Présentation de Haskell

Version 1

Jean-Luc JOULIN

11 octobre 2015

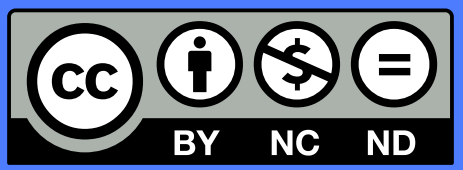
License

Cette présentation est diffusée suivant les termes de la license Creative Common :

BY **Attribution**. Cette présentation peut être librement utilisée, à condition de l'attribuer à l'auteur en citant son nom.

NC **Pas d'utilisation Commerciale**. Aucune utilisation commerciale n'est permise.

ND **Pas de Modification**. Aucune œuvre dérivée basée sur cette présentation n'est autorisée.



Coloration syntaxique

Cette présentation utilise une coloration syntaxique afin de faciliter la lecture du code présenté :

-  Mots clef du langage.
-  Les fonctions, les opérateurs.
-  Les types, les constructeurs.
-  Les valeurs numériques.
-  Les chaînes de caractères.
-  Les parenthèses, les crochets, les commentaires.

Sommaire

Sommaire

Présentation du language

Présentation de Haskell

- Langage fonctionnel pur.
- Basé sur la logique combinatoire.
- créé en 1990.
- Standard actuel : Haskell2010.

Haskell-plateform

- Plateforme de développement.
- Compilateur GHC.
- Bibliothèques standards.
- Outils associés facilitant le développement.
  - Gestionnaire de version (DARCS).
  - Gestionnaire de paquets (Cabal).
  - Générateur de documentation (Haddock).

Intérêts de Haskell

- Langage compilé.
- Langage déclaratif.
- Langage très expressif (peu de lignes de codes).
- Pas d'effets de bords.
- Code facile à comprendre et à maintenir.
- Génère un code fiable et performant.
- Code facilement parallélisable.
- Très adapté pour :
  - Le calcul scientifique.
  - La recherche opérationnelle.
  - Le calcul symbolique.
  - ...

Paradigme fonctionnelle

- Pas d'effets de bords.
  - Pas de variables (locales ou globales).
- Fonctions comme valeurs.
  - Fonctions comme paramètres.
  - Fonctions comme résultats.
- Transparence référentielle.
  - Le résultat de la fonction ne dépend que de ses arguments.
  - L'identificateur peut être remplacé par sa valeur.

Typage

- Typage fort
  - Pas de conversion automatique des types.
  - Détection des erreurs de types par le compilateur.
- Typage statique
  - Sûreté du typage.
  - Code plus rapide et moins coûteux en mémoire.

Inférence de types

- Recherche automatique des types de données.
- Permet d'utiliser le type le plus large possible.
- Facilite le polymorphisme.

Évaluation paresseuse (ou évaluation retardée)

- Évaluation d'une fonction seulement si nécessaire.
- Évite de calculer des résultats inutilisées.
- Permet d'utiliser des structures de données nouvelles (Listes infinies, ...)

Mode interactif

- Mode interactif avec GHCI.
- Permet de tester le résultat des fonctions plus facilement.
- Permet de modifier et de combiner des fonctions.
- Vérification du typage.

Syntaxe du langage

Règles de nommage des fonctions et des types

- Les noms ne doivent pas commencer par des chiffres.
- Les noms ne doivent pas contenir de ponctuation.
- Tous les caractères Unicode (UTF-8) peuvent être utilisés.
- Les noms de fonctions doivent commencer par une minuscule.
- Les noms de types doivent commencer par une majuscule.
- Utilisation du nommage en "camelCase" recommandé.

Règles de nommage des opérateurs

- Les noms ne doivent pas contenir de lettre ni de chiffres.
- Les noms d'opérateurs ne peuvent contenir que :

$ % & # * + . / <>= | ?@^~

Exemples de nommage

`maPremiereFonction`	Nom de fonction correcte.
`MonPremierType`	Nom de type correcte.
`1EssaiDeFonction`	Nom incorrecte.
`<*>`	Nom d'opérateur correcte.

Règles d'indentation de base

- L'indentation permet de définir des blocs de code.
- Le code qui fait partie d'une expression doit être indenté plus que le début de cette expression.
- L'indentation peut être obtenue par des espaces ou des tabulations.
- La quantité de caractères est libre.

 L'utilisation des espaces pour l'indentation est recommandée.

Exemples d'indentations

```
fonct x = if x == 5
  then "Gagne"
  else "Perdu"
```

Indentation correcte.

```
fonct x = if x == 5
then "Gagne"
else "Perdu"
```

Indentation incorrecte.

```
fonct x = if x == 5
  then "Gagne"
    else "Perdu"
```

Indentation correcte.

Structure d'un fichier source Haskell

```haskell
import Data.List
import MonModule


fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)

lstNom =
  [
    "Sylvain"
  , "Olivier"
  , "Victor"
  ]

main :: IO ()
main =
  putStrLn "Programme d'essai Haskell"
  let var = maPremiereFonction 10 5 6
  putStrLn (show var)
  putStrLn (show (fibonacci 12))
```

Importations de modules

Création de fonctions

Fonction principale

Structure d'un module Haskell

```
module MonModule
  (
    maPremiereFonction
  , uneAutreFonction
  )
  where

import ...
import ...
import MonAutreModule

maPremiereFonction a b c = if
                    then uneFonctionInterne a b
                    else uneAutreFonction (b + c)

uneAutreFonction t = t ^ 4

uneFonctionInterne a b = a ^ 2 + b   --Cette fonction n'est
    pas exportee
```

Nom du module

Fonctions à exporter

Importations d'autres modules

Corps du module

Compilation

- GHC **G**lasgow **H**askell **C**ompiler.
- Compiler un fichier source :

```
ghc --make Prog.hs
```

- Compiler et afficher les messages d'erreurs standards :

```
ghc --make -W source.hs
```

- Compiler et afficher tous les messages d'erreurs :

```
ghc --make -Wall source.hs
```

Mode intéractif

- GHCI **G**lasgow **H**askell **C**ompiler **I**nteractive.
- Lancement du mode intéractif :

```
ghci
```

- Chargement d'un fichier source :

```
ghci   Module.hs
```

Les types de base

Bool	Un booléen.
Int	Un entier limité par l'architecture de la machine.
Integer	Un entier sans limitation de taille.
Float	Un nombre à virgule flottante simple précision.
Double	Un nombre à virgule flottante double précision.
Char	Un caractère.
String	Une chaîne de caractères.

Les tuples

- Permet de contenir plusieurs éléments de types identiques ou différents.
- Structure figée.

`(a,a)` Un couple d'éléments a.

`(a,b,c)` Un triplet d'éléments a, b et c.

Les listes

- Permet de contenir plusieurs éléments de types identiques.
- Structure extensible.
- Peut être vide.

`[a]`	Une liste d'éléments a.
`[(a,a)]`	Une liste de couples d'éléments a.
`[(a,b,c)]`	Une liste de triplets d'éléments a, b et c.
`[]`	Une liste vide.

Nouveaux types de données (data)

- Création de nouveaux types de donnés personnalisés.

```
data Forme = Cercle    (Double,Double) Double
           | Rectangle (Double,Double) Double
                       Double
```

- Les constructeurs disponibles pour créer le type Forme sont :

```
Cercle    (10,20) 40
Rectangle (30,40) 7 9
```

Enregistrements

- Possibilité de nommer les valeurs d'un type de données.

```
data Forme = Cercle    {
                         position :: (Double,Double)
                       , diametre ::  Double
                       }
           | Rectangle {
                         position :: (Double,Double)
                       , largeur  ::  Double
                       , hauteur  ::  Double
                       }
```

Champs des enregistrements

- Les différents champs des enregistrements sont accessibles par leur noms.

```
position $ Cercle (10,20) 40
largeur $ Rectangle (30,40) 7 9
```

Nouveaux types de données (type)

- Création d'un synonyme vers un type.
- Les nouveaux types peuvent être remplacés par leur définition et inversement.

```
type Nom = String
type String = [Char]
```

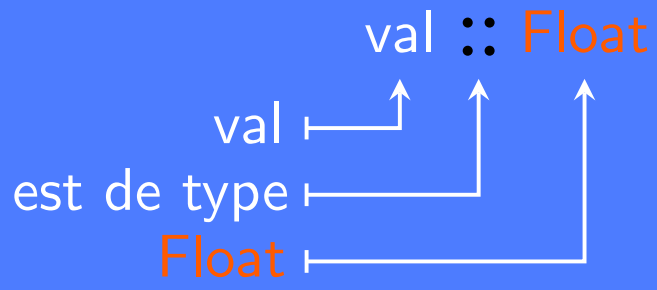
Nouveaux types de données (newtype)

- Création d'un nouveau type de données.
- Les nouveaux types ne peuvent pas être remplacés par leurs définitions.

```
newtype NumPoint = NumPoint Int
```

Identification du type d'une fonction

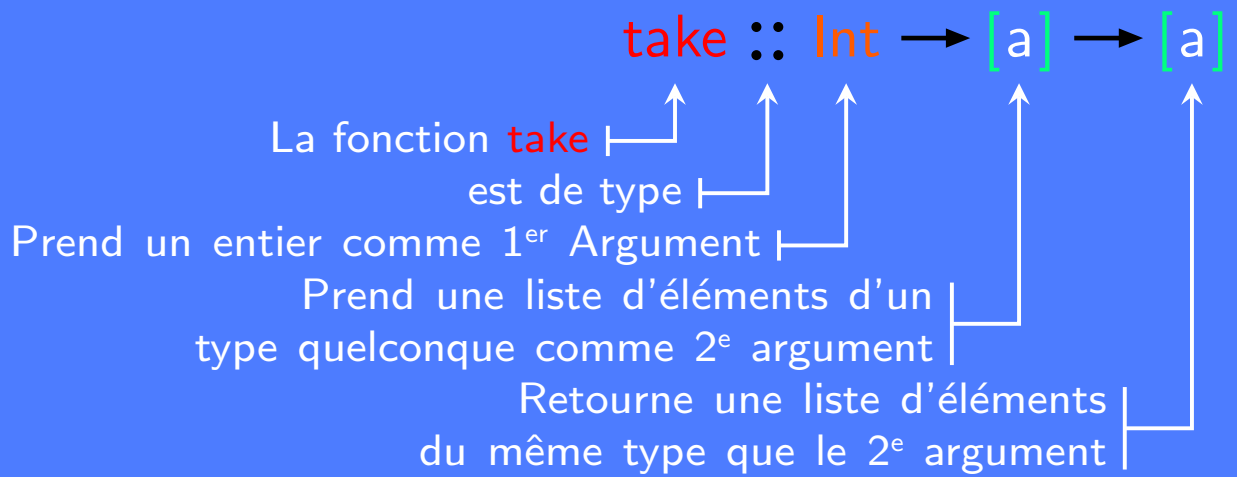
- :: signifie **"est du type"**.
- → signifie **"fonction"** avec :
  - à gauche le paramètre.
  - ce qu'elle retourne.



L'identification n'est pas obligatoire (inférence de type).

Identification du type d'une fonction

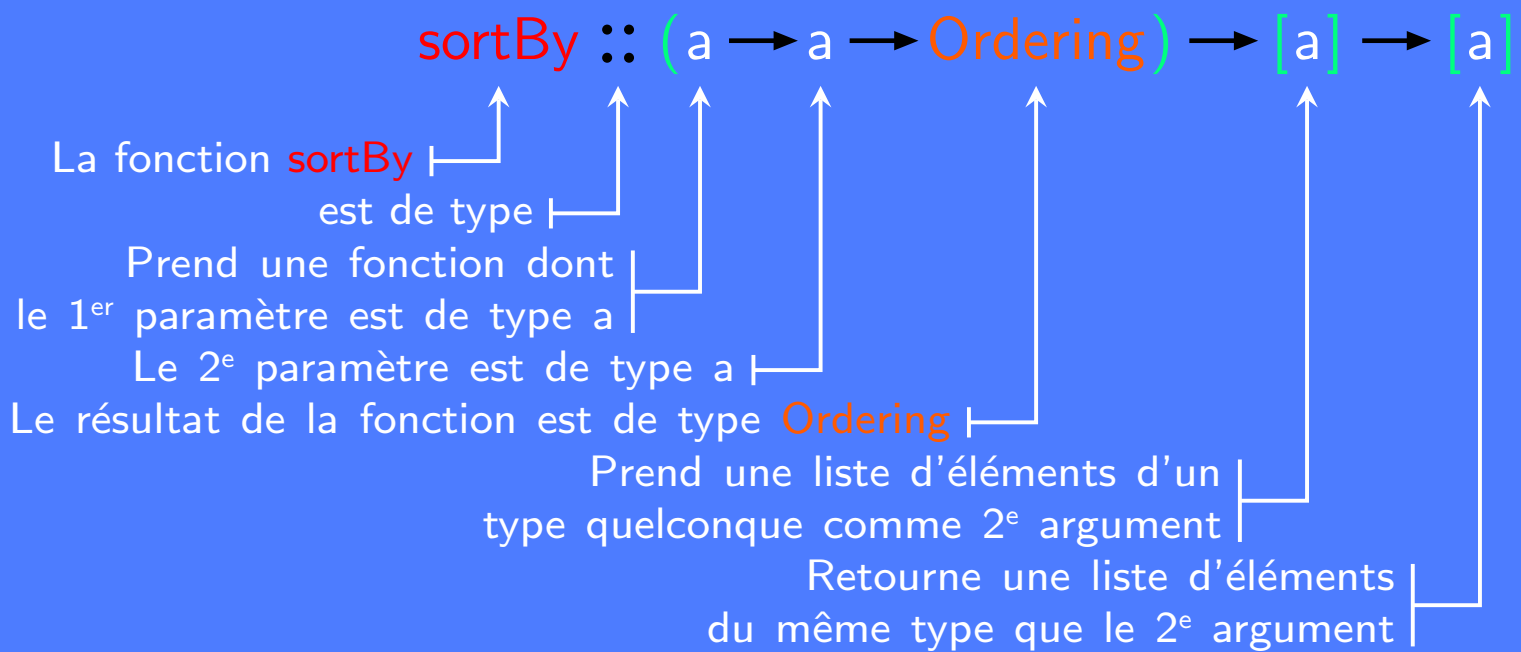
- :: signifie **"est du type"**.
- → signifie **"fonction"** avec :
  - à gauche le paramètre.
  - ce qu'elle retourne.



L'identification n'est pas obligatoire (inférence de type).

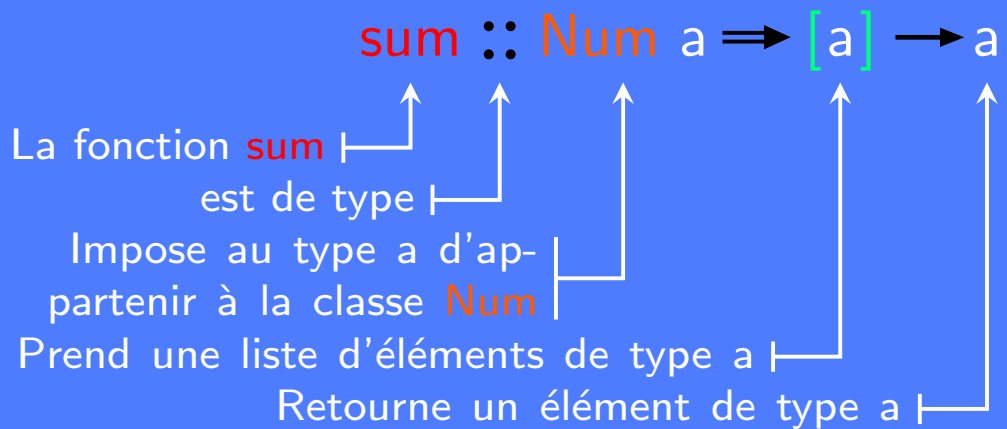
Fonction comme paramètre

- Une fonction peut prendre une autre fonction comme paramètre.



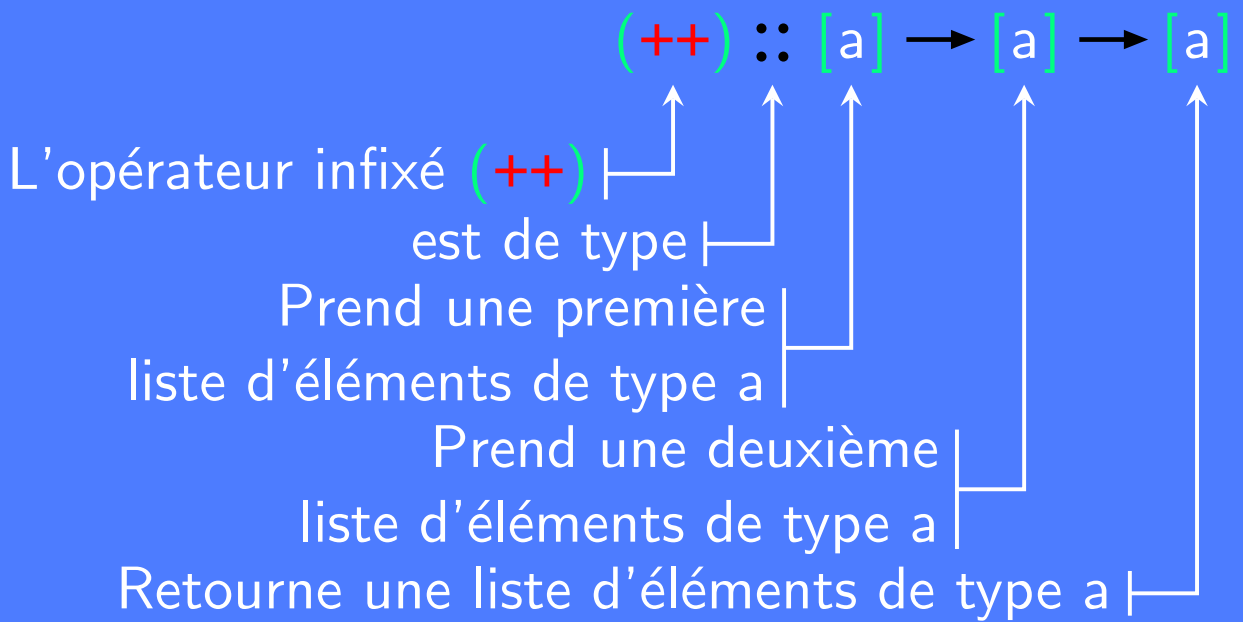
Contrainte de classe

- $\Rightarrow$ signifie **"contrainte de classe"**.
- Fonction polymorphe (Adaptable à plusieurs types de données).
- Peut prendre toutes les types appartenant à cette classe.



Opérateurs infixés

- ( ) signifie **"infixé"**.
- Fonction nativement infixée.



Conditions

- Mots clefs **if**, **then** et **else**.
- Teste une expression et retourne une valeur en fonction du résultat du test.

```
if a == 3
    then True
    else False
```

Une condition doit être décrite dans sa totalité (else obligatoire).

Case

- Mots clefs **case** et **of** ainsi que $\longrightarrow$.
- Permet d'exécuter des expressions en fonction de la valeur d'une variable.

```
case var of 1 -> "resultat 1"
            2 -> "resultat 2"
            3 -> "resultat 3"
            _ -> "autre"
```

 Si la structure case est parcourue sans rencontrer le motif correspondant, une erreur se produira.

- Utiliser _ (underscore) pour **"matcher"** à n'importe quoi.

Gardes

- Permet d'appliquer des tests successifs.
- Retourne la valeur associée au test réussis.
- Importance de l'ordre.
- Sont indiqués par des barres verticales : |

```
devinerNombre nb
  | nb < 10    == "Trops petit!"
  | nb > 10    == "Trops grand!"
  | otherwise == "Gagne!"
```

Les gardes sont évalués dans l'ordre.

Filtrage par motifs (pattern matching)

- Simplifie la définition de fonctions.
- Clarifie le code.

```
factoriel 0 = 1
factoriel n = n * factoriel (n - 1)
```

```
head (deb : _) =  deb
tail (_ : fin) =  fin
```

```
map _ []         = []
map f (x : xs) = f x : map f xs
```

Si un motif n'est pas définie dans sa globalité, une erreur se produira.

Filtrage par motifs (Pièges)

- Un motif doit être défini dans sa totalité.
- Risque d'erreurs fatale.

```
fonct 0 = "a"
fonct 1 = "b"
fonct 2 = "c"
```

```
Prelude> fonct 5
"*** Exception: ???.hs:(3,1)-(4,13): Non-exhaustive patterns
    in function fonct
```

- Utiliser _ (underscore) pour "**matcher**" à n'importe quoi.

```
fonct _ = "tout"
```

```
Prelude> fonct 5
 "tout"
```

Where

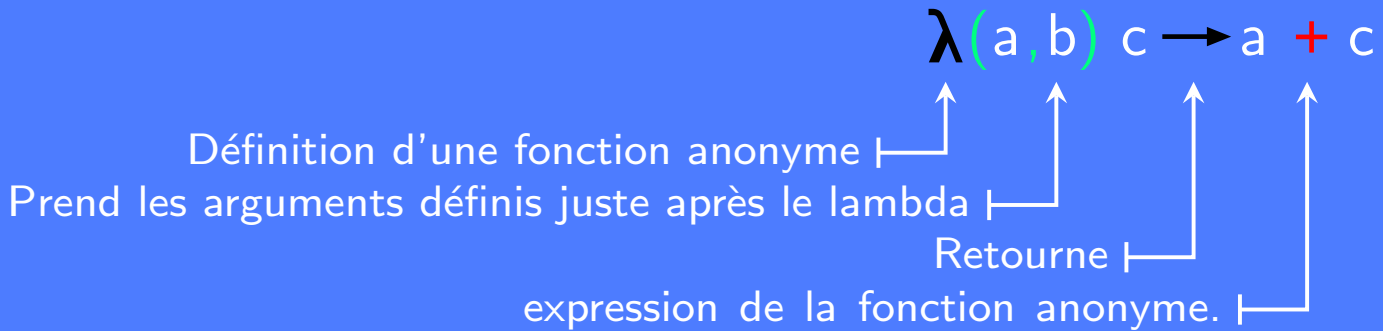
- **where** permet de définir des variables visibles dans toute une fonction.

```
equation typ a b c x
  | typ == Affine = val1
  | typ == Trinome = val2
  where val1 = a * x ^ 2 + b * x + c
        val2 = a * x + b
```

- Les fonctions définies avec **where** sont utilisables dans les tests et les gardes.

Fonctions anonymes (lambdas)

- Fonctions à usage unique.
- Utilisation très localisée.
- \ désigne la lettre grec **λ**.




```
Prelude> zipWith (\(a,b) c->a+c) [(6,1),(3,2),(4,5)] [9,1,2]
 [15,4,6]
Prelude> map (\(a,b,c)->a+b-c) [(1,2,3),(2,3,1)]
 [0,4]
```

Composition de fonctions

- Possibilité de composer des fonctions.
- . a le même sens que ∘ en mathématique.

$$f \;.\; g = \lambda x \rightarrow f\ (g\ x)$$

(.) :: (b → c) → (a → b) → a → c

Compose deux fonctions.

```
Prelude> (reverse . sort) [5,2,4,9,7,3,2]
 [9,7,5,4,3,2,2]
Prelude> (negate . floor) 5.23
 -5
```

Modules

- Ensemble de fonctions, de types et de classes en rapport entre elles.
- Permet d'utiliser des fonctions dans plusieurs programmes.
- Possibilité de créer ses propres modules.

Chargement d'un module

- Import simple.
  Charge toutes les fonctions du module Data.List.
- Import de fonctions choisies.
  ```
  import Data.List (sort)
  ```
  Charge uniquement la fonction sort du module Data.List.
- Import qualifié (nommé).
- Permet de résoudre les collisions de noms.
- Les fonctions devront être appelées avec un préfixe.
  ```
  import qualified Data.List as Listes
  ```

 L'import qualifié d'un module permet de résoudre les conflits de noms entre certains modules.

Création d'un module

- **module** Perso.Monmodule **where**

  -- Definitions des fonctions

  Créé le module Perso.Monmodule qui s'importe avec :
  import Perso.Monmodule
- Le fichier Monmodule.hs doit être placé dans le répertoire Perso.
- Le nom du module et des répertoires doivent commencer par une majuscule.

Modules standards de Haskell

Data.List	Opérations sur les listes.
Data.Ord	Pour gérer les types "ordonnables".
Data.Maybe	Le type Maybe et les fonctions associées.
Data.Either	Le type Either et les fonctions associées.
Data.Char	Pour manipuler les caractères texte.
Data.String	Pour manipuler les chaînes de caractères.
Data.Complex	Pour manipuler les nombres complexes.
Data.IORef	Pour gérer les types mutables.
System.Exit	Pour gérer la fin d'un programme.

Les listes

Taille d'une liste

null :: [a] → Bool

Test si une liste est vide.

```
Prelude> null []
 True
Prelude> null [1]
 False
```

length :: [a] → Int

Nombre d'éléments d'une liste.

```
Prelude> length [1,2,3,4,9,8,6]
 7
```

Présence d'un élément dans une liste

elem :: Eq a ⇒ a → [a] → Bool

Test la présence d'un élément dans une liste.

```
Prelude> elem 8 [1,3,4,6,2]
 False
Prelude> elem 'a' "Une liste de caracteres"
 True
```

notElem :: Eq a ⇒ a → [a] → Bool

Test l'absence d'un élément dans une liste.

```
Prelude> notElem 8 [1,3,4,6,2]
 True
```

Construction des listes

(:) :: a → [a] → [a]

Ajout d'un élément en tête de liste.

```
Prelude> 5 : [1,2,3,4,9,8,6]
 [5,1,2,3,4,9,8,6]
```

(++) :: [a] → [a] → [a]

Concaténation de deux listes.

```
Prelude> [1,2,3] ++ [4,9] ++ [8,6]
 [1,2,3,4,9,8,6]
```

map :: (a → b) → [a] → [b]

Applique une fonction à chaque élément d'une liste.

```
Prelude> map (2*) [1,2,3,4,9,8,6]
 [2,4,6,8,18,16,12]
```

Zipper des listes

zip :: [a] → [b] → [(a,b)]

Assemble deux listes dans une liste de doublets.
Existe pour 3, 4, 5, 6, listes.

```
Prelude> zip ["Jean","Pierre","Paul"] [10,20,14]
 [("Jean",10),("Pierre",20),("Paul",14)]
Prelude> zip3 ["Jean","Pierre","Paul"] ["MACHIN","BIDULE","TRUC"] [10,20
    ,14]
 [("Jean","MACHIN",10),("Pierre","BIDULE",20),("Paul","TRUC",14)]
```

zipWith :: (a → b → c) → [a] → [b] → [c]

Assemble deux listes dans une liste créée avec la fonction passée en argument.
Existe pour 3, 4, 5, 6, listes.

```
Prelude> zipWith (\a b->5*a+b) [1,2,3] [2,4,1]
 [7,14,16]
```

Dézipper des listes

unzip :: [(a,b)] → ([a],[b])

Décompose une liste de doublets en un doublet de listes.
Existe pour 3, 4, 5, 6, listes.

```
Prelude> unzip [("Jean",10),("Pierre",20),("Paul",14)]
 (["Jean","Pierre","Paul"],[10,20,14])
Prelude> unzip3 [("Jean","MACHIN",10),("Pierre","BIDULE",20),("Paul","
    TRUC",14)]
 (["Jean","Pierre","Paul"],["MACHIN","BIDULE","TRUC"],[10,20,14])
```

Découpage des listes

head :: [a] → a

Extraction du premier élément d'une liste.

```
Prelude> head [1,2,3,4,9,8,6]
 1
```

tail :: [a] → [a]

Extraction du reste d'une liste.

```
Prelude> tail [1,2,3,4,9,8,6]
 [2,3,4,9,8,6]
```

 Les fonctions head et tail ne doivent pas être utilisées sur des listes vides. Une erreur se produira.

Découpage des listes

init :: [a] → [a]

Extraction du dernier élément d'une liste.

```
Prelude> init [1,2,3,4,9,8,6]
 [1,2,3,4,9,8]
```

last :: [a] → a

Extraction du dernier élément d'une liste.

```
Prelude> last [1,2,3,4,9,8,6]
 6
```

 Les fonctions init et last ne doivent pas être utilisées sur des listes vides. Une erreur se produira.

Transformation des listes

reverse :: [a] → [a]

Inversion d'une liste.

```
Prelude> reverse [1,2,3,4,9,8,6]
 [6,8,9,4,3,2,1]
```

concat :: [[a]] → [a]

Concaténation d'une liste de listes.

```
Prelude> concat [[1,2,3],[4,9],[8,6]]
 [1,2,3,4,9,8,6]
```

Extraction dans une liste par nombre

take :: Int → [a] → [a]

Prend les premiers éléments d'une liste.

```
Prelude> take 3 [3,4,6,7,9,8,1,2]
 [3,4,6]
```

drop :: Int → [a] → [a]

Délaisse les premiers éléments d'une liste.

```
Prelude> drop 3 [3,4,6,7,9,8,1,2]
 [7,9,8,1,2]
```

splitAt :: Int → [a] → ([a],[a])

Découpe une liste en deux parties à un certain élément.

```
Prelude> splitAt 3 [3,4,6,7,9,8,1,2]
 ([3,4,6],[7,9,8,1,2])
```

Extraction dans une liste par prédicat

takeWhile :: (a → Bool) → [a] → [a]

Prend les éléments d'une liste tant que le prédicat est vrai.

```
Prelude> takeWhile (7 >=) [3,4,6,7,9,8,1,2]
 [3,4,6,7]
```

dropWhile :: (a → Bool) → [a] → [a]

Saute les éléments d'une liste tant que le prédicat est vrai.

```
Prelude> dropWhile (7 >=) [3,4,6,7,9,8,1,2]
 [9,8,1,2]
```

Recherche dans une liste

find :: (a → Bool) → [a] → Maybe a

Renvoi le premier élément conforme au prédicat.

```
Prelude> find (=="Pierre") ["Franck","Sylvain","Marc","Pierre"]
 Just "Pierre"
Prelude> find (=="Olivier") ["Franck","Sylvain","Marc","Pierre"]
 Nothing
```

findIndex :: (a → Bool) → [a] → Maybe Int

Renvoi l'indexe du premier élément conforme au prédicat. Les indexes sont comptés à partir de 0.

```
Prelude> findIndex (=='a') "Une liste de caracteres"
 Just 14
```

Recherche dans une liste

findIndices :: (a → Bool) → [a] → [Int]

Renvoi les indexes de tous les éléments conforme au prédicat. Les indexes sont comptés à partir de 0.

```
Prelude> findIndices (== 'a') "Une liste de caracteres"
 [14,16]
```

Filtrage d'une liste

filter :: (a → Bool) → [a] → [a]

Retourne une liste de tous les éléments conforment au prédicat.

```
Prelude> filter odd [1,2,3,4,5,6,7,8,9,10]
 [1,3,5,7,9]
Prelude> filter (\(a,b) -> b > 18) [("Jean",18),("Pierre",21)
                                   ,("Marc",16),("Max",20)]
 [("Pierre",21),("Max",20)]
```

partition :: (a → Bool) → [a] → ([a],[a])

Retourne un doublet contenant, une liste des éléments conforme au prédicat et une liste des éléments non conformes

```
Prelude> partition (\(a,b) -> b > 18) [("Jean",18),("Pierre",21)
                                      ,("Marc",16),("Max",20)]
 ([("Pierre",21),("Max",20)],[("Jean",18),("Marc",16)])
```

Suites

- Possibilité de créer des "**suites**" avec les listes.
- Utiliser `..` dans la définition de la liste.
- Borne inférieure (obligatoire) et supérieure (facultative).
- Possibilité de donner un incrément (suites arithmétiques).

```
Prelude> [1 .. 10]
 [1,2,3,4,5,6,7,8,9,10]
Prelude> [1,3 .. 10]
 [1,3,5,7,9]
Prelude> [20,18 .. 1]
 [20,18,16,14,12,10,8,6,4,2]
Prelude> ['c' .. 'u']
 "cdefghijklmnopqrstu"
```

- Possiblité de créer des listes infinies.

```
Prelude> [10 ..]
 [10,11,12,13,14,15,16,17,18,19, ...
```

Listes infinies

repeat :: a → [a]

Répète un élément sous la forme d'une liste.

```
Prelude> repeat 'a'
 "aaaaaaaaaaaa ...
```

cycle :: [a] → [a]

Répète une suite d'éléments en boucle dans une liste.

```
Prelude> cycle "abc"
 "abcabcabcabcabcabcabcab ...
```

iterate :: (a → a) → a → [a]

Répète une fonction de façon itérée sur chaque élément de la liste.

```
Prelude> iterate (2*) 1
 [1,2,4,8,16,32,64,128,256,512,1024, ...
```

Chaînes de caractères

- Type d'une chaîne de caractères : String.
- Équivalent à une liste de caractères : [Char].
- Une chaîne de caractères est donc une **liste**.
- Les fonctions applicables aux listes sont applicables aux chaînes de caractères :
  - length
  - elem
  - ...

Concaténation de chaînes de caractères

(:) :: a → [a] → [a]

Ajout d'un caractère au début de la chaîne.

```
Prelude> 'P' : ['r','e','m','i','e','r','s',' ','m','o','t','s']
 "Premiers mots"
Prelude> 'P' :  "remiers mots"
 "Premiers mots"
```

(++) :: [a] → [a] → [a]

Concaténation de deux chaînes de caractères.

```
Prelude> ['P','r','e','m','i','e','r','s'] ++ [' ','m','o','t','s']
 "Premiers mots"
Prelude> ['P','r','e','m','i','e','r','s'] ++ " " ++ "mots"
 "Premiers mots"
Prelude> 'P' : ['r','e','m','i','e','r','s'] ++ " " ++ "mots"
 "Premiers mots"
```

Découpage et assemblage de lignes

lines :: String → [String]

Découpe une chaîne de caractères en plusieurs aux niveau des retours de lignes.

```
Prelude> lines "Une ligne de texte\nUne autre ligne\nLa fin du texte"
 ["Une ligne de texte","Une autre ligne","La fin du texte"]
```

unlines :: [String] → String

Assemble plusieurs chaînes de caractères un insérant un retour à la ligne entre elles.

```
Prelude> unlines ["Une ligne de texte","Une autre ligne","La fin du
    texte"]
 "Une ligne de texte\nUne autre ligne\nLa fin du texte\n"
```

Découpage et assemblage de mots

words :: String → [String]

Découpe une chaîne de caractères en plusieurs aux niveau des espaces.

```
Prelude> words "Une ligne de texte"
 ["Une","ligne","de","texte"]
```

unwords :: [String] → String

Assemble plusieurs chaînes de caractères un insérant un espace entre elles.

```
Prelude> unwords ["Une","ligne","de","texte"]
 "Une ligne de texte"
```

Les entrées sorties

Affichage de texte

putStr :: String → IO ()

Affiche une chaîne de caractères sur la sortie standard.

```
Prelude> putStr "Hello World!"
 Hello World!Prelude>
```

putStrLn :: String → IO ()

Affiche une chaîne de caractères sur la sortie standard avec un retour à la ligne.

```
Prelude> putStrLn "Hello World!"
 Hello World!
```

Affichage de texte

print :: Show a ⇒ a → IO ()

Affiche un élément de la classe Show (affichable) sur la sortie standard avec un retour à la ligne.

```
Prelude> print (5 :: Int)
 5
Prelude> putStrLn (5 :: Int)
<interactive>:21:11:
    Couldn't match type 'Int' with '[Char]'
```

Lecture de texte

getChar :: IO Char

Lecture d'un seul caractère au clavier.

```
Prelude> getChar
 o
 'o'
```

getLine :: IO String

Lecture d'un chaîne de caractères au clavier. Validation de la chaîne avec la touche entrée.

```
Prelude> getLine
 Une ligne de texte
 "Une ligne de texte"
```

Lecture de fichiers textes

readFile :: FilePath → IO String

Lecture d'un fichier depuis l'adresse définie par FilePath.

```
Prelude> readFile "Un fichier.txt"
 "Fichier texte\nEssai de lecture\n"
Prelude> let fichier = readFile "Un autre fichier.txt"
Prelude> fichier
 "Un autre fichier texte\nEssai de lecture\n"
Prelude> let fichier2 = readFile "Un fichier inexistant.txt"
Prelude> fichier2
*** Exception: Un fichier inexistant.txt: openFile: does not exist (No
    such file or directory)
```

Tenter d'ouvrir un fichier qui n'existe pas provoque une erreur.

Évaluation retardée : Le fichier n'est ouvert que lorsqu'on en a besoin.

Écriture de fichiers textes

```
writeFile :: FilePath → String → IO ()
```

Écriture d'un fichier à l'adresse définie par FilePath.

```
appendFile :: FilePath → String → IO ()
```

Ajout en écriture à un fichier à l'adresse définie par FilePath.

Tests

opérateurs booléens

(&&) :: Bool → Bool → Bool

L'opérateur ET

```
Prelude> True && True
 True
```

(||) :: Bool → Bool → Bool

L'opérateur OU

```
Prelude> True || False
 True
```

not :: Bool → Bool

La négation

```
Prelude> not True
 False
```

opérateurs booléens

and :: [Bool] → Bool

L'opérateur ET appliqué sur une liste de booléens

```
Prelude> and [True,True,False]
 False
```

or :: [Bool] → Bool

L'opérateur OU appliqué sur une liste de booléens

```
Prelude> or [True,True,False]
 True
```

Les tests d'égalités

- Possibles seulement avec deux éléments du même type.
- Possibles seulement avec les types appartenant à la classe Eq.

```
<interactive>:2:17:
    No instance for (Eq Forme) arising from a use of ‘==’
    Possible fix: add an instance declaration for (Eq Forme)
```

- Dériver les nouveaux types avec **deriving** dans la définition du type.
- Créer des instances de classe avec **instance** pour les nouveaux types.

Les tests d'égalités

(==) :: Eq a ⇒ a → a → Bool

L'égalité

```
Prelude> 3 == 3
 True
Prelude> "Alain" == "Olivier"
 False
```

(/=) :: Eq a ⇒ a → a → Bool

L'inégalité

```
Prelude> 4 /= 5
 True
```

Les tests de comparaison

- Possibles seulement avec deux éléments du même type.
- Possibles seulement avec les types appartenant à la classe Ord.

```
<interactive>:3:17:
    No instance for (Ord Forme) arising from a use of ‘<’
    Possible fix: add an instance declaration for (Ord Forme)
```

- Dériver les nouveaux types avec **deriving** dans la définition du type.
- Créer des instances de classe avec **instance** pour les nouveaux types.

Les tests de comparaison

compare :: a → a → Ordering

La comparaison de deux données.
Retourne LT (inférieur), GT (supérieur) ou EQ (égale).

```
Prelude> compare 10 2
 GT
Prelude> compare 'f' 'eAccent'
 LT
Prelude> compare "Jean" "Jeanne"
 LT
Prelude> compare Nothing (Just 5)
 LT
```

Down :: a → Down a

Inverse l'ordre des éléments.

```
Prelude> compare (Down 10) (Down 2)
 LT
```

Attention les caractères sont comparés dans l'ordre du code UTF-8.

Les tests de comparaison

(<) :: a → a → Bool

Test l'infériorité entre le premier et le deuxième élément.

```
Prelude> 'a' < 'b'
 True
```

(≤) :: a → a → Bool

Test l'infériorité ou l'égalité entre le premier et le deuxième élément.

```
Prelude> 'a' <= 'a'
 True
```

Attention les caractères sont comparés dans l'ordre UTF-8.

Les tests de comparaison

(>) :: a → a → Bool

Test la supériorité entre le premier et le deuxième élément.

```
Prelude> "Jean" > "Jeanne"
 False
```

(≥) :: a → a → Bool

Test la supériorité ou l'égalité entre le premier et le deuxième élément.

```
Prelude> 10 >= sqrt (100)
 True
```

Attention les caractères sont comparés dans l'ordre UTF-8.

Mathématiques

Logarithmes

log :: Floating a ⇒ a → a

Fonction logarithme Népérien.

```
Prelude> log 10
 2.302585092994046
Prelude> log (exp 1)
 1.0
```

logBase :: Floating a ⇒ a → a → a

Fonction logarithme logarithme de base quelconque.

```
Prelude> logBase 10 10
 1.0
```

Exposants

exp :: Floating a ⇒ a → a

Retourne l'exponentiel de l'argument.

```
Prelude> exp 1
 2.718281828459045
```

(**) :: Floating a ⇒ a → a → a

Retourne le premier argument élevé à la puissance du second.

```
Prelude> 2.1 ** 3.2
 10.74241047739471
```

(^^) :: (Fractional a,Integral b) ⇒ a → b → a

Retourne le premier argument élevé à la puissance du second. L'exposant doit être un entier.

```
Prelude> 2.1 ^^ (-3)
 0.1079796998164345
```

Racines

```
sqrt :: Floating a => a -> a
```

La racine carrée de l'argument

```
Prelude> sqrt 9
 3.0
```

Pi

pi :: Floating a ⇒ a

Le nombre π

```
Prelude> pi
 3.141592653589793
```

Fonctions trigonométriques

cos :: Floating a ⇒ a → a

La fonction cosinus (Radians).

```
Prelude> cos pi
 -1.0
```

sin :: Floating a ⇒ a → a

La fonction sinus (Radians).

```
Prelude> sin (pi / 2)
 1.0
```

tan :: Floating a ⇒ a → a

La fonction tangente (Radians).

```
Prelude> tan (pi / 4)
 0.9999999999999999
```

Fonctions inverses

acos :: Floating a ⇒ a → a

La fonction arc-cosinus (Radians).

```
Prelude> acos 0
 1.5707963267948966
```

asin :: Floating a ⇒ a → a

La fonction arc-sinus (Radians).

```
Prelude> asin 0
 0.0
```

atan :: Floating a ⇒ a → a

La fonction arc-tangente (Radians).

```
Prelude> atan 1
 0.7853981633974483
```

Fonctions numériques

abs :: Num a ⇒ a → a

Retourne la valeur absolue d'un nombre.

```
Prelude> abs (-5)
 5
```

signum :: Num a ⇒ a → a

Retourne le signe d'un nombre.

```
Prelude> signum (-5)
 -1
Prelude> signum (8)
 1
```

Troncatures

round :: (Integral b,RealFrac a) ⇒ a → b

Retourne une valeur entière arrondie au plus proche.

```
Prelude> round 0.49
 0
Prelude> round 0.51
 1
```

ceiling :: (Integral b,RealFrac a) ⇒ a → b

Retourne une valeur entière arrondie par défaut.

```
Prelude> floor 0.51
 0
```

floor :: (Integral b,RealFrac a) ⇒ a → b

Retourne une valeur entière arrondie par excès.

```
Prelude> ceiling 0.49
 1
```

Parité

even :: Integral a ⇒ a → Bool

Test si l'entier est pair.

```
Prelude> even 2
 True
Prelude> even 3
 False
```

odd :: Integral a ⇒ a → Bool

Test si l'entier est impair.

```
Prelude> odd 2
 False
Prelude> odd 3
 True
```

Coordonnées

Coordonnées

Jean-Luc JOULIN
jean-luc-joulin@orange.fr
www.jeanjoux.fr