées pour regrouper les choses dans le motif. Les mots 'Just' et 'Nothing' sont choisis arbitrairement, quoique judicieusement. La majuscule a son importance. Comme vous avez pu le remarquer, en Haskell la première lettre des variables est une minuscule alors que celle des types est

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

une majuscule. Cela fait partie de la syntaxe d'Haskell. Les constructeurs de type ne sont pas des variables, c'est pour ça que cette convention s'applique aussi à eux.

IV-G - La Monade IO

Aller, revenons à notre 'main' et son type 'IO ()'. IO et un type polymorphe. C'est quoi ? Malheureusement, je ne peux pas vous montrer sa définition. 'IO a' fait partie du standard et son implémentation fait partie des rouages cachés d'Haskell# Cette implémentation est effectivement bas niveau et plutôt complexe. 'IO' fait partie de la classe Monad, qui en gros vous permet d'écrire du code pseudo-séquentiel en utilisant la notation 'do'. On en parlera dans une minute. 'IO' signifie bien sûr Entré/Sortie (E/S, Input/Output en anglais, ndt) et donc vous permet de lire et d'écrire des états. C'est donc 'IO' qui permet à Haskell d'interagir avec le monde réel.

```
main :: IO ()
```

Une fonction de type 'IO a' effectuera une opération d'entrée/sortie et retournera une valeur de type 'a'. Dans notre cas, main retournera une valeur de type '()', donc '()' ou 'null'. Main est donc toujours une opération d'entrée/sortie qui retourne null. Voici un exemple de main. Remarquez qu'il n'est jamais nécessaire de donner le type de main.

```
someFunc :: Int -> Int -> [Int]
someFunc ........

main = do
 putStr "prompt 1"
 a <- getLine
 putStr "prompt 2"
 b <- getLine
 putStrLn (show (someFunc (read a) (read b)))
```

Voila comment cela fonctionne: La monade IO un type polymorphe auquel on associe les deux opérations 'return' et '>>=' qui doivent patati patata# Hum... Bon. C'est l'explication qu'on trouve dans les autres tutoriaux et visiblement ça n'a pas suffit puisque vous lisez celui-ci. Laissez-moi réessayer.

Voila comment cela fonctionne: Tout ce qu'on a fait jusqu'à présent n'est pas de l'E/S. Rappelez-vous : toute 'affectation de variable' est définitive. Mais dans notre cas, si vous appelez 'getLine' plusieurs fois vous n'obtiendrez pas forcément le même résultat à chaque fois. 'getLine' n'est égal à rien, ça change tout le temps ! Une valeur comme '4' par contre restera toujours égale à 4. L'E/S pose donc problème dans un langage qui ne supporte que la " vrai égalité " dans les affectations. C'est le principal point d'achoppement dans le design des langages fonctionnels depuis l'origine. Haskell propose une solution.

La plupart des langages fonctionnels font des concessions lorsqu'il s'agit de l'E/S et laissent tomber leur pureté fonctionnelle. Mais bien sûr, Haskell propose quelque chose de plus sioux. En fait, nous allons faire appel à une obscure branche des mathématiques : les monades, qui traite des fonctions de transformation d'états. Les auteurs d'Haskell les ont utilisés pour vous permettre d'écrire des fonctions qui utilisent des états sans casser la pureté. [Ce n'était pas compliqué à expliquer, il suffisait de faire appel aux instances divines pour se débarrasser du problème ! Haskell fonctionne grâce à l'intervention du Saint Esprit. -Eric]

Pour faire court, la monade IO prend un certain état, le change grâce à une fonction, puis le passe à une autre fonction d'E/S. Dans l'exemple, les fonctions 'putStr', 'getLine', and 'putStrLn' font partie d'IO. 'Lier' ces fonctions au 'main' en utilisant la notation 'do' signifie que lorsque le 'main' sera évalué, les fonctions seront évaluées dans l'ordre, comme attendu. Le main ci-dessus écrira "prompt 1" à l'écran puis lira une chaîne, écrira "prompt 2" etc. La fonction '(read x)' prend une chaîne et retourne un nombre dans le bon type, pourvu que la chaîne représente un nombre. Le 'show' prend le résultat de 'someFunc' et retourne une chaîne que putStrLn pourra afficher. Nous y reviendrons.

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

IO ressemble beaucoup à du code impératif, et ça en est, à ceci près. Les Entrées/Sorties doivent se dérouler en séquence dans un programme au contraire des fonctions mathématiques. La plupart des fonctions Haskell sont écrites comme des fonctions mathématiques : l'ordre des opérations n'a pas d'importance. Alors enfin comment fait-on pour les E/S ? Peut-on les décrire "mathématiquement" ? Eh bien# Oui. La théorie des Monades dit qu'une fonction peut être "égale" au fait de transformer un état. La théorie des monades formalise l'idée d'une séquence d'actions, et c'est ce qui laisse aux créateurs d'Haskell la possibilité d'avoir un type 'IO a' qui se comporte comme n'importe quelle autre type.

Lorsqu'une séquence de fonctions monadiques est évaluée, les fonctions le sont dans l'ordre. Mais avant cette évaluation, Haskell a le loisir de traiter ce type 'IO a' comme n'importe quel autre type, c'est-à-dire comme des expressions en attente d'évaluation. Le type 'IO a' n'est donc pas un élément spécial d'Haskell, bien que les Entrées/Sortie soient implémentées dans la librairie standard grâce à des appels systèmes. Au total, faire des E/S vous oblige à utiliser des types polymorphes et une théorie mathématique plutôt obtuse# C'est pour ça que je n'en parle que maintenant.

Toutes les fonctions d'Haskell sont " mathématiques " (utilisent la " vrai égalité ") et de plus, sauf mention contraire, elles sont paresseuses. Ce qui signifie que le seul moyen de forcer une évaluation est d'associer une fonction 'monade IO' au 'main'. **Rien ne sera évalué s'il n'est pas dans la valeur de retour d'une fonction monadique, ou s'il devient nécessaire de calculer cette valeur de retour**. Le fait que la monade IO force l'évaluation n'est pas nécessairement important, mais il explique certains comportements bizarres que vous pourriez constater. Il y a quelques fonctions 'strict' dans Haskell qui lorsqu'elles sont évaluées, commencent par évaluer tous leurs paramètres. Ce comportement est en général signalé dans la documentation.

On peut utiliser les monades IO pour écrire des programmes dans un style impératif en Haskell. C'est un résultat du comportement de la monade IO. Mais ce serait inapproprié, car vous perdriez toute la puissance d'Haskell. Il fallait le mentionner !

Tout ce qui traite du "reste de l'ordinateur" fait partie de la monade IO : les appels aux pilotes, les librairies réseaux, les fichiers, les processus, et enfin les appels systèmes. Il y a d'autres monades dans la librairie standard d'Haskell, et vous pouvez aussi écrire les vôtres. Ce ne sera pas facile, mais pas parce que c'est compliqué : en fait le plus difficile sera de comprendre pourquoi il y a aussi peu de code à écrire# Mais nous verrons ça plus tard.

IV-H - Le 'main' à la loupe

Voici à nouveau l'exemple du main:

```
someFunc :: Int -> Int -> [Int]
someFunc .........

main = do
 putStr "prompt 1"
 a <- getLine
 putStr "prompt 2"
 b <- getLine
 putStrLn (show (someFunc (read a) (read b)))
```

Cette présentation d'Haskell se veut générale. Cette description de la monade IO et du main devrait vous suffire dans la plupart des cas. Vous pouvez l'utiliser pour la plupart des tâches simples, et pour tester vos idées dans GHC. Vous n'en aurez pas besoin pour Hugs ou GHCi : il suffit de mettre vos définitions de fonctions dans un fichier, de le charger dans Hugs ou GHCi et de lancer les fonctions.

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

J'ai dit plus tôt que l'indentation remplaçait avantageusement les caractères '{', '}', et ';'. Ce n'est pas toujours vrai : cette notation existent en Haskell et est parfois (mais rarement) préférable. La technique d'aligner les blocs avec des blancs est simple et efficace, comme le montrent les exemples.

On peut trouver la documentation de la fonction 'read' ici:

Prelude: the 'read' function

Votre code se trouve dans 'someFunc'. Sa valeur de retour est injectée dans 'show', ce qui fait que 'someFunc' peut être définie avec une grande variété de types de retour, comme 'Int', '[Int]', 'String', ou même '(String, [Int])'. Le type de 'show' se trouve ici :

Prelude: the 'show' function

'show' est une méthode de classe définie pour les membres de la classe 'Show'. Exactement comme les méthodes '+' and '*' sont définies pour les membres de 'Num'. Vous pouvez les voir ici :

Prelude, Section: the Num class

Les types des fonctions d'E/S, en particulier 'putStr', 'getLine', et 'putStrLn' sont donnés ici:

System.IO, Section: Special cases for standard input and output

Ainsi que dans le Prelude, ce qui explique qu'ils soient toujours accessibles:

Prelude, Section: Simple I/O operations

Comme vous pouvez le voir dans la documentation, lorsque vous donnez une String à la fonction putStrLn, vous vous retrouvez avec une valeur du type 'IO ()'. Le 'null' signifie qu'aucune information utile n'est retournée par la fonction. Il change l'état de la monade IO en écrivant quelque chose à l'écran, et ne retourne rien.

La flèche '<-' est utilisée pour lier le résultat d'une opération d'IO à une variable. Le type de 'getLine' est 'IO String'. La notation 'do' autorise une fonction monadique comme 'getLine' à être précédée par la fameuse flèche. La variable devient accessible à partir de ce point. Si vous ne faites pas d'affectation avec une flèche et une variable, la valeur de retour de votre fonction sera ignorée. Mais l'état de la monade est quand même affecté par la fonction ! En effet même si vous ne stockez pas le résultat de l'appel dans une variable, la fonction fera quand même la lecture, les tampons mémoire en seront affectés etc. Ce qui fait que le prochain getLine ne retournera pas la même chose.

Les valeurs de retour non affectées sont effectivement ignorées, à une exception près. Ce n'est pas un hasard si la dernière ligne de la séquence a le même type que le main. La dernière ligne de la séquence d'opération de la monade doit être du même type IO que la monade elle-même. Si ça ne correspond pas, utilisez un 'return' :

```
getName :: IO String
getName = do
 putStr "Please enter your name: "
 name <- getLine
 putStrLn "Thank you.  Please wait."
 return name
```

'putStrLn' est de type String -> IO (), alors que 'getName' est du type IO String. 'return name' est utilisé pour terminer la fonction avec le bon type, et pour retourner la bonne donnée. Sans le dernier message, on peut écrire cette fonction de manière beaucoup plus succincte :

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

```
getName :: IO String
getName = do
 putStr "Please enter your name: "
 getLine
```

Les monades, ça ressemble fort à de l'impératif. Mais une fois dans votre fonction 'someFunc', tout ceci disparaît et vous vous retrouvez avec la paresse et l'égalité qui vont bien. Mais est-ce vraiment une bonne idée que votre code fonctionnel et paresseux soit appelé par du code pseudo-impératif? A mon sens, oui. Pour certaines opérations comme les IO il est absolument nécessaire d'assurer l'ordre d'exécution, et pour le reste vous bénéficiez des puissants outils fonctionnels. Allez, si votre tête vous fait mal, faites un break ! J'en ai eu moi aussi besoin pour écrire tout ça. Le prochain et je l'espère dernier chapitre parlera de déclaration de types avancés.

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

V - Haskell et vous

V-A - Où sont passées les boucles " for "?

Vous avez pu le remarquer, il n'y a aucune boucle 'for' en Haskell. Vous pourriez en écrire en utilisant les monades IO, mais nous avons déjà dit que ce ne serait pas une bonne chose. Alors, où sont-elles ? Si vous avez déjà tout compris à Haskell, vous pouvez sauter ce paragraphe# Mais si comme moi vous avez besoin de beaucoup d'explications, lisez ceci.

Les boucles 'for' ne sont pas nécessaires. Pas seulement en Haskell, mais en général. La seule raison qui fait que vous pourriez avoir besoin d'une boucle 'for' serait si vous aviez un traitement à faire sur certain nombre d'enregistrements en mémoire. Par exemple pour stocker la bonne valeur au bon endroit. Haskell vous débarrasse de ce problème. Regardez cet exemple :

```
bar :: Int -> Int
bar = ...

foo :: [Int] -> [Int]
foo (x:xs) = bar x : foo xs
foo [] = []
```

Bon d'accord, c'est un peu trop simple. Cet exemple est équivalent à 'foo = map bar'. Voici un exemple moins simplet. Comment feriez-vous si vous aviez à implémenter une simulation de paramètres physiques, comme la gravité, et que vous aviez à calculer une nouvelle position pour chaque objet en partant de la position actuelle de tous les objets ? La fonction suivante se charge d'une partie du processus : il faut trouver, pour chaque objet, la somme de sa masse multiplié par la masse des autres objets divisé par la distance.

En C, on écrirait une paire de boucles imbriquées. La boucle extérieure lirait la masse et la position d'un objet à partir d'un tableau d'objets. La boucle interne calcul mass1 * mass2 / distance et fait la somme. Voici la solution en Haskell, nous allons commencer par écrire les types :

```
type Mass = Double
type Pos = (Double, Double, Double) --x, y, z
type Obj = (Mass, Pos)

{-
Prend une liste d'objets.
Retourne une liste des sommes des M * Mi / Di.
L'ordre est conservé.
-}
calcMassesOverDists :: [Obj] -> [Double]
```

Voilà pour les préliminaires. Le '-' indique que le reste de la ligne est un commentaire. '{' et '}' ouvrent et ferment un bloc de commentaires.

```
calcMassesOverDists objs = calcHelper objs objs

distXYZ :: Pos -> Pos -> Double
distXYZ (x1, y1, z1) (x2, y2, z2) = sqrt (xd * xd  + yd * yd + zd * zd)
 where
 (xd, yd, zd) = (x1 - x2, y1 - y2, z1 - z2)

calcHelper :: [Obj] -> [Obj] -> [Double]
calcHelper (obj:objs) objList = (sum (calcMMoD obj objList)) : calcHelper objs objList
```


Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

```
calcHelper [] _ = []

calcMMoD :: Obj -> [Obj] -> [Double]
calcMMoD obj@(m1, pos1) ((m2, pos2):rest) =  safeValue : calcMMoD obj rest
 where
 dist = distXYZ pos1 pos2
 safeValue = if pos1 == pos2 then 0 else m1 * m2 / dist
calcMMoD _ [] = []
```

D'accord, il y a un peu de nouveauté dans cet exemple: le 'obj@' devant le '(m1, pos1)'. Le '@' se lit 'comme' : cela signifie que 'obj' se référera aux valeurs de type 'Obj' tandis que '(m1, pos1)' fera de la reconnaissance de motifs dans les valeurs de 'obj'. C'est bien pratique, car cela m'évite de réécrire '(m1, pos1)' lors de l'appel récursif de 'calcMMoD'. C'est également plus clair.

Pour 'distXYZ', j'ai mis le sous-calcul de (xd, yd, zd) sur une seule ligne pour plus de clarté. Il m'a suffit de définir le tuple '(x1 - x2, y1 - y2, z1 - z2)' puis je le fait correspondre au motif '(xd, yd, zd)', définissant ainsi 'xd', 'yd', et 'zd' du même coup. Remarquez finalement que dans calcMMoD, dist n'est pas évalué si pos1 == pos2, ce qui nous évite une division par zéro.

J'ai pu comparer pos1 et pos2 car un tuple qui contient des types de la classe 'Eq' est aussi dans la classe 'Eq'. La définition permettant ceci se trouve ici, vous aurez cependant à descendre de quatre pages pour la retrouver :

Prelude, Section: the Eq class

Cherchez la ligne suivante:

```
(Eq a, Eq b) => Eq (a, b)
```

Cette définition est donnée dans la section 'Instances' de la classe Eq. Cela signifie que quelque part dans le code source du Prelude, il y a une instance de Eq définie pour '(a, b)' avec la condition que a et b soient aussi membres de Eq. Cette définition est plutôt simple, mais il fallait la donner.

'sqrt' (racine carré, square root en anglais, ndt) est une fonction de classe, qui est définie pour tous les types dans la classe 'Floating'. Cette classe inclus bien sûr 'Double'. Voici son type :

Prelude, Section: the Floating class

'sum' est une fonction polymorphe qui est définie (une seule fois) pour tous les types de la classe 'Num', qui inclut bien sûr 'Double'. Le type de 'sum' se trouve ici :

Prelude, Section: Special Folds

J'aurais pu faire en sorte que calcMMoD retourne directement la somme, mais le code compilé est plus rapide si les tâches sont bien séparées. En effet le 'sum' du prelude est basé sur la récursion terminale et est très fortement optimisé. Plus d'infos ici :

http://fr.wikipedia.org/wiki/Récursion_terminale

Alors où se trouvent les boucles 'for' ? Chaque fonction récursive fait des itérations sur une liste, les deux combinées fonctionnent comme une paire de 'for' imbriquées. Ce code n'est pas mal, mais on peut écrire des versions de 'calcMassesOverDists' et 'calcMMoD' plus courtes avec les mêmes types. Ainsi, on peut utiliser une sous-fonction beaucoup plus simple pour 'calcMMod':

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

```
calcMassesOverDists :: [Obj] -> [Double]
calcMassesOverDists objList = map (\obj -> sum (calcMMod obj objList)) objList

calcMMoD :: Obj -> [Obj] -> [Double]
calcMMoD obj objList = map (mMoDHelper obj) objList

mMoDHelper :: Obj -> Obj -> Double
mMoDHelper (m1, pos1) (m2, pos2) = if pos1 == pos2 then 0 else m1 * m2 / distXYZ pos1 pos2

distXYZ :: Pos -> Pos -> Double
distXYZ (x1, y1, z1) (x2, y2, z2) = sqrtDouble (xd * xd  + yd * yd + zd * zd)
 where
 (xd, yd, zd) = (x1 - x2, y1 - y2, z1 - z2)
```

Je fais passer 'mMoDHelper' à la trappe en utilisant une lambda fonction :

```
calcMassesOverDists :: [Obj] -> [Double]
calcMassesOverDists objList = map (\obj -> sum (calcMMod obj objList)) objList

calcMMoD :: Obj -> [Obj] -> [Double]
calcMMoD (m1, pos1) objList = map (\(m2, pos2) ->
 if pos1 == pos2 then 0 else m1 * m2 / distXYZ pos1 pos2) objList

distXYZ :: Pos -> Pos -> Double
distXYZ (x1, y1, z1) (x2, y2, z2) = sqrtDouble (xd * xd  + yd * yd + zd * zd)
 where
 (xd, yd, zd) = (x1 - x2, y1 - y2, z1 - z2)
```

Je peux aussi me débrouiller pour éviter d'écrire calcMMoD, mais ça en devient ridicule :

```
calcMassesOverDists :: [Obj] -> [Double]
calcMassesOverDists objList = map
 (\obj1@(m1, pos1) -> sum (map (\(m2, pos2) ->
  if pos1 == pos2 then 0 else m1 * m2 / distXYZ pos1 pos2) objList) )
 objList

distXYZ :: Pos -> Pos -> Double
distXYZ (x1, y1, z1) (x2, y2, z2) = sqrtDouble (xd * xd  + yd * yd + zd * zd)
 where
 (xd, yd, zd) = (x1 - x2, y1 - y2, z1 - z2)
```

Dans tous les cas, la division par zéro est évitée grâce au test pos1 == pos2. Notez que toutes les fonctions et toutes les variables commencent par une minuscule, comme 'mMoDHelper'.

Dans l'exemple, mMoDHelper est partiellement évaluée. On lui donne un seul de ses deux paramètres. Le type de 'mMoDHelper obj' est:

```
mMoDHelper obj :: Obj -> Double
```

Qui à son tour est correct comme premier argument de 'map'.

Transformer une boucle 'for' en 'map' n'est pas toujours la bonne solution. Dans calcMMoD, c'est direct : la fonction 'sum' prend la liste générée par 'map' et en fait la somme. Mais ces fonctions simples ne correspondent pas toujours à ce que vous voulez faire. Pour les choses plus complexes, il y a des fonctions " avancées " comme 'foldr' et ses variantes.

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

Apprendre comment fonctionnent 'foldr' et 'foldl' est en quelque sorte un rite de passage pour l'apprenti Haskelleur. Vous y viendrez au fur et à mesure, par exemple en étudiant leurs définitions dans le Prélude ainsi que la définition d'autres fonctions qui utilisent 'foldr' et 'foldl' dans leur définition comme 'concat', 'or', et 'sum'. La fonction map n'était que la mise en bouche!

Au final, le plus difficile est sans doute d'arriver à quitter ses habitudes de programmeur séquentiel. Rappelez vous également que comme foldr et map sont utilisés partout, GHC les a fortement optimisé. C'est donc une bonne idée de les utiliser le plus souvent possible.

V-B - Types avancés

Ce chapitre va je l'espère vous démontrer encore un peu plus la flexibilité du système de types d'Haskell. Vous devez être familier avec le concept d'Arbre : une structure de données qui possède des n#uds qui contiennent des valeurs, et qui peuvent aussi pointer vers d'autres n#uds de la structure. L'arbre binaire en est une des formes les plus communes, c'est un arbre dont chaque n#ud a au plus deux enfants. Notez qu'un arbre n'est pas un graphe : un arbre ne boucle pas sur lui-même, c'est à sens unique. Voici un petit exemple :

```
data Tree a = Null | Node a (Tree a) (Tree a)
```

Ce type est plutôt compliqué. C'est un type 'de données', défini avec le mot clef 'data', un peu comme 'Maybe a' l'était. On peut aussi dire qu'il est polymorphe comme 'Maybe a' : 'Tree a' prend un type en paramètre. Il a deux constructeurs de données, 'Null' et 'Node'. 'Null' n'a pas d'arguments et 'Node' en a trois. Le nom indique que c'est un arbre. Après enquête il s'agit d'un arbre binaire puisqu'il y a deux enfants dans 'Node'. Alors comment ça marche ? Commençons par écrire une valeur de ce type :

```
t :: Tree Int
t = Node 3 (Node 2 Null Null) (Node 5 (Node 4 Null Null) Null)
```

Si nous devions représenter cet arbre, sans inclure les branches nulles, ça ressemblerais à ça:

```
    3
   / \
  2   5
     /
    4
```

Le premier noeud a deux enfants, et le fils de droite a un enfant, qui est le gauche. Faisons un tour du coté des constructeurs. Dans l'exemple, 'Tree a' est un type, 'Node' et 'Null' sont des constructeurs de données pour ce type. Un constructeur de données, ou plus simplement constructeur, agit comme une fonction qui regroupe des objets pour former un objet du bon type de données. On les met en #uvre uniquement pour les types définis à l'aide de 'data' ou 'newtype'. C'est un constructeur différent des constructeurs de type comme IO et Maybe. Les constructeurs de données fonctionnent comme des fonctions, et n'appartiennent pas à la signature de type. A l'inverse, les constructeurs de types fonctionnent comme des fonctions sur les types, et appartiennent **seulement** à la signature de type. Les deux ne sont pas dans le même espace de noms, donc bien souvent un type aura un constructeur de données du même nom.

Les constructeurs sont aussi utilisés pour 'déconstruire' des objets d'un type de donnée, comme ceci:

```
inOrderList :: Tree a -> [a]
inOrderList Null = []
```


Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

```
inOrderList (Node item left right) =
 inOrderList left ++ [item] ++ inOrderList right
```

'inOrderList' utilise la reconnaissance de motifs pour déterminer quels paramètres utiliser. De plus il 'déconstruit' la valeur qu'utilise le constructeur 'Node' et associe les valeurs aux variables 'item', 'left' et 'right', qui sont respectivement des types 'a', 'Tree a' et 'Tree a'. Nous connaissons ces types car la définition de Node est "Node a (Tree a) (Tree a)", et 'a' n'est pas explicité. Si vous n'êtes pas familier avec les arbres, le 't' ci-dessus est un arbre de recherche binaire et une évaluation de 'inOrderList t' donnerait le résultat suivant :

```
inOrderList t
= [2, 3, 4, 5]
```

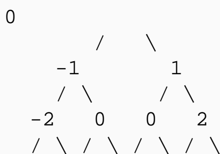
Les valeurs sont dans l'ordre ascendant, c'est dans la définition des arbres de recherche binaire. Lisez ce qui suit si vous n'êtes pas déjà familier avec eux.

Il y a un truc rigolo dans la définition de 'Tree a' : elle est récursive. Elle utilise 'Tree a' pour définir chaque enfants. On peut faire ceci en Haskell comme en C, mais il y a bien sûr une petite différence, et bien sûr on va utiliser l'évaluation paresseuse. En C/C++, pour utiliser un type dans sa propre définition, vous devez déclarer un pointeur sur un objet du même type, vous ne pouvez pas l'utiliser directement. En Haskell, on peut inclure le type directement, comme dans l'exemple de l'arbre. Que pensez-vous de cette définition :

```
foo :: Int -> Tree Int
foo n = Node n (foo (n - 1)) (foo (n + 1))

t2 = foo 0
```

Quelle est la valeur de 't2'? Et son type? La valeur de 't2' nécessite une quantité d'espace infini. On peut dessiner les premiers niveaux de l'arbre comme ça :



```
0
       /   \
    -1       1
    / \     / \
  -2   0   0   2
  / \ / \ / \ / \
```

Et caetera. Le type de 't2' est simple, et facile à déduire du type de 'foo':

```
t2 :: Tree Int
```

Si jamais vous obteniez un message d'erreur suivant : " (#) unification would result in infinite type " (" l'unification pourrait donner un type infini "), cela signifie que le type nécessiterait une quantité d'espace infini. Cela arrive typiquement avec la reconnaissance de motifs sur les listes. Dans ce cas une vérification attentive montrera une erreur dans le code aboutissant à une liste imbriquée de profondeur infinie. Le type ressemblera alors à ceci :

```
[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[...
```

Revenons à 't2'. Son type n'est pas infini, même s'il est défini récursivement. Cependant sa valeur, si on l'évalue, nécessitera une quantité de mémoire infinie. Son type est définie de manière récursive : Une valeur de type 'Tree a' peut contenir une autre valeur de type 'Tree a'. Le fait que les valeurs de type 'Tree a' utilisant le constructeur 'Node'

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

n'apparaissent pas dans le type de t2 assure que le type est finit. Ce qui fait qu'Haskell peut l'utiliser. De même, de multiple valeurs de type 'Tree Int' peuvent exister : certaines peuvent être finies, d'autres non, et toutes auront le même type. Okay ?

V-C - Pour conclure

Tout ce qui précède nous amène à deux points essentiels sur les types. Le premier, Haskell utilise l'évaluation paresseuse. Par exemple, le type suivant est valide et Haskell n'y trouvera rien à redire :

```
data Forever a = AThing a (Forever a)
```

C'est un type polymorphe, avec un unique constructeur 'AThing', et il est récursif. Les valeurs de ce type seront toujours infinies si on cherche à les évaluer en entier. Mais ça ne pose pas de problème et vous pourriez utiliser ceci dans un programme si vous le vouliez. Haskell n'essayera pas d'évaluer un objet de ce type en entier, à moins que vous ne le lui demandiez.

Le deuxième point essentiel est que pour apprendre et utiliser Haskell il faut préalablement désapprendre et remettre en question les hypothèses de base que la plupart des développeurs se fabriquent à propos de la programmation. A l'inverse de la plupart des langages, Haskell n'est pas seulement 'un peu' différent, et il nécessitera un certain investissement de la part du débutant. Ce tutoriel vous facilitera un peu la tâche, je l'espère.

Si vous jouez un peu avec Haskell, essayez d'écrire plus que des petits programmes de tests. Les problèmes trop simples ne sauront pas tirer parti de la puissance d'Haskell. Comme Omo Micro, il développera toute sa puissance sur les tâches difficiles (ndt hum#). Quand vous serez à l'aise en Haskell, attaquez l'écriture de programmes que en temps normal vous trouveriez trop ardus. Par exemple, tentez d'implémenter un algorithme de parcours de graphe et le typage d'un arbre équilibré. Au début, vous serez sans doute tenté d'utiliser votre mode de pensée impératif. Mais au fur et à mesure, vous constaterez que l'outillage et les méthodes d'Haskell vous permettent de simplifier drastiquement votre code.

Une super astuce: "N'utilisez pas de tableaux". La plupart du temps, on a besoin d'accéder aux membres un par un, dans l'ordre. Dans ce cas, vous n'avez pas besoin d'un tableau, et vous perdriez beaucoup de temps à en mettre un sur pied. Utilisez une liste. Si vous avez vraiment besoin d'accéder aléatoirement à vos données, n'utilisez pas de tableau non plus. Utilisez une liste et '!!'. Si vous avez besoin de vitesse, alors là utilisez un tableau. Commencez par les Arrays, puis passez aux IOArrays. Mais faites le uniquement si vous avez vraiment besoin de vitesse. Votre projet scolaire ne nécessite sûrement aucun tableau à moins que vous ne soyez en train d'écrire un jeu. Dans ce cas vous pourriez avoir besoin d'un accès aléatoire rapide. Enfin# Peut-être.

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

VI - Le mot de la fin

VI-A - La 'Transparence référentielle' est-elle vraiment utile ?

Les langages fonctionnels ont souvent été accusés de générer du code lent. Ce n'est pas le cas d'Haskell.

La "transparence référentielle" donne à Haskell deux grosses améliorations : une exécution plus rapide et moins gourmande en mémoire, ainsi qu'un code plus concis et plus compréhensible.

Tout d'abord, étudions les causes de cette vitesse d'exécution supérieure, et de ce gain en consommation mémoire. Il n'existe pas de code s'exécutant plus rapidement qu'un code évalué durant la compilation, à l'exception des optimisations uniquement possibles durant l'exécution. La façon la plus simple d'évaluer durant la compilation est de trouver où l'on doit modifier la valeur d'une variable, et où cette variable modifiée sera lue ultérieurement, et de n'effectuer ce changement qu'à ce moment. Cela permet d'économiser une écriture en mémoire, certainement aussi une lecture, et potentiellement des allocations mémoire.

Aussi bien en C qu'en Haskell, les fonctions hérite de l'espace de nommage de leur portée. En C, cela signifie qu'on peut accéder à n'importe quelle variable dans la portée à n'importe quel moment.

Pire, il est possible que vous ayez assigné un pointeur avec la valeur d'un espace mémoire d'une variable local, que vous soyez dans une application multi-threadée, ou que vous fassiez de l'arithmétique sur les pointeurs!

Au total, votre compilateur doit abandonner la majorité des espoirs de connaissance de la manière dont vont être mises à jour les variables au cours de l'exécution.

Comme il est possible d'écrire du code qui utilise ces techniques de mise à jour inattendue des variables, le compilateur C doit vraiment lire une variable à chaque fois que cela est écrit à moins qu'il puisse être prouvé qu'aucune mise à jour ne touchera cette variable entre deux points du programme. Chaque appel de fonction pourra potentiellement modifier toutes les variables dans la portée. Toutes les données passées en argument à une fonction d'un autre module à travers un pointeur non déclaré const doivent être relues après l'appel de la fonction. Cela arrive assez souvent dans des projets de grande taille.

Qu'en est-il d'Haskell ? Haskell possède une évaluation paresseuse et possède un système de références totalement transparent. Si vous passez une valeur à une fonction, elle ne sera pas mise à jour, ce qui implique que GHC peut supposer cela. En particulier, il n'y a jamais besoin de copier une valeur lorsqu'on la passe en argument à une fonction, ou de la relire. Ainsi une fonction prenant en argument une liste et renvoyant une partie de cette liste ne copiera rien. Comme aucune mise à jour n'est possible, il n'y aucun danger de référencer la version originale. Cela permet de faire des grosses économies en mémoire, ce qui se traduit par moins de lecture en mémoire et moins de fautes de pages.

Autre élément important, étant donné que l'ordre d'évaluation n'a aucune importance pour le programmeur, le compilateur peut déterminer l'ordre le plus rapide et réordonnancer le code.

D'autres points rendent Haskell incroyablement rapide. Tout d'abord, l'évaluation paresseuse d'Haskell ajoute un petit plus à la vitesse. Ensuite, il n'y a que peu de composants élémentaires en Haskell : les fonctions, le if-then-else, les types polymorphes, les exceptions, les entrées/sorties et le garbage collector. La plupart des fonctions des librairies étant construites à partir de foldr ou d'un variant, elles ne font aucun travail réel. Cela représente un très petit nombre de morceaux élémentaires, et rend donc Haskell facile à compiler et à optimiser.

Si vous avez besoin de manipuler la mémoire, C est le langage que vous utiliserez. Python rend l'écriture du code plus rapide. Lisp facilite la manipulation logique. Java rend votre code portable. Selon moi, Haskell est excellent pour la plupart des objectifs industriels : de gros projets avec de nombreuses parties.

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

VI-B - Contacts et références

Contacter l'auteur : **etherson@yahoo.com**

Traduction française : Corentin Dupont **haskell.cdupont@spamgourmet.com**

Ressources sur Haskell:

http://www.haskell.org/

Le "Gentle Introduction to Haskell", la référence:

http://www.haskell.org/tutorial/

Version française par Nicolas Vallée:

http://gorgonite.developpez.com/livres/traductions/haskell/gentle-haskell/

Le "Tour of the Haskell Syntax", une autre bonne référence:

http://www.cs.uu.nl/~afie/haskell/tourofsyntax.html

GHC's Hierarchical libraries, une référence excellente à condition d'avoir quelques bases:

http://www.haskell.org/ghc/docs/latest/html/libraries/index.html

Compilateurs:

http://www.haskell.org/hugs/

http://www.haskell.org/ghc/

VI-C - Digression finale

Tout au long de ce tutoriel, je fais référence à la 'puissance' d'Haskell. Ca me fait penser à une vieille équation sur les langages de programmation:

```
flexibilité * contrôle du détail = constante
```

On pourrait récrire cette affirmation comme ceci:

```
flexibilité * contrôle du détail = puissance
```

Haskell possède à la fois plus de flexibilité et plus de contrôle que la plupart des langages. Je ne connais rien qui batte le contrôle du C, mais Haskell peut faire tout ce que le C fait à moins que vous n'ayez besoin de faire des lectures et écritures octet par octet en mémoire. C'est pour ça qu'Haskell est " puissant ".

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

J'ai écrit ce tutoriel car j'ai eu beaucoup de mal à apprendre l'Haskell, mais maintenant je l'adore. Je n'ai pas trouvé de tutoriel qui prennent vraiment les étudiants en informatique par la main face aux difficultés d'Haskell. Il m'a fallut deux semestres à l'Université pour vraiment le maîtriser, et encore heureusement qu'un de mes amis m'a aidé.

Je pensais que quelqu'un, par exemple une personne de haskell.org, écrirait un tutoriel destiné au programmeur C, mais finalement personne ne l'a fait. Maintenant je comprends mieux pourquoi. Apparemment Haskell est maintenu et amélioré par des universitaires. Ceux-ci viennent le plus souvent de langages fonctionnels comme LISP et n'ont pas besoin de faire le grand " saut ".

Pour qu'Haskell puisse sortir de l'ombre, il faut que des étudiants comme nous l'étions puisse mettre le grappin dessus et l'étudier plus facilement. Haskell est si différent des standards de l'industrie (C/C++, Ada, Java#) qu'il ne suffit pas de simplement lire du code pour le comprendre. C'est pour ça que cette passerelle était nécessaire. Il faut que des étudiants l'utilisent, pas seulement des profs d'université. Peut-être est-ce qu'ensuite des entreprises s'y intéresseront. En particulier avec le portage d'OpenGL et d'OpenAL en cours.

Les auteurs d'Haskell nous ont offert un merveilleux langage, espérons qu'il aura la place qu'il mérite dans l'industrie.

Eric Etheridge

etherson@yahoo.com

Traduction française et adaptation:

Corentin Dupont

Email : Corentin point dupont chez gmail point com

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/

Permission is granted to copy and distribute under the terms of the Creative Commons licence, Version 2.5 or any later version published by the Creative Commons Corporation; with Attribution, No Commercial Use and No Derivs. Read the full license here : http://creativecommons.org/licenses/by-sa/2.5/