Le langage **Haskell** pour les journalistes et autres amateurs de simplifications abusives

Jean-Luc Ancey

Parinux

Janvier 2013

Commentaire audio diapo 2, 0 : 00 : 00

Ce diaporama est voué à être diffusé
*avec l'**enregistrement audio** de la conférence*
pour laquelle il a été élaboré.
Chacun des écrans que vous êtes en train de lire
*fait mention dans son en-tête du **chronométrage***
lui correspondant dans l'enregistrement.

In memoriam

Aaron Swartz
(8 novembre 1986, 11 janvier 2013)

Précautions oratoires

Des oreilles amies (CC-BY-SA) vous écoutent.

Précautions oratoires

Quelques incompréhensions entre Haskell et LaTeX
quant à l'emploi des caractères spéciaux m'ont forcé
à terminer ce diaporama en catastrophe.
Si vous y voyez des erreurs, merci de les signaler !
Le backslash, que LaTeX comme Haskell
utilisent fréquemment et de façon incompatible, sera faute de
mieux noté † sur quelques rares écrans.

Cette conférence sera pleine d'exemples absurdes, caricaturaux et au style épouvantable. Ces horreurs, principalement dues à mon incompétence, cherchent quand même aussi à mettre en lumière des différences de philosophie informatique. **En aucun cas elles ne visent à dénigrer les langages impératifs.**

A l'usage des non-informaticiens qui pourraient être là par curiosité, je dirai beaucoup de choses sans montrer de code. Dans le peu que je montrerai, je préférerai toujours la notation moche dont l'emploi peut être systématique à la notation élégante qu'on ne pourrait employer qu'occasionnellement.

Illusoire de prétendre expliquer tout un langage en quelques heures... surtout quand on n'a pas tout compris soi-même.
Mais on peut essayer de donner envie.

Un journaliste se considère utile s'il permet aux gens de comprendre en une heure ce qu'il a mis beaucoup plus longtemps à comprendre lui-même. Mais assurément, **je n'ai pas tout compris**, et en particulier, pas l'élégance.

Sans être informaticien, j'ai programmé en Basic, en JavaScript, en Fortran, en C, en C++, en Perl, en Tcl/Tk, en Python, en Java, en PHP... Ce qui signifie que pour le meilleur et pour le pire, **j'ai le cerveau d'un programmeur impératif**.

Quand on procède à l'adaptation de routines que l'on connaît bien, on est bien forcé de se rendre compte à quel point Haskell est différent d'autres langages.

Merci d’être là

Thompson-Wheeler

L’annonce de la conférence a suscité l’intérêt de gens bien plus compétents que moi (notamment un partisan d’OCaml, Fabrice Le Fessant, qui a tenu un discours très argumenté).

Je signale en passant qu’il y avait ce soir même un *meetup* de Haskell-Paris. A l’usage des linuxiens, je signale l’existence du concept de HUG.

Par ailleurs, j’ai aussi parlé avec des développeurs expérimentés qui confessaient avoir vainement tenté de se mettre à Haskell. J’ai souffert comme eux, mais je crois avoir compris 1) pourquoi on ne comprend pas ; 2) pourquoi ce n’est vraiment pas incurable.

Le plus intrigant : pas de variables temporaires ; pas de compteurs ; pas de boucles *for* ni *while*.

Un programme Haskell (purement fonctionnel) se construit presque intégralement sur la notion d’égalité *au sens mathématique du terme*. Ce n’est presque jamais, pas même dans le détail, un inventaire chronologique d’actions à accomplir.

Pourquoi je m’y suis mis

Pourquoi je m'y suis mis diapo 21, 0 : 12 : 09

Quand on est comme moi l'inventeur d'un petit jeu de triangles...

Pourquoi je m'y suis mis

... ça fait un prétexte pour discuter avec des matheux surdoués.

Exercice pratique sur un de mes listings (génération pseudo-aléatoire à base d’automates cellulaires). Verdict : Python trois fois plus concis avec le lambda-calcul.

Premières impressions au bout de six semaines à lire de la doc, en n’ayant toujours pas écrit un seul listing et seulement réussi quelques exercices dans un interpréteur...

Une espèce d’assembleur logique, certes pas inutilisable mais amenant à employer surabondamment des techniques qu’on croyait obsolètes.

```c
typedef struct {
    int numTri;
    double x0, y0;
    double x1, y1;
    double x2, y2;
    int numSommet0, numSommet1, numSommet2;
    double perimetre;
    double superficie;
    BOOL touchePerimetre;
    int red;
    int green;
    int blue;
} Triangle;
```

Par exemple, quand on a l'habitude d'utiliser des structures fourre-tout dans ce gout-là...

```haskell
compteTriangles :: ([Int], [[Double]], [[Int]], [Double], [Double], [Bool], [[Int]]) -> Int
compteTriangles mesTriangles = resultat
  where
    (numTri, coord, numSommets, perimetre, superficie, touchePerimetre, couleurs) = mesTriangles
    resultat = length numTri
```

... voilà à peu près ce qu’on obtient quand on essaie de transcrire bovinement la logique à laquelle on est habitué.

Une autre façon d’être illisible : des morceaux de code beaucoup plus brefs et donc plus clairs, mais un fil conducteur plus difficile à suivre.

```haskell
yAUnChar :: [Char] -> Char -> (Bool, Int)
yAUnChar chaine charCherche = yAUnChar' chaine charCherche (length chaine)

yAUnChar' :: [Char] -> Char -> Int -> (Bool, Int)
yAUnChar' [] _ _ = (False, -1)
yAUnChar' (x:xs) charCherche longIni =
  if x == charCherche
    then (True, (longIni - length(x:xs)))
    else yAUnChar' xs charCherche longIni
```

Moins utile de nommer les "variables". Difficile de nommer les fonctions (d’où la pratique des ’ et des ”).

Des comptes rendus d'erreur parfois cryptiques. Par exemple, si on ose écrire ceci...

```
dansIntervalle :: Int -> Int -> Int -> Bool
dansIntervalle monInt borneInf borneSup = monResultat
  where
    monResultat = if (monInt > borneInf) and (monInt <= borneSup)
                  then True
                  else False
```

```
Prelude> :l cryptique.hs
[1 of 1] Compiling Main             ( cryptique.hs, interpreted )

cryptique.hs:5:22:
    Couldn't match expected type `t -> t1 -> Bool'
           against inferred type `Bool'
    In the expression: (monInt > borneInf) and (monInt <= borneSup)
    In the expression:
        if (monInt > borneInf) and (monInt <= borneSup) then
            True
        else
            False
    In the definition of `monResultat':
        monResultat = if (monInt > borneInf) and (monInt <= borneSup) then
                          True
                      else
                          False
Failed, modules loaded: none.
Prelude>
```

... voici comment l’interpréteur réagit.

Un mal de chien à utiliser la sortie standard pour essayer de comprendre ses bugs.

Licence BSD. Un gourou chez Microsoft.

"We’ve used Haskell as a laboratory for exploring advanced type system ideas. And that can make things complicated."

Concept d’**inférence de types**.

"At some point, maybe it will become just too complicated for any mortal to keep their head around, and then perhaps it’s time for a new language – that’s the way that languages evolve."

Haskell est comme le mariage...

... il vous donne tous les moyens de **résoudre les problèmes que vous n’auriez jamais eus sans lui**.

On y prend goût.

On a du mal à l’écrire, mais quand enfin on a réussi à faire taire le compilateur, ça marche vraiment très bien.

Comparaison personnelle avec l’effet que m’a fait Java.

On referme la doc et on l’oublie (même qu’on a tort, mais au moins on n’est pas bloqué).

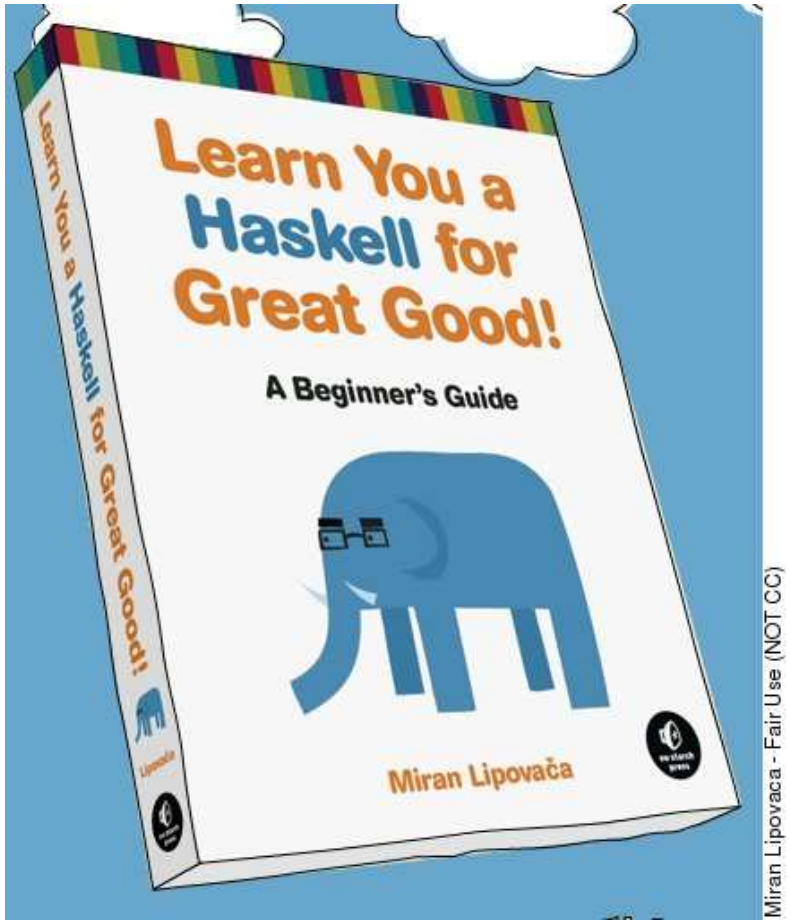
On se met à réfléchir à tout autre chose.

Pub. C’est aussi facile à écrire que du Python, et ça s’exécute aussi vite que du C.

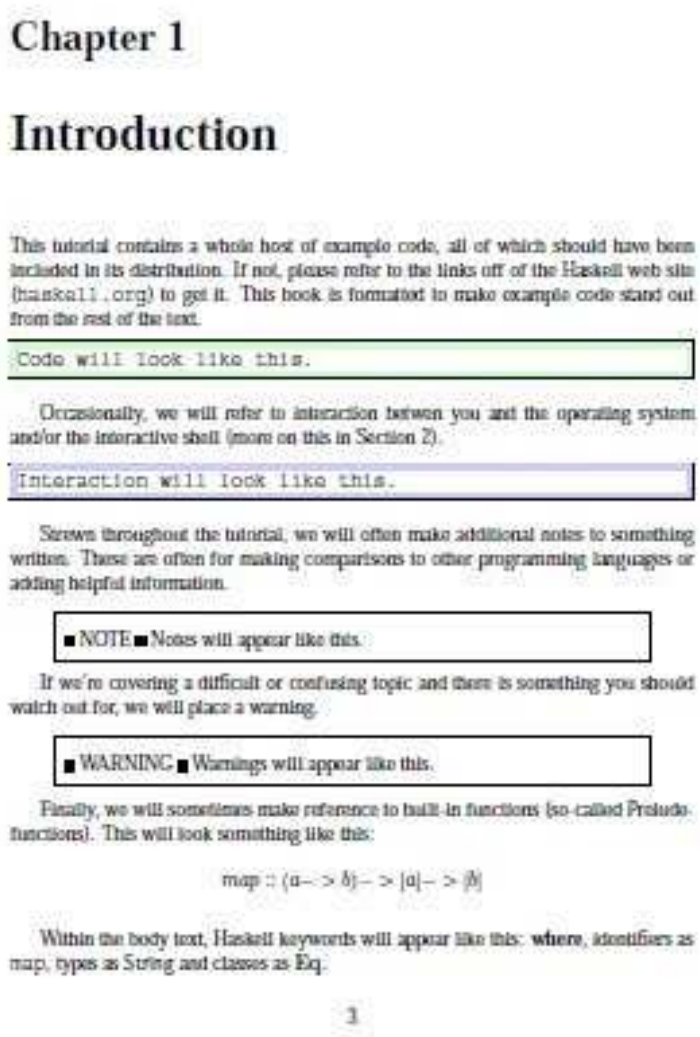
Pas mal d’amateurs y ont repéré une *"killer feature"*. Dans mon cas, c’est le triplet *Maybe*, *Just* et *Nothing*.

Idées que je voudrais faire passer : si on comprend juste quelques petites choses déconcertantes (comme la question des IO), on fait tout de suite sauter les barrières et on prend du plaisir. L’élégance, ça s’acquiert plus lentement et j’en suis encore très loin, mais on comprend d’où elle peut venir.

Ressources pédagogiques



Miran Lipovaca.



Chapter 1

Introduction

This tutorial contains a whole host of example code, all of which should have been included in its distribution. If not, please refer to the links off of the Haskell web site (`haskell.org`) to get it. This book is formatted to make example code stand out from the rest of the text.

```
Code will look like this.
```

Occasionally, we will refer to interaction between you and the operating system and/or the interactive shell (more on this in Section 2).

```
Interaction will look like this.
```

Strewn throughout the tutorial, we will often make additional notes to something written. These are often for making comparisons to other programming languages or adding helpful information.

■ NOTE ■ Notes will appear like this.

If we're covering a difficult or confusing topic and there is something you should watch out for, we will place a warning.

■ WARNING ■ Warnings will appear like this.

Finally, we will sometimes make reference to built-in functions (so-called Prelude functions). This will look something like this:

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

Within the body text, Haskell keywords will appear like this: **where**, identifiers as map, types as String and classes as Eq.

1

Fair Use (NOT CC)

Yet another Haskell Tutorial.

Rosetta Code.

Ghci. Rapport avec l'inférence de types.

Haskell IO for imperative programmers.

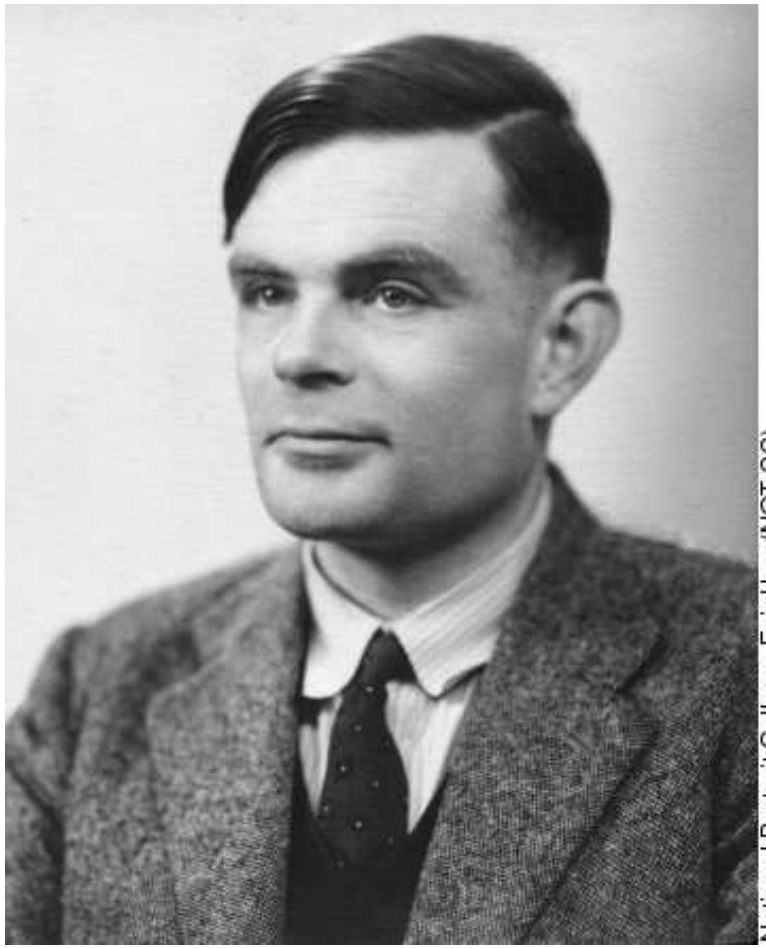
"I don’t want to show you how to do imperative programming in Haskell ; I want to show you how not to."

"It is possible to write imperative programs in Haskell ! Easy, even. Now that you know you can, don't."

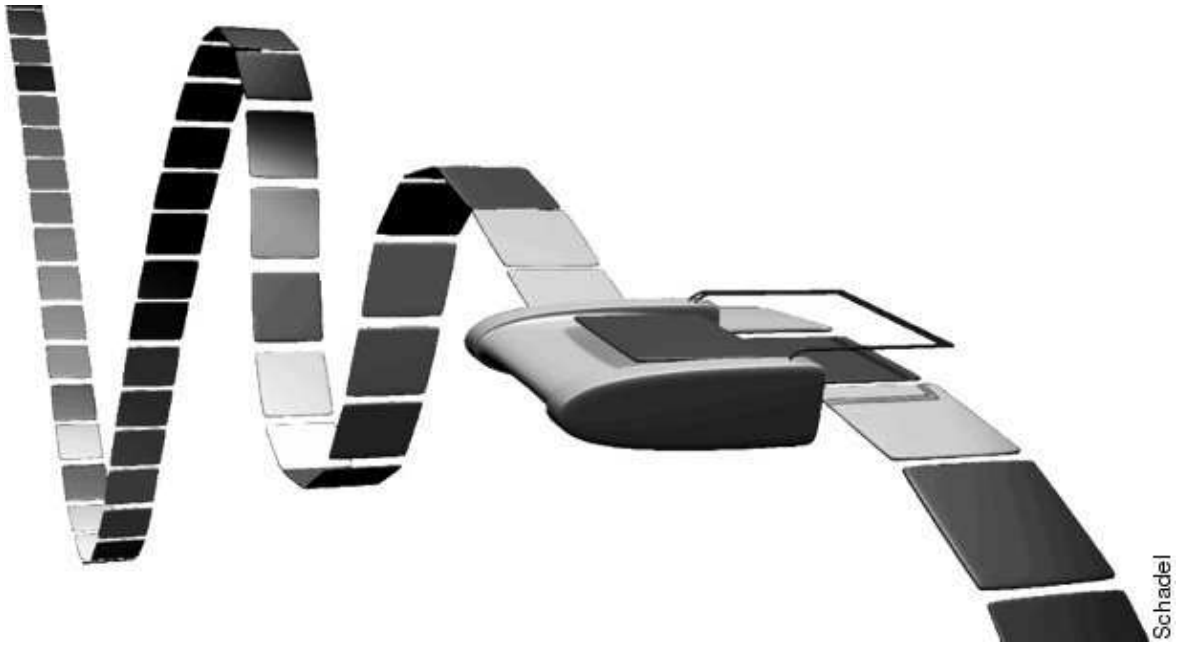
Peyton-Jones. Interview à Computer World (2008).

Google Talk "amuse-bouche" de Mark Lentczner. *"It twists your mind in a delightful way."*

Un peu d’histoire



Alan Turing.



La machine de Turing.

Alonzo Church.

Première stabilisation du langage en 1998. Haskell 98 reste la référence la mieux documentée, mais de nombreux modules complémentaires sont venus l'enrichir.

Eloge de la paresse

"To say that Haskell is lazy means that it is aggressively non-strict – it never evaluates anything before it absolutely needs to (unless you tell it otherwise, but that's another story). ***This changes the way you code.****"*

"In a lazy language, you evaluate expressions only when their value is actually required, not when you call a function – it's call by need. A lazy person waits until their manager says 'I really need that report now', whereas an eager will have it in their draw all done, but maybe their manager will never ask for it."

"I’m sure you’ve seen some attempts to explain laziness by way of apparently contrived examples involving infinite lists, but ***if you’re coming to Haskell from an imperative background you don’t get it. Trust me, you don’t.***"

Initiation à l'évaluation paresseuse par les tubes *(pipes)*.

```
find . | grep "\.gif" | more

find . | grep "\.gif" | sort | more
```

Le nœud du problème

Le nœud du problème diapo 68, 0 : 51 : 10

"Basically, in a functional language, you shouldn’t be doing input/output in an expression because input/output is a side effect."

Le nœud du problème diapo 69, 0 : 52 : 23

Parties de code pures et impures. Monades.

Le nœud du problème diapo 70, 0 : 54 : 04

*"The secret to success is to **start thinking about IO as a result that you achieve rather than something that you do**."*

Le nœud du problème diapo 71, 0 : 55 : 08

"Make sure that your education includes not just reading a book, but actually writing some functional programs, as it changes the way you think about the whole enterprise of programming. It's like if you can ski but you've never snowboarded : you hop on a snowboard and you fall off immediately."

Syntaxe : les fonctions "first class citizens"

```c
int valeurAbsolue (int maValeur) {
    if (maValeur < 0) {
        maValeur = maValeur * -1;
    }
    return maValeur;
}
```

```
---  valeur_absolue.c   Bot L31    (C/l Abbrev)--
```

```haskell
-- valeurAbsolue :: Int -> Int
-- valeurAbsolue :: Double -> Double
-- valeurAbsolue :: (Num a, Ord a) => a -> a
valeurAbsolue x = x'
  where
    x' = if x < 0
           then (x * (-1))
           else x
```

```
**-  valeur_absolue.hs   Bot L22    (Haskell Ind
```

Usage très différent des parenthèses. Très possible (quoique je ne le fasse guère) qu'une fonction soit envoyée en argument, ou qu'elle renvoie comme résultat une autre fonction. C'est nécessaire de le savoir pour comprendre certains aspects bizarres de la syntaxe, mais on n'est pas obligé de travailler comme ça quand on débute.

```haskell
trim :: String -> String
-- Nota: trim, trimLeft et trimRight ont ete recuperes sur le wiki
-- Rosetta Code. Je n'y reconnais pas mon style, et pour etre franc,
-- je n'ai meme pas essaye de les comprendre
trim = trimLeft . trimRight

trimLeft :: String -> String
trimLeft = dropWhile isSpace

trimRight :: String -> String
trimRight str | all isSpace str = ""
trimRight (c : cs) = c : trimRight cs
```

Les compositions de fonctions avec . et $. Le "syntactic sugar" (auquel j'ai très peu recours). Mentionner Lentczner.

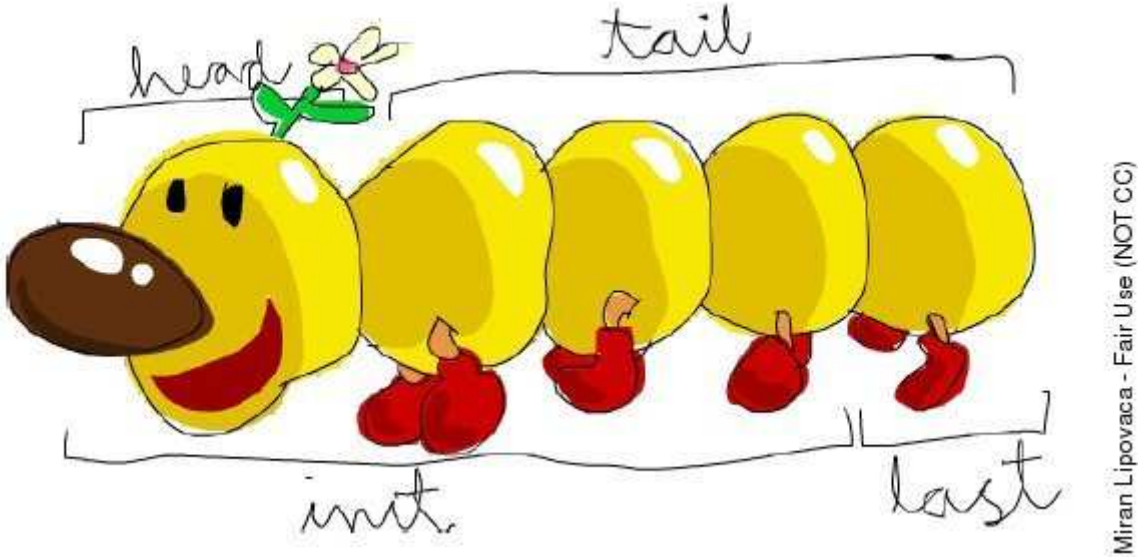
Syntaxe : listes et tuples

Les listes **[35, 12, 17, 24]** ne peuvent contenir que des éléments du même type. Quand elles sont renvoyées par une fonction, leur longueur n'est pas prédéterminée.

Les tuples

(78, "Yvelines", "Versailles", ["Mantes", "Rambouillet", "Saint-Germain"])

peuvent avoir des éléments de types différents, mais un tuple envoyé en argument ou renvoyé par une fonction a une structure connue et prédéterminée (bref, il est décrit dans la signature des fonctions).



Le monstre de Lipovaca : **head**, **tail**, **init**, **last**.
On peut aussi mentionner **take** et **drop**.

On récupère un élément précis d'une liste avec la notation
maListe ! ! index

La notation **(x :xs)**.

Les listes infinies... notamment la seule qu'on utilise couramment : **[0..]**. Utilisation courante avec **zipWith**.

Les listes en compréhension.

[x <- maListe, x >= limiteInf, x < limiteSup]

Syntaxe : le lambda-calcul

Les fonctions anonymes, du genre

(†x -> x / 2 + 5)

maNouvelleListe = **map** maFonction maListe

maNouvelleListe = **map** (†x -> x / 2 + 5) maListe

maListeReduite = **filter** maFonction maListe
maListeReduite = **filter** (†x -> x <= 0) maListe

Très déconcertant mais d'usage courant : les listes de tuples faites exprès pour un map.

.

```
mesPtsAvecCentre =
map (†monPoint -> ([xCentre, yCentre], monPoint))
mesPoints
```

.

```
mesDirections =
map (†(dir, dist) -> dir) (map convRecPol
mesPtsAvecCentre)
```

"Even though you've written code that looks like it's constructing a huge list of data in memory, that's not what's happening. You've actually written instructions for generating a sequence of data ***as you need them****."*

Enfin, pour des opérations du type totalisation, **foldr** et **foldl**.

.

foldl (+) 0 maListe

.

Seul *foldr* peut travailler sur des listes infinies.

Syntaxe : oubliez le C

Syntaxe : oubliez le C

```c
#include <stdio.h>
#include <stdlib.h>

double addition (double, double);

int main (int argc, char** argv) {
    double monResultat;
    double gloups1 = 2.0;
    double gloups2 = -2.0;

    monResultat = addition(gloups1, gloups2);

    return 0;
}

double addition (double arg0, double arg1) {
    double monResultat;

    monResultat = arg0 + arg1;

    return monResultat;
}
```

```
---  exemple_a_la_con.c    All L23     (C/l Abbrev)--------
```

```haskell
import IO

main = do
    let gloups1 = 2.0
    let gloups2 = (-2)
    let monResultat = ( addition gloups1 gloups2 )
    putStr ""

addition :: Double -> Double -> Double
addition x y = z
  where
    z = x + y
```

```
---  exemple_a_la_con.hs    All L13     (Haskell Ind Doc)--
(No changes need to be saved)
```

Exemples niais.

La récursivité

```haskell
repeteChar :: Char -> Int -> [Char]
repeteChar _ 0 = ""
repeteChar monChar combien = monChar:repeteChar monChar (combien - 1)
```

Répétition d’un caractère.

```haskell
import IO

main = do
    let maChaine = "portez ce vieux whisky au juge blond qui fume"
    let maChaineBis = quicksort maChaine
    putStrLn maChaineBis

quicksort :: [Char] -> [Char]
quicksort [] = []
quicksort (x:xs) = nouvelleChaine
  where
    infOuEgal      = filter (\y -> (y <= x) ) xs
    strictementSup = filter (\y -> (y > x ) ) xs

    nouvelleChaine = (quicksort infOuEgal) ++ [x] ++ (quicksort strictementSup)
```

Quicksort avec des fonctions anonymes.

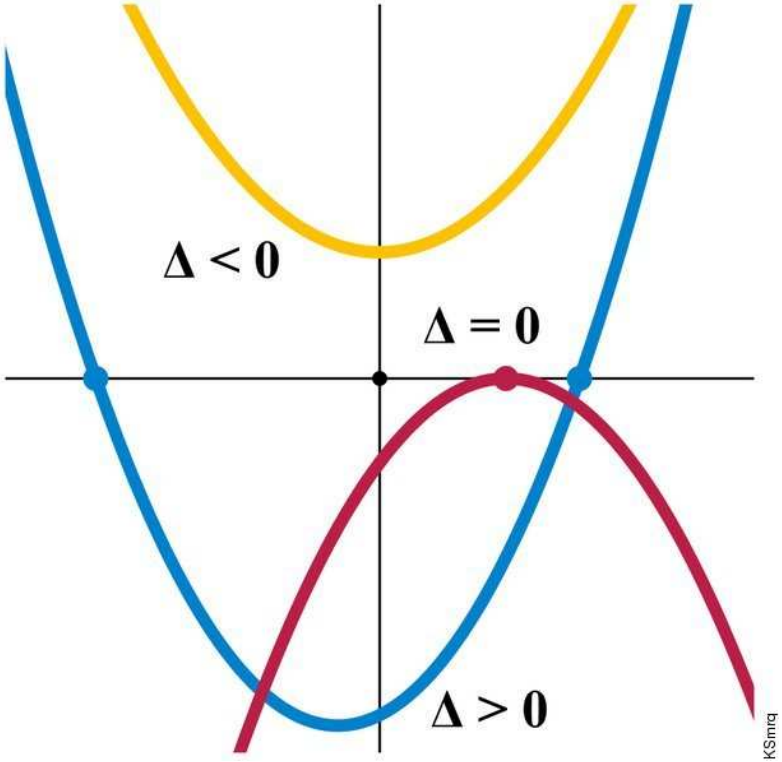
```haskell
import IO

main = do
    let maChaine = "portez ce vieux whisky au juge blond qui fume"
    let maChaineBis = quicksort maChaine
    putStrLn maChaineBis

quicksort :: [Char] -> [Char]
-- quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = nouvelleChaine
  where
    infOuEgal      = [a | a <- xs, a <= x]
    strictementSup = [a | a <- xs, a > x ]

    nouvelleChaine = (quicksort infOuEgal) ++ [x] ++ (quicksort strictementSup)
```

Quicksort avec des listes en compréhension.

```
[faux_code] $
[faux_code] $ ghc --make quicksort.hs -o quicksort
[faux_code] $ ./quicksort
        abcdeeeeefghiiijklmnoopqrstuuuuuvwxyz
[faux_code] $
```

Comme c’est beau, ça marche.

Maybe, Just et Nothing



Contrairement à ce que pensent les shadoks, quand il n'y a pas de solution il y a un problème.

En impératif (ici, C), comment on fait quand on n'est pas sûr d'obtenir un résultat utilisable.

```c
typedef struct {
    BOOL deltaPositifOuNul;
    double xPrime;
    double xSeconde;
} resultatEqSecDeg;

resultatEqSecDeg* resoudEquatSecDeg (resultatEqSecDeg* resultat,
                                     double alfa, double bravo,
                                     double charlie) {
    double delta;
    double racineDelta;

    delta = bravo * bravo - 4.0 * alfa * charlie;
    if (delta >= 0) {
        resultat->deltaPositifOuNul = TRUE;
        racineDelta = sqrt(delta);
        resultat->xPrime = ((bravo * -1) + racineDelta) / 2.0;
        resultat->xSeconde = ((bravo * -1) - racineDelta) / 2.0;
    } else {
        resultat->deltaPositifOuNul = FALSE;
    }

    return resultat;
}
```

--- **equation_second_degre.c** 14% L14 (C/l Abbrev)----------

La même chose en Haskell...

```haskell
resoudEquatSecDeg' :: Double -> Double -> Double -> [Maybe Double]
resoudEquatSecDeg' alfa bravo charlie = monResultat
  where
    delta = bravo * bravo - 4.0 * alfa * charlie
    condition = (delta >= 0)
    racineDelta = if condition
                  then Just (sqrt delta)
                  else Nothing
    racineDelta' = if condition
                   then (\(Just x) -> x) racineDelta
                   else -1 -- on s'en fout, on s'en sert pas
    xPrime    = if condition
                then Just (((bravo * (-1)) + racineDelta') / 2.0)
                else Nothing
    xSeconde = if condition
                then Just (((bravo * (-1)) - racineDelta') / 2.0)
                else Nothing

    monResultat = [xPrime, xSeconde] -- ou encore xPrime:xSeconde:[]
```

... ou plutôt :

```haskell
resoudEquatSecDeg :: Double -> Double -> Double -> Maybe [Double]
resoudEquatSecDeg alfa bravo charlie = monResultat
  where
    delta = bravo * bravo - 4.0 * alfa * charlie
    condition = (delta >= 0)
    racineDelta = if condition
                  then Just (sqrt delta)
                  else Nothing
    racineDelta' = if condition
                   then (\(Just x) -> x) racineDelta
                   else -1 -- on s'en fout, on s'en sert pas
    xPrime   = if condition
               then Just (((bravo * (-1)) + racineDelta') / 2.0)
               else Nothing
    xSeconde = if condition
               then Just (((bravo * (-1)) - racineDelta') / 2.0)
               else Nothing

    monResultatIntermed = [xPrime, xSeconde] -- ou encore xPrime:xSeconde:[]
    monResultat = if condition
                  then Just (map (\(Just x) -> x) monResultatIntermed)
                  else Nothing
```

Un squelette pour Haskell

Attaquez-vous à des listes très abondantes, structurez-les en SGML de bas de gamme, et travaillez avec des tubes *(pipes)*.

```haskell
module Main
    where

import IO
import System( getArgs )
import System.Exit
import JeLisPasLesDocs

main = do
    hSetBuffering stdin LineBuffering
    args <- getArgs
    plantagePropre (length args == 0) "Indiquez un mot-graine en argument."

    let graine = args !! 0
    enVrac <- getContents
    let myBigList = lines enVrac
    let baliseLue = "trucmuche"

    let indxBaliseDebut = trouveBaliseDebut myBigList baliseLue
    plantagePropre (indxBaliseDebut == (-1)) "La balise de debut est absente."
    let indxBaliseFin = trouveBaliseFin myBigList baliseLue
    plantagePropre (indxBaliseFin == (-1)) "La balise de fin est absente."

    let myReducedList = contenuBalise myBigList baliseLue
    let mesResultats = traiteListe graine myReducedList
    putStr (unlines mesResultats)
```

Exemple de module principal.

```haskell
module JeLisPasLesDocs (
   indexOf,
   jlaSplit,
   plantagePropre,
   showDouble,
   substr,
   trim
) where

import System.Exit
import Data.Char

substr :: [a] -> Int -> Int -> [a]
-- M'etonnerait que Haskell n'ait pas prevu quelque chose en standard
-- pour ca, mais vous n'esperez quand meme pas que je lise la doc?
substr chaine prems auDela =
   take (auDela - prems) (drop prems chaine)
```

Exemple de module secondaire.

Finalement, à quoi ça sert ?

Finalement, à quoi ça sert ? diapo 107, 1 : 37 : 16

"Do one thing and do it well."

Finalement, à quoi ça sert ? diapo 108, 1 : 37 : 48

*"Haskell has a sort of unofficial slogan : **avoid success at all costs**. (...) If you're sure you're going to succeed it's probably not research."*

Le risque, c'est de consacrer de l'énergie à un langage dont la pérennité n'est pas assurée... et d'autant plus qu'on y prend goût.

Finalement, à quoi ça sert ? diapo 110, 1 : 38 : 52

"There's a big mental rewiring process that happens when you switch from C++ or Perl to Haskell. And that comes just from being a purely functional language, not because it's particularly complex. Any purely functional language requires you to make that switch."

Finalement, à quoi ça sert ? diapo 111, 1 : 40 : 46

"You should teach purely functional programming in some shape or form as it makes you a better imperative programmer. Even if you're going to go back to C++, you'll write better C++ if you become familiar with functional programming."

Finalement, à quoi ça sert ? diapo 112, 1 : 41 : 00

Python. JavaScript (ECMAscript).

Finalement, à quoi ça sert ? diapo 113, 1 : 42 : 13

OCaml.

Finalement, à quoi ça sert ? diapo 114, 1 : 42 : 40

"Mainstream languages are, by default, dangerous for parallel evaluation. And purely functional languages are by default fine at parallel evaluation."

Licence

L'ensemble de la présentation est sous licence CC-BY-SA, à l'exception de quelques images en fair use*, signalées comme telles quand c'était nécessaire.*