Présentation de Haskell

Jean-Luc JOULIN

Version 2

18 septembre 2022

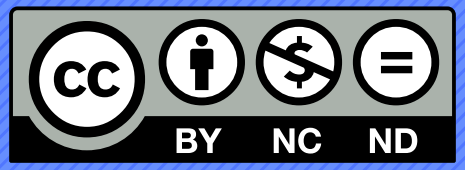
License

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 langage

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.
  - ...

Avantages de Haskell

- Découpage en fonctions facilement testables
- Code plus expressif
- Fonctions facilement composables
- Adaptés aux architectures multicoeur

Préjugés sur Haskell

- Uniquement en milieu universitaire.
- Réservés à une élite (Recherche en intelligence artificielle).
- Difficiles à apprendre.
- Très Lents.

Lenteur supposée de Haskell

Résultats des tests provenant du site benchmarksgame

	binary tree	spectralnorm	k-nucleotide	mandelbrot	nbodys
C++	0.97s	1.43s	1.96s	1.03s	4.9s
Rust	1.02s	0.71s	2.86s	1.06s	4.05s
Fortran	2.15s	0.72s	error	1.41s	4.22s
Java	2.48s	1.55s	4.92s	4.12s	6.77s
Haskell	4.04s	1.47s	24.28s	6.58s	6.53s
PHP	17.8 s	7.05s	20.54s	23.27s	1.1min
Ruby	26.46s	58.81s	1.3min	2.7min 😨	3.6min 😨
Python 3	47.8 s	1.98min	46.09s	2.96min 😨	9 min 😱

- Un peu plus lent que C, C++, Rust, Fortran.
- Beaucoup plus rapide que PHP, Ruby, Python 3.

Un langage interprété sera toujours plus lent qu'un langage compilé

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.

Utilisations concrètes

Haskell est utilisé par :

Google	Gestions de cluster et de serveurs virtuels.
Microsoft	Outils internes.
Barclays	Création et génération des produits financiers.
Bank of America	Traitement de données internes.
Facebook	Génération de code PHP et lutte contre les spams.
NVidia	Outils internes.
Selectel	Fiabilisation de la parallélisation.

...

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

```haskell
fonct x = case x of
  5 -> "Gagne"
  _ -> "Perdu"
```

Indentation correcte.

```haskell
fonct x = case x of
5 -> "Gagne"
_ -> "Perdu"
```

Indentation incorrecte.

```haskell
fonct x = case x of
  5 -> "Gagne"
   _ -> "Perdu"
```

Indentation incorrecte.

Fichiers sources Haskell

- Fichiers texte.
- Encodage UTF8 (Unicode).
- Extension .hs

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

Les commentaires

- Commentaires sur une ligne :

```haskell
maFonction a b c = d
  where
        d = (a + b + 1) ^ c  -- Un petit calcul
```

- Commentaires sur plusieurs lignes : {- ,-}

```haskell
{-
Ancienne version de maFonction
maFonction a b c = d
  where
        d = (a + b) ^ c

-}
maFonction a b c = d
  where
        d = (a + b + 1) ^ c
```

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 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.

Les types de base

Bool	Un booléen pouvant valoir True ou False.
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.

Format des nombres

- Nombres en base décimal

```
Prelude> 511
511
```

- Nombres en héxadécimal : 0x ou 0X

```
Prelude> 0x1Ff
511
```

- Nombres en octal : 0o ou 0O

```
Prelude> 0o777
511
```

- Notation scientifique :

```
Prelude> 76.569865139e21
7.6569865139e22
```

Chaînes de caractères

- Les caractères sont encadrés par des apostrophes : 'a', 'R'.
- Les chaînes de caractères sont encadrées par des guillemets : "Ma Chaine".
- Équivalent à une liste de caractères : "Ma chaine" :: [Char].
- Caractères spéciaux :

'\'	Caractère d'échapement.
'\\'	Le caractère \.
'\n'	Retour à la ligne.
'\t'	Tabulation horizontale.
'\''	Le caractère apostrophe.
'\"'	Le caractère guillemet.
'\123'	Le caractère de code ASCII 123.

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
```

Les 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
```

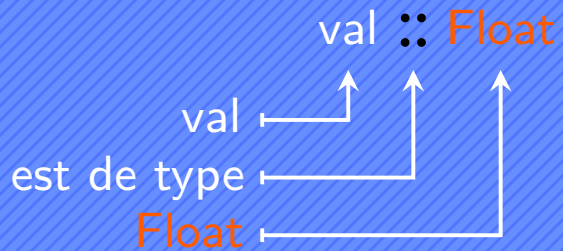
Nouveaux types de données (newtype)

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

```
newtype NumPoint = NumPoint Int
```

Signature d'une fonction

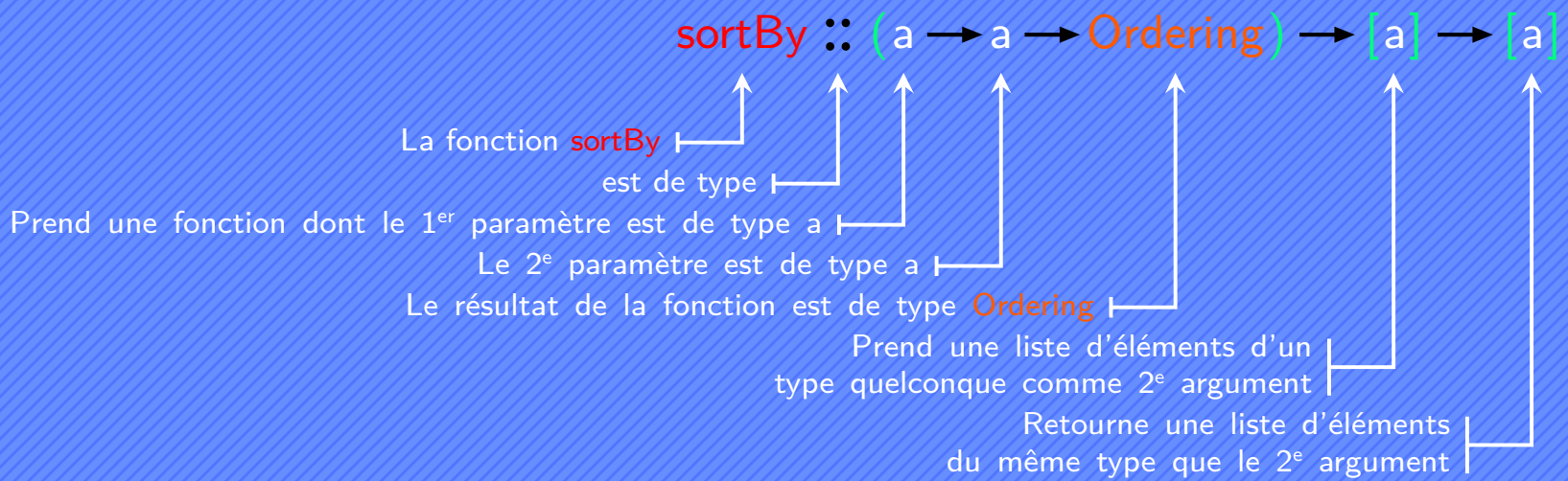
- :: signifie **"est du type"**.
- → signifie **"fonction"** avec :
  - à gauche : le paramètre.
  - à droite : ce qu'elle retourne.



Indiquer la signature dans le fichier source n'est pas obligatoire (inférence de type).

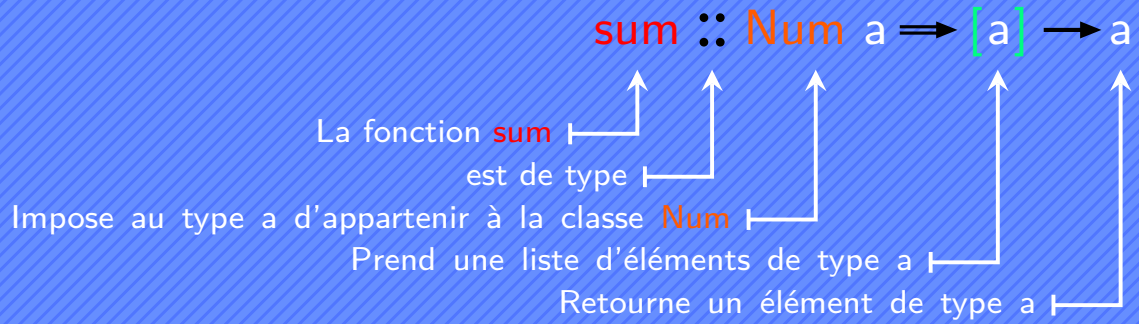
Fonction comme paramètre

- Une fonction peut prendre une autre fonction comme paramètre.



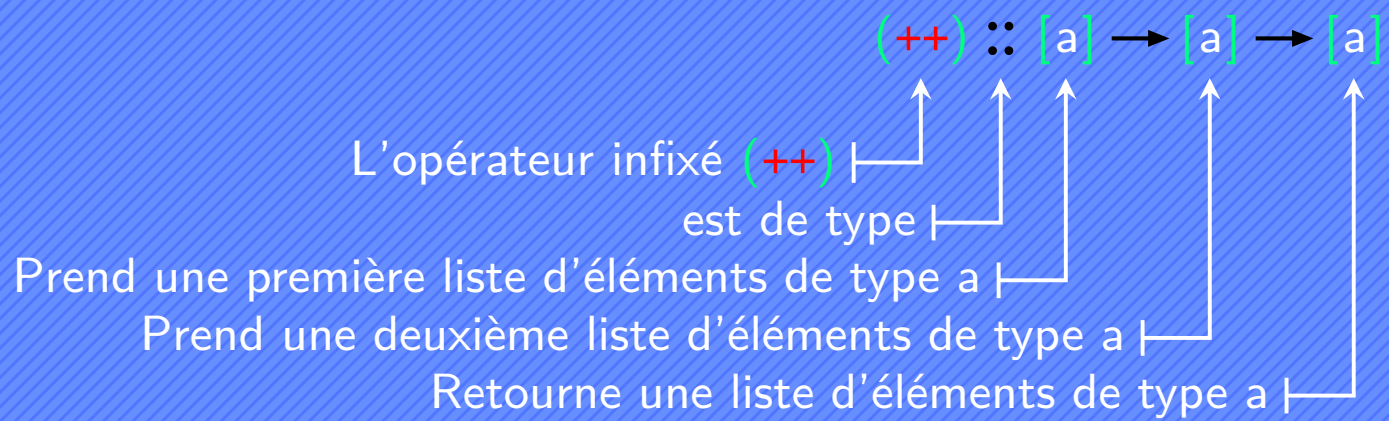
Contrainte de classe

- $\Rightarrow$ 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.



Fonctions infixés

- Fonction nativement préfixés.
- Possibilité d'utiliser une notation infixée.
- Encadrer la fonction avec des accents ‘ ‘

```
Prelude> elem 1 [4,9,8,1,3]
True
Prelude> 1 `elem` [4,9,8,1,3]
True
```

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 →.
- Permet d'exécuter des expressions en fonction de la valeur d'une variable ou d'un motif.

```
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.
- Les gardes sont évalués l'ordre.
- Sont indiqués par des barres verticales : |

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

Filtrage par motifs (par types)

Filtrage par motifs = Pattern matching

- Simplifie la définition de fonctions.
- Clarifie le code.
- Motifs sur des valeurs.

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

- Motifs sur des types.

```
fonc Nothing  = "Valeur indefinie"
fonc (Just 5) = "Valeur correcte"
fonc (Just a) = "Valeur incorrecte : "++show a
```

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

Filtrage par motifs (constructeurs de listes)

Filtrage par motifs = Pattern matching

- Motifs sur des listes.
- Premier élément et reste de la liste

```
head (x : _ ) =  x
tail (_ : xs) =  xs
```

- Liste vide.

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

- Liste avec 1, 2, 3 éléments.

```
fonc (x1 : []) = x1
fonc (x1 : x2 : []) = x2
fonc (x1 : x2 : x3 : []) = x2 + x3
```

⚠ Si un motif n'est pas défini 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

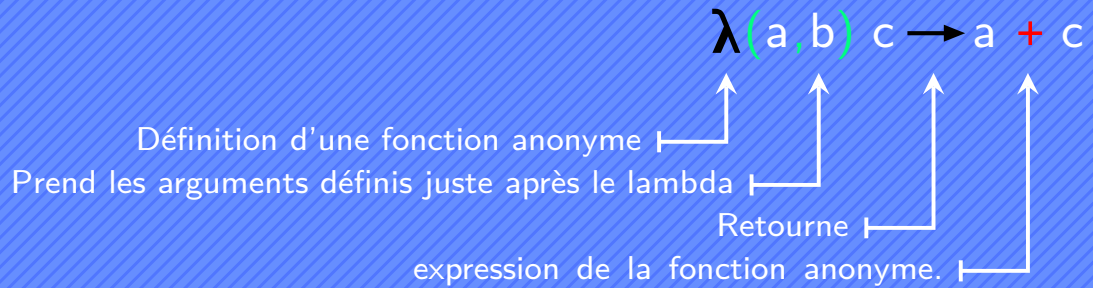
- **where** permet de définir des variables visibles dans toute une fonction.

```
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

- Déclaration d'un module avec la structure :
  module Perso.Monmodule **where**

  ```
  myFunction1 = ...

  myFunction2 = ...
  ```

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

```
src+/Perso+—/Monmodule.hs          (Perso.Monmodule)
          +—/Monautremodule.hs     (Perso.Monautremodule)
   +/Work+—/Module1.hs             (Work.Module1)
         +—/Module2.hs             (Work.Module2)
```

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 classes

Classes

- Classes de fonctions.
- Permet de regrouper des fonctions communes.

Classes standards

Eq	Classe supportant les tests d'égalité.
Ord	Classe contenant les tests de comparaison.
Show	Classe pouvant être représentée par une chaîne de caractères.
Read	Classe pouvant convertir une chaîne de caractère en type.
Enum	Classe contenant des énumérations.
Num	Classe contenant les types numériques.
Floating	Classe contenant les nombres à virgule flottante.
Integral	Classe contenant les nombres entiers.

La classe Eq

- Fonctions pour tester l'égalité entre des éléments.
- Définit les opérateurs `(==)` et `(/=)`.
- Peut être étendue à tous les types avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition :
  - Soit à `(==)`.
  - Soit à `(/=)`.

La classe Show

- Fonctions pour convertir un élément d'un type particulier en chaîne de caractères.
- Définit les fonctions :
  - show Permet de convertir un élément en String.
  - showList Permet de convertir une liste d'éléments en String.
- Peut être étendue à tous les types avec les instances ou la dérivation.
- Pour être étendue avec un instance, il est nécessaire de donner une définition :
  - Soit à show.
  - Soit à showsPrec.

La classe Num

- Fonctions pour effectuer des opérations de base sur les nombres.
- Définit les opérateurs standards sur les nombres : $(+)$, $(-)$, $(*)$.
- Définit les fonctions negate, abs, signum, fromInteger.
- Peut être étendue à tous les types avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition à $(+)$, $(*)$, abs, signum, fromInteger et :
  - Soit à $(-)$.
  - Soit à negate.

Pour l'opérateur de division $(/)$, voir la classe Fractional.

La classe Fractional

- Permet de convertir les nombres réels en nombre rationnels.
- Définit l'opérateur (/).
- Définit les fonctions recip et fromRational.
- Peut être étendue à tous les types appartenant à la classe classe Num avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition à fromRational et :
  - Soit à (/).
  - Soit à recip.

La classe Floating

- Permet d'effectuer des opérations sur les nombres à virgule flottantes.
- Définit les fonctions :
  - Nombre pi.
  - Exponentielles exp, (**).
  - Logarithmiques log, logBase.
  - Racines sqrt.
  - Trigonométriques sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh.
- Peut être étendue à tous les types appartenant à la classe Fractional avec les instances ou la dérivation.
- Pour être étendue avec une instance, il est nécessaire de donner une définition à pi, exp, log, sin, cos, asin, acos, atan, sinh, cosh, asinh, acosh et à atanh.

Création de classe

- Définir le nom de la classe.
- Définir le prototype des fonctions de cette classe.

```
class MaClasse a where
  test           ::  a   -> a -> Bool
  action         :: [a] -> Int
  essai1, essai2 :: [a] -> a
```

Dérivation

- Simplifie la définition de nouveaux types.
- Permet d'inclure de nouveaux types dans les classes courantes.
- Inclure **deriving** dans la définition du type.
- Fonctionne avec les types créés avec **data** et **newtype**.
- Fonctionne pour les classes : Eq, Ord, Show, Show

```
data Forme = Cercle    (Double,Double) Double
           | Rectangle (Double,Double) Double Double
           deriving (Eq,Ord,Show)
```

Instances de classes

- Permet d'ajouter des nouveaux types à une classe.
- Permet de définir des fonctions propres à chaque type.
- Applicable aux classes standards comme aux nouvelles classes.

```
instance Show Forme where
  show (Cercle (x,y) r)      = "Le cercle de centre "++show (x,y)++" a un rayon de "+
       +show r
  show (Rectangle (x,y) l h) = "Le rectangle de centre "++show (x,y)++" a une largeur
        de "++show l

instance Ord Forme where
  compare (Cercle (x1,y1) r1)       (Cercle (x2,y2) r2)       = compare r1 r2
  compare (Rectangle (x1,y1) l1 h1) (Rectangle (x2,y2) l2 h2) = compare (l1*h1) (l2*
       h2)
```

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 1

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 2

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.

Inversion et concatenation d'une liste

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]
```

Intégration d'éléments dans des listes

intersperse :: a → [a] → [a]

Intercale un élément entre chaque élément d'une liste du même type.

```
Prelude> intersperse 0 [1,2,3,4,5]
[1,0,2,0,3,0,4,0,5]
Prelude> intersperse '-' "abcdef"
"a-b-c-d-e-f"
```

intercalate :: [a] → [[a]] → [a]

Intercale une liste d'éléments entre chaque liste dans une liste de liste d'éléments du même type.

```
Prelude> intercalate [0,1] [[2,3],[4,5],[6,7]]
[2,3,0,1,4,5,0,1,6,7]
Prelude> intercalate ":-:" ["string1","string2","string3"]
"string1:-:string2:-:string3"
```

Extraction et découpage d'une liste

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 1

find :: (a → Bool) → [a] → Maybe a

Renvoi le premier élément conforme au prédicat (si il éxiste).

```
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 2

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)])
```

Insertion dans une liste

insert :: Ord a ⇒ a → [a] → [a]

Insére un élément dans une liste de façon a conserver l'ordre de la liste.

```
Prelude> insert ("Thierry",16) [("Marc",16),("Jean",18),("Pierre",21)]
 [("Marc",16),("Jean",18),("Pierre",21),("Thierry",16)]
```

insertBy :: (a → a → Ordering) → a → [a] → [a]

Insère un élément dans une liste en précisant la fonction de comparaison avec les autres éléments de la liste.

```
Prelude> insertBy (\(n1,a1) (n2,a2) -> compare a1 a2 ) ("Thierry",16) [("Marc",16),("Jean",18),("
      Pierre",21)]
 [("Thierry",16),("Marc",16),("Jean",18),("Pierre",21)]
```

Supression dans une liste

delete :: Eq a ⇒ a → [a] → [a]

Supprime le premier élément d'une liste correspondant à l'élément.

```
Prelude> delete ("Thierry",16) [("Marc",16),("Jean",18),("Pierre",21),("Thierry",16)]
[("Marc",16),("Jean",18),("Pierre",21)]
```

deleteBy :: (a → a → Bool) → a → [a] → [a]

Supprime le premier élément d'une liste en précisant la fonction de comparaison avec les autres éléments de la liste.

```
Prelude> deleteBy (\(n1,a1) (n2,a2) ->  a1 == a2 ) ("Olivier",16) [("Thierry",16),("Marc",16),("
      Jean",18),("Pierre",21)]
[("Marc",16),("Jean",18),("Pierre",21)]
```

Tri d'une liste

sort :: Ord a ⇒ [a] → [a]

Tri une liste dans l'ordre croissant.

```
Prelude> sort [6,4,3,12,9,7,10,4,8]
[3,4,4,6,7,8,9,10,12]
Prelude> sort [("Marc",21),("Jean",18),("Pierre",21),("Marc",16)]
[("Jean",18),("Marc",16),("Marc",21),("Pierre",21)]
```

sortBy :: (a → a → Ordering) → [a] → [a]

Tri une liste dans l'ordre croissant en précisant la fonction de comparaison avec les autres éléments de la liste.

```
Prelude> sortBy (\(n1,a1) (n2,a2) ->  compare a1 a2 )   [("Marc",21),("Jean",18),("Pierre",21),("
      Marc",16)]
[("Marc",16),("Jean",18),("Marc",21),("Pierre",21)]
```

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, ...
```

Les suites ne sont utilisables qu'avec des types de la class Enum

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, ...
```

Listes en compréhension

- Liste dont le contenu est obtenu par filtrage d'une autre liste.

```
[ x | x <- [0 ..], x^2 > 3]
```

- Plus lisible.
- Plus concis.
- Equivalent à :

```
filter (λ x -> x^2 >3) [0 ..]
```

- Proche de la notation mathématique.

$$\{x \mid x \in \mathbb{N}, x^2 > 3\}$$

Les Maps

- Famille de fonctions d'ordre supérieur.
- Appliquent une fonction sur l'ensemble des éléments d'une structure de données.
- Fonction map pure :
- Fonction dérivées :
  - Fonction concatMap :
  - Fonction mapMaybe :

Les Maps monadiques

- Famille de fonctions d'ordre supérieur.
- Appliquent une fonction monadique sur l'ensemble des éléments d'une structure de données.
- Version monadique mapM avec résultat :
- Version monadique mapM_ sans résultat :

Les Folds

- Famille de fonctions d'ordre supérieur.
- Combinent une fonction est une structure de données.
- Application itérée de la fonction sur la structure de données.
- Différentes associativité.
  - Associativité à droite.
  - Associativité à gauche.
- Différentes versions.
  - Stricte.
  - Retardé.

Folds avec associativité à gauche

- Fold avec associativité à gauche.
- Plus couramment utilisé.

```haskell
foldl (+) 7 [1,2,4,6]
```

```
Prelude> foldl (+) 7 [1,2,4,6]
20
Prelude> foldl (-) 7 [1,2,4,6]
-6
```

- Existe en version sans élément de départ.
- Existe en version stricte.

Folds avec associativité à droite

- Fold avec associativité à droite.
- Moins courant.

```
foldr (+) 7 [1,2,4,6]
```

```
Prelude> foldr (+) 7 [1,2,4,6]
20
Prelude> foldr (-) 7 [1,2,4,6]
4
```

- Existe en version sans élément de départ.
- Pas de versions strictes.

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
  - ...

Construction 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 monades

Qu'est qu'une monade ?

- Fonction particulière.
- Fonction impure.
- Traitement séquentiel des instructions.
- Autorise les effets de bords.
- Autorise les changements d'états.

Intérêt d'une monade

- Permet de recevoir des données de l'extérieur.
  - Lecture de fichiers.
  - Frappe aux clavier.
  - Gestions d'exceptions.
- Permet de transmettre des données vers l'extérieur.
  - Écriture de fichiers.
  - Affichage à l'écran.

La fonction main

- Première fonction à créer pour obtenir un exécutable.
- Type `main :: IO ()`.
- Permet de faire interagir le programme avec l'extérieur.
- Traitement séquentiel des événements.
- Exception aux fonctions pures.

Les entrées sorties

Entrées et sorties

- Les entrées et sorties sont possibles seulement au sein d'une monade.
- Possibles entre autres dans la fonction principale main.
- Le type des monades de sorties est : IO ()
- Le type des monades d'entrée est : IO a

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>:17:11: error:
      Couldn't match type `Int` with `[Char]`
      Expected type: String
        Actual type: Int
```

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> fichier <- readFile "Un autre fichier.txt"
Prelude> fichier
"Un autre fichier texte\nEssai de lecture\n"
Prelude> 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.

doesFileExist :: FilePath → IO Bool

Test l'existence du fichier à l'adresse FilePath.

```
Prelude> doesFileExist "~/.bashrc"
False
Prelude> doesFileExist "/etc/fstab"
True
```

Écriture de fichiers textes

writeFile :: FilePath → String → IO ()

Écriture d'une chaîne de caractère et remplacement d'un fichier à l'adresse définie par FilePath.

```
ghci> writeFile "monfichier.txt" "BlaBlaBla"
ghci> writeFile "monfichier.txt" "TraLaLa"
ghci> txt <- readFile "monfichier.txt"
ghci> txt
"TraLaLa"
```

appendFile :: FilePath → String → IO ()

Ajout d'une chaîne de caractère à un fichier à l'adresse définie par FilePath.

```
ghci> writeFile "monfichier.txt" "BlaBlaBla"
ghci> appendFile "monfichier.txt" "TraLaLa"
ghci> txt <- readFile "monfichier.txt"
ghci> txt
"BlaBlaBlaTraLaLa"
```

Les 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 sur une liste

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
```

Prédicat sur une liste

any :: (a → Bool) → [a] → Bool

Applique un prédicat sur une liste et renvoit True si au moins un élément de la liste valide ce prédicat.

```
Prelude> any even [1,2,3,5,9]
True
```

all :: (a → Bool) → [a] → Bool

Applique un prédicat sur une liste et renvoit True si tous les éléments de la liste valident ce prédicat.

```
Prelude> all even [1,2,3,5,9]
False
```

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 opérateurs d'égalité

(==) :: 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 fonctions 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 opérateurs de comparaison 1

(<) :: 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 opérateurs de comparaison 2

(>) :: 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

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
```

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
```

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
```

pi :: Floating a ⇒ a

Le nombre π

```
Prelude> pi
3.141592653589793
```

Fonctions trigonométriques 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
```

Types utiles

Le type Maybe

- Le type Maybe a est utilisé pour représenter une donnée qui peut valoir :

  Nothing Le type ne contient pas de valeur (valeur indéfinie).

  Just a Le type contient un élément de type a (valeur définie).

- N'importe quel type a peut être utilisé : Maybe Bool, Maybe Int, Maybe String, ...
- Le type Maybe est accessible après avoir importé le module Data.Maybe.

Application du type Maybe

- Une des application du type Maybe a est la recherche dans une liste.

$$\text{find} :: (a \to \text{Bool}) \to [a] \to \text{Maybe } a$$

- L'élément à chercher peut être présent ou pas.
- L'élément à chercher doit être retournée.

Nothing	L'élément est introuvable dans la liste
Just a	L'élément éxiste et est retourné comme résultat.

Les fonctions de test

isNothing :: Maybe a → Bool

Renvoie True si la donnée ne contient rien.

```
Prelude> isNothing Nothing
True
```

isJust :: Maybe a → Bool

Renvoie True si la donnée contient quelque chose.

```
Prelude> isJust (Just "a")
True
Prelude> isJust Nothing
False
```

Les fonctions d'évaluation

fromMaybe :: a → Maybe a → a

Prend une valeur par défaut et une valeur de type Maybe.
Si la valeur Maybe vaut Nothing, la valeur par défaut est renvoyée.
Si la valeur Maybe vaut Just val, val est renvoyé

```
Prelude> fromMaybe 18 Nothing
18
Prelude> fromMaybe 18 (Just 5)
5
```

catMaybes :: [Maybe a] → [a]

Prend une liste de Maybe et retourne la liste des valeurs Just

```
Prelude> catMaybes [Nothing,Just 5, Nothing, Just 4]
[5,4]
```

Coordonnées

Coordonnées

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