```
and $ map (/=4) [1,2,3]    -- True
or $ map (<4) [1,2,3,4]    -- True
all $ map (/=4) [1,2,3]    -- True
any $ map (==4) [1,2,3]    -- False
```

Et d'autres fonctions utiles :

```
splitAt 2 [1,2,3,4]    -- ([1,2],[3,4])
splitAt 2 "cool"       -- ("co","ol")
takeWhile (<3) [1,2,3,4] -- [1,2]
dropWhile (<3) [1,2,3,4] -- [3,4]
break (==3) [1,2,3,4] -- ([1,2],[3,4])
span (/=3) [1,2,3,4]   -- ([1,2],[3,4])
inits "lol" -- ["","l","lo","lol"]
tails "lol" -- ["lol","ol","l",""]
```

Nous avons aussi la calculatrice polonaise :

```
solveRPN "2 3 +" -- 6
```

Modules

Un module contient des fonctions, des types.
Pour importer *Data.List*, deux méthodes :

```
import Data.List    -- Dans fichier
:m + Data.List      -- Dans interpréteur
```

De même, on peut restreindre l'import (*hiding* = mais pas)

```
import Module.X (fonc1, fonc2)
import Module.X hiding (fonc1, fonc2)
```

Maybe

Maybe fait office d'exception :

```
let f a b = if b == 0 then Nothing
            else Just (div a b)
fmap (\x -> x + 4) (f 4 2)
Just 6
```

Functor applicatif

Il faut importer **import Control.Applicative**,
Type du *functor applicatif* :

```
(<*>):: Applicative f=>f(a -> b)-> f a -> f b
```

Exemples :

```
[\x -> x+1, \ x -> x*2] <*> [2 ,4]
    -- [3, 5, 4, 8]
[ ( * ) ] <*> [1, 2, 3] <*> [4, 5]
    -- [4, 5, 8, 10, 12, 15]
```

Et pour le type *Maybe* :

```
Just ( * 3 ) <*> Just 4 -- Just 12
```

Les monades

Les monades servent à faire des calculs tout en se protégeant des *null*, ces valeurs valent alors *Nothing*.

```
: t (>>=)
(>>=) :: Monad m => m a->( a -> m b )->m b
```

Exemple :

```
[ 2 .. 4] >>= ( \ x -> [ 0 .. x ] )
--[0 ,1 ,2 ,0 ,1 ,2 ,3 ,0 ,1 ,2 ,3 ,4 ]
Nothing >> Just 4 -- Nothing
Just 3 >> Nothing -- Nothing
Just 3 >> Just 4  -- Just 4
Just 4 >> Just 3  -- Just 3
```

Les monoïdes

C'est un élément munie d'une loi de composition interne ainsi que d'un élément neutre :

```
import Data.Monoid
Just(Sum 2) `mappend` Just (Sum 3)
-- Just (Sum {getSum = 5})
Just(Sum 2) `mappend` Nothing
-- Just (Sum {getSum = 2})
```

Une LCI prend en compte que les éléments sous symétriques, commutatifs et associatifs.

```
Just 2 >>= (\x -> guard(x > 1)) >> Just 3
        -- Just 3
Just 0 >>= (\x -> guard(x > 1)) >> Just 3
        -- Nothing
```

Maybe a est un monoïde si *a* est un monoïde.

#Jaizappé ... ... le Haskell

Histoire

Haskell est un langage de programmation fonctionnel créé en 1990. Il est sert au λ-calcul et au calcul combinatoire. Il s'éxécute avec **ghci** et des fichiers en *.hs*.

Bases

Commentaires

Un commentaire est un message caché à l'exécution.

```
-- commentaire sur 1 lignes
{- commentaire sur 1 ou + -}
```

Arithmétique

On a les opérateurs : $+ - \times \backslash$:

```
ghci> 5 * 4 - (3 + 2) / 1
15.0
```

```
succ 5 -- incrémente
pred 5 -- décrémente
```

```
max 5 4 -- retourne 5
min 5 4 -- retourne 4
```

Booléen

On a les opérateurs : *True False not && ||* :

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

Et pour les nombres et chaines : == /= :

```
5 /= 4          -- True
5 == 5          -- True
"lol" == "xD"   -- False
```

Les chaines sont entourés par des doubles quotes !

Charger un fichier

Un fichier permet d'omettre les **let** :

```
:l monFichier     -- Charge le fichier
:reload           -- Recharge le fichier
```

Fonctions

Appel d'une fonction

```
bar 3 "lol" 5.34
bar (3, "lol", 5.34)
```

Définition d'une fonction

```
carre x = x * x -- fait le carré
func x = if(x > 5) then x - 5 else x + 5
{- Si param x > 5, retourne x - 5
   Sinon retourne x + 5              -}

somme x y z = x + y + z
somme 1 2 3 -- 6
```

Fonction avec valeurs prédéfinies

```
fonc 1 = "Unité"
fonc 10 = "Dizaine"
fonc x = "N/A"
```

Fonction récursive

```
fonc 0 = 0
fonc x = x + fonc(x - 1)
```

Fonction λ (anonyme)

Précédé par \ et sans nom :

```
carre ((\a -> a + 2) 3) -- 25
```

Ici, à usage unique, on augmente de 2 le paramètre.

```
(\a b c -> ....) -- usage avec param > 1
```

Fonction $ (joker)

```
map ($5) [(carre), (*5)] -- [25,25]
```

Ou sert à calculer les paramètres :

```
length $ [1,2] ++ [3,4] -- 4
```

Composition de fonction

L'exemple fait le carré puis la racine :

```
map (sqrt . carre) [5,3,9]
[5.0,3.0,9.0]
```

Types

Base

Pour récupérer le type de quelques choses :

```
:t x
```

Définition de type

```
data NewType = value|value
data Coul = RGB Int Int Int deriving Show
:t RGB -- RGB :: Int -> Int -> Int -> Coul
data Arbre a = ArbreVide |
  Arbre {noeud::a, fg::Arbre a, fd::Arbre a}
  deriving Show
:t Arbre
--Arbre::a -> Arbre a -> Arbre a -> Arbre a
```

Le dernier cas nous donne les accesseurs directement !
Sinon, on peut le faire manuellement :

```
composanteRouge (RGB r _ _) = r
```

Classe de type

```
class MonEgal a where
        egal :: a -> a -> Bool
        diff :: a -> a -> Bool
        diff x y = not(egal x y)
        egal x y = not(diff x y)
data MonType = V1|V2
instance MonEgal Coul where
        egal V1 V1 = True
        egal V2 V2 = True
        egal _ _ = True
instance Show MonType where
        show V1 = "valeur 1"
        show V2 = "valeur 2"
```

Variables et listes

Bases

```
let a = 3 -- met 3 dans a
let b = [3, 5, 6] -- met une liste dans b
let c = [] -- liste vide
```

On peut mettre une liste dans une liste :

```
['l'] ++ ['o', 'l'] -- == "l" ++ "ol"
[4, 3] ++ [2, 1]    -- == [4, 3, 2, 1]
5:[4, 3, 2, 1]      -- optimiser
```

De même une chaine est une liste :

```
"lol" -- équivaut ['l', 'o', 'l']
'l':"ol" -- pour 1 élément (optimiser)
```

Opérations

L'opérateur **!!** sert à prendre l'élément en X :

```
[4, 3, 2, 1] !! 2 -- retourne 2
```

Opérations : **< <= => > == /=**,
Suit l'ordre des cases :

```
[3, 2, 1] > [4, 2, 1] -- False
[3, 2, 1] > [3, 2]    -- True
```

Sélections précises :

```
head [9, 7, 5, 3] -- 9
tail [9, 7, 5, 3] -- [7, 5, 3]
init [9, 7, 5, 3] -- [9, 7, 5]
last [9, 7, 5, 3] -- 3
```

Fonctions sur les listes :

```
null [] -- True (car pas d'élément)
drop 2 [9, 7, 5, 3]  -- [5, 3]
elem 1 [5, 9, 3, 7]  -- False (non trouvé)
zip [1, 4] "hi"      -- [('h',1),('i',4)]
```

Il y a aussi *maximum*, *minimum*, *sum*, *product*, *length* et *reverse*.
Faire des listes préfabriquées :

```
[1..31]       -- De 1 à 31, idem avec lettre
[0..]         -- 0 à infini
[2,4..9]      -- [2,4,6,8]
cycle [1,2]   -- [1,2,1,2,...]
take 5 [1,2]  -- [1,2,1,2,1]
take 2 [9, 7, 5, 3]  -- [9, 7]
replicate 2 4 -- [4,4]
```

On peut même faire des ensembles :

```
[x+3 | x <- [1..4]] -- [4,5,6,7]
```

Même avec du code :

```
[if x == 5 then "o" else "L"|x<-[3..7],odd x]
-- ["L", "o", "L"]
```

On peut faire **import Data.List** dans le prélude :

```
intersperse 4 [5, 7, 2]-- [5, 4, 7, 4, 2]
--intercalate : idem mais avec listes
transpose [[1,2],[3,4]] -- [[1,3],[2,4]]
concat [[1,2],[3,4]]    -- [1,2,3,4]
```

Mots clés utiles :