Introduction à Haskell

Doc. Malik Koné

11 février 2023

Table des matières

1 Qu'est-ce que Haskell **3**
- 1.1 Préambule . . . 3
- 1.2 Fonctions . . . 3
- 1.3 Programmation fonctionnelle . . . 3
- 1.4 Caractéristiques d'Haskell . . . 4
- 1.5 Un peu d'histoire . . . 4
- 1.6 Exemples de programmes Haskell . . . 4

2 De quoi avons-nous besoin pour commencer ? **5**
- 2.1 GHCi : Glasgow Haskell Compilater Interactif . . . 5
- 2.2 Prélude . . . 6
- 2.3 Comment appeler une fonction . . . 8
- 2.4 Scripts Haskell . . . 8
- 2.5 Règle de nommage des fonctions et des variables . . . 10
- 2.6 Exercices . . . 11

3 Types et Classes de type **12**
- 3.1 Types de base . . . 12
- 3.2 Types liste . . . 15
- 3.3 Types tuple . . . 16
- 3.4 Types fonction . . . 16
- 3.5 Fonction curryfiées (*Curried functions*) . . . 18
- 3.6 Type Polymorphique . . . 18
- 3.7 Types surchargés, Overloaded types . . . 20
- 3.8 Classes de base . . . 20
- 3.9 Show – les types que l'on peut afficher . . . 21
- 3.10 Num – les types numériques . . . 21
- 3.11 Integral . . . 22

3.12 Fractional . . . 22
3.13 Exercices . . . 22

4 Déclaration de fonctions **23**
4.1 En utilisant les fonctions existantes . . . 23
4.2 Expressions conditionnelles . . . 24
4.3 Gardes . . . 24
4.4 Motifs (*pattern matching*) . . . 24
4.5 Expressions lambda . . . 26
4.6 Opérateurs . . . 27
4.7 Exerices . . . 28

5 Listes en compréhension **29**
5.1 Filtres *Guards* . . . 29
5.2 la fonction zip . . . 30
5.3 Exercices . . . 30

6 Fonctions récursives **31**
6.1 Concepts de bases . . . 31
6.2 Récursion et listes . . . 31
6.3 Récursions multiples . . . 32
6.4 Récursion mutelle . . . 33
6.5 Conseils pour construire des fonctions récursives. . . . 33
6.6 Exerices . . . 34

7 Fonctions de haut-niveau (higher-order) **35**
7.1 Exemples de base . . . 35
7.2 fold . . . 36
7.3 Composition . . . 36
7.4 Exercices . . . 36

8 Déclaration des Types et des Classes **38**
8.1 Comment déclarer de nouveau type . . . 38
8.2 Classes de type . . . 40
8.3 Exercices . . . 40

9 Annexes **40**

1 Qu'est-ce que Haskell

1.1 Préambule

Ce document n'aurait pas été possible sans le livre de Graham Hutton "Programming in Haskell (Cambridge, 2018)" ainsi que du cours de futur-Learn "progaming in Haskell" et des notes du hackton Haskell et Plutus de WADA. Merci

Haskell est un langage programmation avancé, fonctionnel. Les fonctions y jouent un rôle très important. On dit qu'elles sont de première classe. En Haskell, la notion de type est centrale. Un peu comme en mathématique toute fonction a un domaine de définition et un ensemble image. Ils doivent être précisés avant de coder.

Nous donnons un aperçu des fonctions, puis nous introduisons la notion de programmation fonctionnelle et nous présenterons les atouts majeurs de Haskell.

1.2 Fonctions

Une fonction Haskell est une relation entre un ou plusieurs paramètres (*argument*) et un résultat. On la définit à l'aide d'une équation, comme suit~ :

```
double x = x + x
```

Ici, une fonction nommée 'double' a un paramètre x et renvoie le résultat $x + x$.

1.3 Programmation fonctionnelle

La programmation fonctionnelle est un style de programmation qui favorise l'utilisation des fonctions. Haskell est un langage fonctionnel parce les fonctions y jouent un rôle de premier plan.

Java, C++, Python sont des langages *impératifs* parce qu'on y définit **comment faire** les calculs et non **quels résultats** avoir. Voici par exemple comment on programme la somme des entiers de 1 à n en style impératif,

```
int total = 0;
for (int count = 1; count <= 1; count ++)
    total = total + count;
```

on décrit ci-dessus comment faire le calcul.

Voici un exemple du même programme avec un style fonctionnel.

```
sum [1..n]
```

Ici, on définit ce que l'on veut comme résultat. Pour cela, on fait appel à deux fonctions '[1..n]' qui renvoie la liste des entiers de 1 à n et ensuite à la fonction 'sum'.

1.4 Caractéristiques d'Haskell

Voici quelqu'unes des caractéristiques du langage Haskell. Il permet :

- les listes en compréhension
- les fonctions récursives, c'est-à-dire des fonctions qui s'appellent elle-même.
- des programmes concis
- les fonctions de haut niveau (*higer-order functions*), des fonctions qui utilisent d'autre fonction comme paramètre ou résultat. c'est-à-dire qu'une fonction peut s'appliquer à une autre fonction.
- un typage évolué des entrées et sorties des fonctions
- une évaluation paresseuse (*lazy evalutation*), seuls les résultats utilisés sont calculés.

1.5 Un peu d'histoire

Haskell découle du *lambda calculus* développé dans les années 30 par Alonzo Church. Il est aussi inspiré du Lisp (années 50), premier langage à base de liste. Depuis 1987, il existe un comité pour le développement d'Haskell rassemblant des informaticiens de renom. Au moins trois récipiendaires du prix ACM Turing Award ont participé au développement de Haskell. Le langage est nomé en honneur à Haskell Curry un logisticien Britanique.

1.6 Exemples de programmes Haskell

1.6.1 Faire la somme d'une liste de nombre

On peut définir la fonction 'somme' comme suit :

```
somme [] = 0
somme (n:ns) = n + somme ns
```

et l'appeller ainsi :

```
somme [1,2,3]
```

En Haskell, toutes les fonctions ont un type. Celui de 'somme' est :

```
somme :: Num a => [a] -> a
```

Cela veut dire que pour une liste d'éléments de type 'a' de la classe de type 'Num', la fonction 'somme' renvoie un seul élément de type 'a'. Il y a de nombreux types en Haskell et plusieurs types de nombres (ie. de la classe 'Num').

Les types Haskell donnent des informations sur les fonctions. Ils permettent d'éviter beaucoup d'erreurs avant même l'exécution du programme.

1.6.2 Trier

Voici un autre exemple Haskell pour un programme de tri

```
tri_vite [] = []
tri_vite (x:xs) = tri_vite plus_petits ++  [x] ++ tri_vite plus_grands
                where
                  plus_petits = [a | a <- xs, a <= x]
                  plus_grands =  [b | b <- xs, b > x]
```

Nous reviendrons sur cet exemple plus tard.

2 De quoi avons-nous besoin pour commencer ?

2.1 GHCi : Glasgow Haskell Compilater Interactif

Comme pour tout programme, il nous faut un éditeur de texte. Celui avec lequel vous êtes le plus confortable est souvent le meilleur. Ensuite il faut un compilateur Haskell. Lorsque vous programmerez vous aurez donc deux fenêtres ouvertes. L'une avec l'éditeur, l'autre avec le compilateur.

Le compilateur que nous allons installer est le plus commun. Il s'appelle Glasgow Haskell Compilater (GHC). Installez-le en suivant en suivant les instructions du lien.

La version interactive du compilateur est pratique pour apprendre et c'est ce que nous utiliserons pour notre cours. Sachez toute fois que l'on peut compiler un programme de la façon suivante :

```
ghc -O2 nom_du_programme.hs
```

ou encore

```
ghc --make mon nom_du_programme.hs
```

Une fois ghci installé, tester le. Vous devriez voir quelque chose comme cela :

```
mlk@teur ~ $ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude>
```

À gauche, '>' est l'invite de commande. Vous voyez écrit aussi 'Prelude'. C'est une bibliothèque de fonction chargé par défaut dans le compilateur.

Le système attend votre commande. Essayer les commandes suivantes :

```
> 2+3*4

> (2+3)*4

> sqrt (3^2 + 4^2)
```

2.2 Prélude

La bibliothèque de départ d'Haskell contient un nombre important de fonctions, notamment des fonctions qui s'appliquent à des listes. Les éléments des listes sont notés entre crochets, exemple

```
[1,2,3,4,5]
```

Nous donnons ici les définitions et des exemples des fonctions les plus souvent utilisés.

2.2.1 head

Sélectionne le premier élément d'une liste non vide

```
head  [1,2,3,4,5]
```

2.2.2 tail

Enlève le premier élément d'une liste mon vide.

```
tail [1,2,3,4,5]
```

2.2.3 !! (index)

Sélectionne le n^{ime} élément d'une liste non vide. Notez que le premier indice est 0.

```
[1,2,3,4,5]  !! 2
```

2.2.4 take

Sélectionne les n premiers éléments d'une liste.

```
take 3 [1,2,3,4,5]
```

2.2.5 drop

Supprime les n premiers éléments d'une liste.

```
drop 3 [1,2,3,4,5]
```

2.2.6 length

Calcule la longueur d'une liste.

```
length [1,2,3,4,5]
```

2.2.7 sum

Fait la somme d'une liste de nombre.

```
sum [1,2,3,4,5]
```

2.2.8 product

Calcule le produit des éléments d'une liste.

```
product [1,2,3,4,5]
```

2.2.9 ++ (concatène)

Concatène (ou rassemble) deux listes en une seule.

```
[1,2,3] ++ [4,5]
```

2.2.10 reverse

Renverse les éléments d'une liste.

```
reverse [1,2,3,4,5]
```

2.3 Comment appeler une fonction

Les paramètres d'une fonction sont simplement séparés par des espaces. Les parenthèses précisent la priorité des opérations sachant que l'application d'une fonction est prioritaire sur les opérations standards mathématiques, ainsi :

```
f a + b
```

veut dire $f(a) + b$ et non $f(a + b)$, et

```
f a b + c*d
```

veut dire $f(a, b) + cd$ et non $f(a, b + cd)$.

Voici une table associant notation mathématique et notation Haskell

notation mathématique	notation Haskell
$f(x)$	f x
$f(x, y)$	f x y
$f(g(x))$	f (g x)
$f(x, g(y))$	f x (g y)
$f(x)g(y)$	f x * g y

2.4 Scripts Haskell

Les scripts Haskell ont généralement pour extensions '.hs'. Supposons que nous ayons le code suivant dans un fichier appelé 'test.hs'

```
double x = x + x


quadruple x = double (double x)
```

Ce programme définit deux fonctions mai il n'affiche rien, et il ne demande rien à l'utilisateur non plus. Pour tester les fonctions, il est pratique de l'ouvrir dans ghci comme suit :

```
ghci test.hs
```

Cela va charger en mémoire les fonctions et nous pourront les appeler à l'intérieur de ghci

```
> quadruple 10
40
> take (double 2) [1,2,3,4,5,6,7,8]
[1,2,3,4]
```

Noter que lorsque nous effectuons une modification dans le fichier 'test.hs', 'ghci' ne prend pas en compte automatiquement les modifications. Il faut utiliser la commande ' :reload' (avec les :).

Par exemple, modifions 'test.hs' en lui ajoutant les deux fonctions suivantes :

```
factorielle n = product [1..n]

moyenne ns = sum ns  `div` length ns
```

Puis dans ghci, il faut écrire avant de pouvoir tester les nouvelles fonctions.

```
> :reload
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, one module loaded.

> factorielle 10
3628800
```

Des commandes importantes de ghci à connaître sont :

- : ? ou :help, pour afficher l'aide de ghci ;
- :browse *module*, pour montrer toutes les définitions chargées par le module ;
- :doc *fonction*, pour avoir la documenation d'une fonction ;
- :i ou :info *objet*, pour avoir des informations sur un object ;
- :l ou :load *fichier*, pour charger un module ;
- :q ou :quit, pour sortir du ghci ;
- :r ou :reload *fichier*, pour recharger un module qui vien d'être modifié ;
- :t ou :type *objet*, pour connaître le type d'un objet.

Par exemple :

```
> :doc head
```

renvoie

```
Extract the first element of a list, which must be non-empty.
```

2.5 Règle de nommage des fonctions et des variables

Le nom des fonctions et de leurs paramètres doivent commencer avec lettre une minuscule et peuvent ensuite contenir 0 ou plusieurs caractères suivant 'a-zA-Z0-9'_' .

Comme dans tous langages de programmation, certains mots-clefs ont des significations spéciales et ils ne peuvent pas désigner des variables ou des fonctions. En Haskell ces mots sont les suivants :

```
case class data default deriving do else foreing if import in infix infixl infixr inst
```

2.5.1 Convention pour les noms de variables

Par convention, on appelle 'n', 'x', et 'cs' les variables qui contient respectivement des nombres, des valeurs quelconques et des caractères. On ajoute 's' aux noms des variables pour désigner une liste ou un ensemble. Par exemple, on écrira 'ns' pour une liste de nombres, ou 'xs' pour une liste de variables quelconques et 'css' pour une liste de caractères.

2.5.2 Disposition du code

Comme dans python, Haskell utilise des espaces pour grouper les expressions ensemble. Dans le code ci-dessous,

```
a = b + c
    where
      b = 1
      c = 2

d = a * 2
```

'b' et 'c' sont des variables définies localement dans la fonction 'a'.

On peut rajouter des accolades et séparer les définitions avec des ;.

```
a = b + c
    where
      {b = 1;
       c = 2};

d = a * 2;
```

Ce qui permet dde réécrire le code sur une seule ligne.

```
a=b+c where {b=1; c=2}; d=a*2;
```

1. Indentation Comme les éditeurs de texte interprètent différemment les tabulations, il est préférable d'utiliser des espaces plutôt que des tabulations, pour l'indentation. L'idéal et que son éditeur transforme automatiquement les tabulations en espaces.

   Par convention on utilise 8 espaces pour indenter le code.

2. Commentaires

   (a) Simples

   Les commentaires simples commencent avec le symbole '–'. Ils doivent tenir sur une ligne, comme ci-dessous :

   ```
   -- Factorielle des n premiers entiers
   factorielle n = product [1..n]
   ```

   (b) Imbriqués Les commentaires imbriqués sont encadrés par '{-' et '-}'. Ils peuvent s'étendre sur plusieurs lignes et contenir des commentaires. Ils sont pratiques pour commenter des blocs de code entiers, comme ci-dessous :

   ```
   {-  3 fonctions du fichier test.hs commentées

   double x = x + x

   quadruple x = double (double x)

   -- Factorielle des n premiers entiers
   factorielle n = product [1..n]
   -}

   moyenne ns = sum ns  `div` length ns
   ```

   Voyons maintenant plus en détail ce qui fait l'une des forces de Haskell, son système de typage.

2.6 Exercices

2.6.1 Exercice n°1

Mettez les parenthèses sur les expressions numériques suivantes $2^3 * 4$, $2 * 3 + 4 * 5$ et $2 + 3 * 4^5$

2.6.2 Exercice n°2

Corrigez les trois erreurs de syntaxe dans le script ci-dessous et vérifiez qu'il fonctionne dans 'ghci'

```
N = a 'div' length xs
    where
      a = 10
     xs = [1,2,3,4,5]
```

Écrire ce script en une seule ligne.

2.6.3 Exercice n°3

La fonction 'init' renvoie la dernière partie d'une liste par exemple

```
init [1,2,3,4,5]
```

```
[1,2,3,4]
```

Définissez cette fonction de deux façons différentes avec les fonctions vu dans ce chapitre

3 Types et Classes de type

Commençons par définir les types de base, puis ceux des listes et des fonctions. Nous finirons en parlant dans classe de types.

On utilisera la commande ' :t' dans ghci pour voir les types des objets.

3.1 Types de base

Les types suivant sont définis par défaut dans *Prelude*.

3.1.1 Bool – les booléens

```
:t True
```

```
:t False
```

Lorsque l'on demande le type de 'True' ou 'False' la réponse est 'Bool'. En Haskell, tous ce qui suit les symboles ' : :' indique le type de l'objet.

Les opérateurs classiques sur les booléens sont '&&', '||' et 'not'.

3.1.2 Char – caractères

On utilise les guillemets simples pour noter un caractère.

```
:t 'a'
```

On peut aussi utiliser la valeur numérique (décimal ou hexadécimal) telle que définie par le standard unicode. Cela est utile pour saisir des caractères de contrôles, comme '\n' (nouvelle ligne) ou '\t' (tabulation).

```
:t '\97'
```

```
:t '\x61'
```

```
'\x61' :: Char
```

```
('\97' == 'a') && ('a' == '\x61')
```

```
True
```

1. Fonction sur les caractères Il existe de nombreuses fonctions sur les caractères, notamment contenue dans la bibliothèque 'Data.Char' que l'on peut charger avec

   ```
   import Data.Char
   ```

   Un exemple

   ```
   :t toUpper
   ```

   ```
   toUpper :: Char -> Char
   ```

   Ce type veut dire que 'toUpper' prend en entrée un caractère et renvoie un caractère en résultat.

   ```
   toUpper 'a'
   ```

3.1.3 String – chaînes de caractères

Les 'String' sont encadrés par des guillemets doubles. Ce sont des séquences de caractères.

```
:t "efgh"
```

```
"efgh" :: [Char]
```

3.1.4 Int – entiers à précision fixe

Les 'Int' sont des entiers, par exemple -100, 9 ou 0. Ils sont compris entre -2^{63} et $2^{63} - 1$. En dehors de cet intervalle, les résultats sont chaotiques.

```
2^63 - 1
```

```
9223372036854775807
```

9 223 372 036 854 775 807 est positif, mais si on lui ajoute 1 et que l'on force le type 'Int' on a

```
2^63 :: Int
```

```
-9223372036854775808
```

un nombre négatif. On a dépassé la borne $2^63 - 1$

3.1.5 Integer – entiers sans précision fixée

Les 'Integer' sont des entiers aussi grands que ce que permet la mémoire de l'ordinateur.

```
2^63 :: Integer
```

```
9223372036854775808
```

Cette fois ci le résultat est positif. Le désavantage des 'Integers' c'est qu'ils sont plus lents à traiter que les 'Int'.

3.1.6 Float – nombres à virugle de précision simple

Les 'Float' représentent les nombres décimaux, par exemple -12.34, 1.0 ou 3.1415927 (notez le point à la place de la virgule). Leur taille en mémoire est limité et leur précision dépend de la taille du nombre, par exemple la précision de

```
sqrt 2 :: Float
```

```
1.4142135
```

est de 7 chiffres après la virgule, alors que celle de

```
sqrt 9999 :: Float
```

```
99.995
```

est de 3 chiffre seulement.

'sqrt' est une fonction la librairie *Prelude* qui calcule la racine carrée un nombre.

3.1.7 Double – nombres à virgule avec deux fois plus de précision

Ce type est similaire à 'Float' mais avec deux fois plus de place en mémoire afin d'accroître la précision.

```
sqrt 2 :: Double
```

```
1.4142135623730951
```

Ci-dessus la précision est maintenant de 16 et ci-dessous elle est maintenant de 14.

```
sqrt 9999 :: Double
```

```
99.99499987499375
```

Notez qu'il faut toujours faire attention aux problèmes d'arrondis lorsque l'on utilise ces types de nombre pour des calculs précis.

Nous venons de voir qu'il existe plusieurs types de nombre, 'Int', 'Integer', 'Float', et 'Double'. Nous pourront les regrouper dans la classe de type 'Num' que nous détaillerons plus tard.

3.2 Types liste

Une liste est une **séquence** d'éléments de **même type**. Elle les encadre avec des crochets et les séparer par des virgules. On note $[T]$ le type de la liste dont tous les éléments sont de type T, par exemple :

```
:t  [False, True, False ]
```

```
[False, True, False ] :: [Bool]
```

```
:t  ['a', 'b', 'c', 'd']
```

```
['a', 'b', 'c', 'd'] :: [Char]
```

Un peu plus compliqué

```
:t "Zéro"
```

```
"Zéro" :: [Char]
```

On remarque qu'une chaîne de caractères est une liste de caractères.

Une liste peut aussi contenir d'autre liste.

```
:t  ["Un", "Deux", "Trois"]

["Un", "Deux", "Trois"] :: [[Char]]
```

Le nombre d'éléments d'une liste s'appelle *length*. La liste '[]' est vide, elle a une longueur nulle. Les listes qui n'ont qu'un seul élément comme '[False]' ou '['a']' sont appelés *singleton*.

Il n'y a pas de restrictions sur le type d'élément que peut contenir une liste ou sur sa taille. Cette dernière peut même être infinie.

3.3 Types tuple

Le tuple est une **séquence** finie d'éléments potentiellement **de type différent**. On le note entre les parenthèses '()' et ses éléments sont séparés par des virgules. On écrira $(T_1, T_2, \cdots, T_n)$ le type d'un tuple contenant des éléments de type $T_1, T_2 \cdots T_n$

```
:t (False, True)

(False, True) :: (Bool, Bool)

:t ('a', "Bob", True)

('a', "Bob", True) :: (Char, [Char], Bool)
```

La longueur d'un tube est appelée *arity*. Le tuple de longueur zéro '()' est le tuple vide, mais attention les singletons tuples n'existe pas car leur notation serait confondu avec l'usage mathématique des parenthèses. Les parenthèses dans '(1+3)*4' doivent être comprise pour leur signification mathématique. Elles ne désignent pas un tuple.

Évidement, les tuples peuvent contenir d'autres tuples.

```
:t (False, (True, "Alpha"), [('b','c')])
```

Il n'y a pas de limite quant au type des éléments contenus dans un tuple, mais leur taille est forcément finie.

3.4 Types fonction

Une fonction est une relation entre des paramètres d'un certain type et un résultat qui peut-être d'un autre type. On note $T_1 -> T_2$ le type de toutes les fonctions qui mettent en relation un paramètre de type T_1 avec un résultat de type T_2. Par exemple,

```
:t not

:t even
```

Notez le symbole '=>'. Il indique une contrainte de classe. le type 'a' doit appartenir à la classe de type 'Integral'. Cette classe que nous détaillerons plus tard est une classe pour les types numériques pour lesquelles les opérations 'div' et 'mod' sont définies.

'even' (est pair) est une fonction qui renvoie 'True' si le nombre est pair et 'False' sinon.

3.4.1 Fonction avec plusieurs paramètres et résultat

Si une fonction a plusieurs paramètres, on peut les passer en tant que tuple.

```
ajouter :: (Int, Int) -> Int
ajouter (x,y) = x+y
```

Par convention, en Haskell, on fait précéder la définition d'une fonction par son type. Cela sert de documentation et facilite le débogage.

Les résultats peuvent aussi être des listes, par exemple

```
zéro_à :: Int -> [Int]
zéro_à n = [0..n]
```

Notez que les fonctions ne sont pas obligatoirement définies sur tous l'ensemble de leur type (ou domaine). La fonction 'head' définie sur les listes, ne l'est pas pour la liste vide.

```
:t head

head []
```

Nous venons de voir comment définir des fonctions avec plusieurs variables, mais il existe une autre approche qui fait la force de Haskell. Ce sont les fonctions curryfiées.

3.5 Fonction curryfiées (*Curried functions*)

Les fonctions qui prennent leur argument un par un et qui renvoie des fonctions s'appellent fonctions curryfiées ou *curried function*. En Haskell, une fonction peut renvoyer une fonction comme résultat.

Comparons la fonction 'ajouter' curryfiée ci-dessous à celle non currifiée du chapitre précédent.

```
ajouter' :: Int -> (Int -> Int)
ajouter' x y = x+y
```

La fonction 'ajouter' curryfiée prend un paramètre une entrée (un 'Int') et renvoie une fonction qui transforme un 'Int' en 'Int' ('Int->Int'). Au chapitre précédent la fonction 'ajouter', non curryfié, qui prenait 2 paramètres en entrée et qui renvoyait un seul 'Int' comme résultat. Toutes les fonctions à plusieurs paramètres peuvent être curryfiées. L'avantage c'est la concision du code et sa lisibilité qui font éviter les bugs..

Voici un autre exemple de fonction curryfiée

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x*y*z
```

'mult' renvoie une fonction qui renvoie une fonction qui renvoie un 'Int'

```
mult x y z
```

veut dire

```
((mult x) y) z
```

Par convention on omettra les parenthèses au niveau des types, ainsi

```
mult :: Int -> (Int -> (Int -> Int))
```

est équivalent à

```
mult :: Int -> Int -> Int -> Int
```

3.6 Type Polymorphique

Nous avons la fonction 'head'. Elle renvoie le premier élément d'une liste quel que soit le type de cet élément.

```
head [1,2,3,4]
```

```
1
```

```
head ["Soumahoro", "Kanté"]
```

```
Soumahoro
```

```
head "Kirina"
```

```
'K'
```

C'est donc une fonction qui s'applique à plusieurs types. Cette spécificité est précisée lorsqu'on regarde son type.

```
:t head
```

```
head :: [a] -> a
```

Son type fait apparaître une **variable de type** (*type variable*) 'a'. Les variables de type sont notées avec des lettres minuscules par exemple 'a', 'b', ou 'c'.

Un type qui fait appel à une variable de type est appelé **type polymorphique**.

3.6.1 Exemples de fonctions ayant des types polymorphiques

Regardons le type de certaines fonction courante en Haskell

1. fst
   la fonction 'fst' qui extrait le premier élément d'une paire

   ```
   :t fst
   ```

   ```
   fst :: (a, b) -> a
   ```

   'fst' a un type qui contient deux variables de type. C'est naturellement un type polymorphique.

   ```
   fst ("Palmier", 4)
   ```

   ```
   Palmier
   ```

2. take 'take' renvois les n premiers éléments d'une liste

   ```
   take :: Int -> [a] -> [a]
   ```

3. zip 'zip' Rassemble deux listes de type différent en une seule liste de couple

   ```
   zip :: [a] -> [b] -> [(a, b)]
   ```

4. id la fonction 'id' identité renvoie le paramètre à l'identique

   ```
   id :: a -> a
   ```

3.7 Types surchargés, Overloaded types

Certaines fonctions s'appliquent à plusieurs types mais avec certaines contraintes. Par exemple la fonction (+) s'applique 'Int' et 'Float' mais pas aux listes. On note cette contrainte *class constraint* 'C a =>', où C est le nom d'une classe de type et 'a' une variable de type.

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

Cela veut dire que la fonction (+) s'applique à des types de la classe Num (ou numérique).

Un type qui contient les contraintes de classe est appelé un **type surchargé** ou *overloaded type*.

Voici d'autres exemples

```
negate :: Num a => a -> a
abs ::Num a => a -> a
```

3.8 Classes de base

Une *class* est un ensemble de type pour lesquelles certaines opérations (appelées méthodes) sont définies.

Voici les classes de base définies par Haskell

3.8.1 Eq – les types égalité

Cette classe contient les types qui peuvent être comparés en utilisant les méthodes suivantes

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

Tous les types de base, les listes et les tuples, sont des instances de cette classe de type.

3.8.2 Ord – les types ordonnés

Cette classe contient tous les types de la classe égalité 'Eq' et qui, en plus, supportent les méthodes 6 suivantes :

```
(<) :: a -> a -> Bool
(>) :: a -> a -> Bool
(<=) :: a -> a -> Bool
```

```
(>=) :: a -> a -> Bool
min :: a -> a -> a
max :: a -> a -> a
```

Tous les types de base sont des instances de cette classe de type. Les listes et les tuples, aussi, si les types de leurs éléments sont instance de cette classe.

```
('a', 10) < ('a', 100)
```

```
[1,2,3] > [0,20,4,32]
```

Notez que les listes et tuples sont rangées comme dans un dictionnaire, en comparant l'ordre des éléments un par un.

```
('a','b') < ('a', 10)
```

3.9 Show – les types que l'on peut afficher

Cette classe contient tous les types dont les valeurs peuvent être converties en chaîne de caractères avec la méthode suivante :

```
show :: a -> String
```

— exemples :

```
show [1,2,3]
```

```
show ('a', False)
```

3.10 Num – les types numériques

La classe de type 'Num' comprend les 'Int', 'Integer', 'Float' et 'Double'. De façon général, elle comprend tous les types qui supportent les six méthodes suivantes :

```
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a
```

Si l'on regarde le type du nombre 4, par exemple

```
:t 4
```

```
4 :: Num p => p
```

On voit qu'il doit être de la catégorie Num.
La fonction 'signum' renvoie le signe d'un nombre.

```
signum (-3)
```

```
-1
```

Notez que les parenthèses sont importantes. Si on les oublie 'signum' essayera de s'appliquer sur le symbole '-'.

3.11 Integral

Ce sont les types de la classe Num, mais qui en plus supporte les deux méthodes suivantes :

```
div :: a -> a -> a
mod :: a -> a -> a
```

Il s'agit la division euclidienne et du modulo.

3.12 Fractional

Ce sont les types numériques qui, en plus, supportent les 2 méthodes suivantes :

```
(/) :: a -> a -> a
recip :: a -> a
```

```
:doc recip
```

3.13 Exercices

3.13.1 Exerice n°1

Quels sont les types des valeurs suivantes ?

```
['a','b','c']
('a','b','c')
[(False, '0'), (True,'1')]
([False,True], ['0','1'])
[tail, init, reverse]
```

3.13.2 Exercice n°2

écrire des définitions quelconque mais correct et qui ont les types suivants.

```
bools :: [Bool]
nums :: [[Int]]
add : Int -> Int -> Int -> Int
copy :: a -> (a,a)
apply :: (a->b) -> a -> b
```

3.13.3 Exercice n°3

trouver manuellement les types des fonctions suivantes ?

```
second xs = head (tail xs)
échange (x,y) = (y,x)
pair x y = (x,y)
double x = x*2
palindrome xs = reverse xs == xs
deuxfois f x = f (fx)
```

Prenez en compte les éventuelles contraintes de type vérifiant suite ce que vous avez trouvé à l'aide GHCi

4 Déclaration de fonctions

4.1 En utilisant les fonctions existantes

La façon la plus simple de faire des fonctions et d'utiliser des fonctions qui existent déjà.

```
impair :: Integral a => a -> Bool
impair n = n `mod` 2 == 0
```

Ici, on fait appel à l'opérateur 'mod'. On verra ce qu'est un opérateur plus loin.

```
séparer :: Int -> [a] -> ([a],[a])
séparer = n xs = (take n xs, drop n xs)
```

Ici, on fait appelle à 'take' et 'drop' et ci-dessous simplement à '/'.

```
inverse :: Fractional a => a -> a
inverse n = 1/n
```

4.2 Expressions conditionnelles

```
abs' :: Int -> Int
abs' n = if n >= 0 then n else -n

signum' :: Int -> Int
signum' n = if n < 0 then
                -1
            else
                if n == 0 then
                    0
                else
                    1
```

4.3 Gardes

À la place des expressions conditionnelles, on peut utiliser des équations gardées *guarded equations*. Un garde c'est un prédicat qui peut être vrai ou faux. On le note sous forme d'équation ou d'inéquation, juste après un '|'. Une fonction en utilise généralement plusieurs. Ils permettent de choisir les instructions à éxecuter. Seule la première des expressions dont le garde est vérifiée est exécutée. Si un garde est faux, on regarde le deuxième, puis le troisième et ainsi de suite.

```
abs'' n | n >= 0 = n
        | otherwise = -n
```

'|' se lit 'tel que'. Dans la définition de la fonction 'abs' il y a 2 guardes. ' n >= 0' et 'otherwise'. Ce dernier guarde est toujours vrai.

L'avantage des gardes est qu'ils clarifient le code.

Un autre exemple

```
signe n | n < 0 = -1
        | n == 0 = 0
        | otherwise = 1
```

4.4 Motifs (*pattern matching*)

Beaucoup de fonctions ont des définitions intuitives. On peut les construite en utilisant une suite de motifs. Les motifs sont un peu comme pour les gardes qui permettent de choisir branche de code utiliser, mais ils s'appliquent directement aux arguments.

Par exemple pour la fonction suivante, en fonction des valeurs des paramètres on affichera soit 'True' soit 'False'.

```
et :: Bool -> Bool -> Bool
True `et` True = True
True `et` False = False
False `et` True = False
False `et` False = False
```

Dans cette définition on peut rassembler les trois dernières expressions grâce au joker '_' qui remplace n'importe quelle valeurs. La définition de la fonction devient alors :

```
et :: Bool -> Bool -> Bool
True `et` True = True
_ `et` _ = False
```

4.4.1 motifs avec des tuples

On peut aussi utiliser des tuples dans un motif.

```
premier :: (a,b) -> a
premier (a,_) = a
```

Ici, on selectionne les tuples de deux éléments et on renvoie le premier de ses éléments, ignorant le second.

4.4.2 motifs avec des listes

La même technique s'applique pour les listes. Un motif de liste, avec une taille donnée, sélectionnera toutes les listes de cette taille.

```
test :: [Char] -> Bool
test ['a',_,_] = True
test _ = False
```

La fonction 'test' renverra 'True' pour toutes les listes d'exactement trois éléments si elles commencent par le caractère 'a'.

Avec les list, on peut aussi utiliser l'opérateur ' :' (*cons*). il permet de construire une liste ou de décomposer les listes.

```
1 :  [2,3]  == [1,2,3]
```

```
True
```

Il ne faut pas le confondre avec l'opérateur de concaténation '++', qui s'utilise comme suit :

```
[1] ++ [2,3] == [1,2,3]
```

```
True
```

Voici deux exemples :

4.4.3 Entête d'une liste

```
entête :: [a] -> a
entête (x:_) = x
```

4.4.4 Queue d'une liste

```
queue :: [a] -> [a]
queue (_:xs) = xs
```

4.5 Expressions lambda

On peut définir des fonctions sans leur donner de nom. On appelle ces fonctions *lambda*. On définie leur paramètre avec la barre oblique '\' et la flèche '->' indique le résultat.

```
\x -> x + x
```

est la fonction qui prend une variable x et renvoie son double. On peut l'appeller comme cela :

```
(\x -> x + x) 3
```

```
6
```

Le nombre de variables des fonctions lambda n'est pas limité.

```
\x -> (\y -> x + y)
```

est l'écriture lambda de la fonction ajouter qui a deux paramètres

Ce type de notation est utile pour définir des fonctions à l'intérieur d'autres fonctions. Par exemple, voici la fonction 'impaires' qui génèrent les premiers nombres impairs. À la place d'écrire

```
impaires :: Int -> [Int]
impaires n = map f [0..n-1]
             where f x = x*2 +1
```

on peut noter

```
impaires' :: Int -> [Int]
impaires' map (\x -> x*2 +1) [0..n-1]
```

La fonction *map* applique a un ensemble de valeurs, la fonction passé comme son première paramètre.

4.6 Opérateurs

Un opérateur est une fonction à deux paramètres qui s'écrit entre ses paramètres, par exemple $2 + 3$. N'importe quelle fonction avec deux paramètres peut être converti en opérateur l'encadrant par des accents graves *back-quotes*, par exemple avec la division entière :

```
10 `div` 3
```

```
3
```

On peut aussi curryfier n'importe quel opérateur en l'encadrant avec des parenthèses, par exemple

```
(+) 2 3
```

```
5
```

De façon générale si '#' est un opérateur alors on a :

```
(#)  équivaut \x -> (\y -> x # y)
(x #) équivaut \y -> x # y
(# y) équivaut \x -> x # y
```

par exemple

```
(1/) équivaut à  \x -> 1/x
```

```
(1/) 4
```

```
0.25
```

et

```
(/2) équivaut à  \x -> x/2
```

```
(/2) 33
```

```
16.5
```

4.7 Exerices

4.7.1 Exercice n°1

En utilisant les fonctions de la librairie 'Prelude' définit une fonction appelé 'moitié' qui prend une liste d'éléments et qui renvoie un tuple contenant les deux moitiés de cette liste.

```
moitié :: [a] -> ([a],[a])
```

La fonction doit renvoyer une erreur si la liste n'a pas un nombre pair d'éléments.

4.7.2 Exercice n°2

Définit, de trois façons différentes, une fonction 'troisième' qui prend une liste d'éléments et qui renvoie le troisième élément de cette liste.

```
troisième :: [a] -> a
```

1. En utilisant les fonctions 'head' et 'tail'
2. en utilisant l'opérateur d'indice '!!'
3. en utilisant la correspondance de motif

4.7.3 Exercice n°3

Définir, de trois façons différentes, la fonction 'boutsécurisé' qui se comporte comme la fonction 'tail' sauf qu'en cas de liste vide, elle renvoie la liste passée en paramètre. Vous pouvez utiliser la fonction 'null' de 'Prelude' pour savoir si la liste est vide

```
boutsécurisé :: [a] ->[a]
null :: [a] -> Bool
```

1. En utilisant des expressions conditionnelles
2. En utilisant des équations gardées
3. En utilisant la correspondance de motif

4.7.4 Exercice n°4

définir l'opérateur '(ou)' de la même façon que ce que nous avons fait pour '(et)'

5 Listes en compréhension

Une liste en compréhension est une liste définie en utilisant les éléments d'une autre liste. En mathématiques, la liste des carrés des 5 premiers entiers notée en compréhension s'écrit

$$\{x^2 | x \in [1 \cdots 5]\}$$

.

En Haskell on note

```
[x^2 | x <- [1..5 ]]
```

```
[1,4,9,16,25]
```

On appelle la liste utilisé un *générateur*. On peut en utiliser plusieurs, par exemple

```
[(x,y) | x <- [1,2,3],  y <- "aB"]
```

```
[(1,'a'),(1,'B'),(1,'c'),(1,'d'),(2,'a'),(2,'B'),(2,'c'),(2,'d'),(3,'a'),(3,'B'),(3,'c
```

génère la liste des couples dont le premier élément est 1, 2 ou 3 et le deuxième 'a' ou 'B'.

L'ordre des générateurs est important.

```
[(x,y) | y <- "aB", x <- [1,2,3]]
```

```
[(1,'a'),(2,'a'),(3,'a'),(1,'B'),(2,'B'),(3,'B')]
```

C'est le second qui change plus vite.

5.1 Filtres *Guards*

Dans les listes en compréhension on peut utiliser des filtres (ou *guards*) pour limiter les valeurs du générateurs à prendre en compte. Les guardes viennent après le symbole ',', par exemple :

```
diviseurs_de :: Int -> [Int]
diviseurs_de n = [x | x <- [1..n], n `mod` x == 0]
```

5.2 la fonction zip

La fonction 'zip' est souvent utilisé avec les listes en compréhension. C'est une fonction qui crée une liste de paires en prenant deux éléments dans deux listes différentes, jusqu'à épuisement de l'une d'elles.

```
zip "abcd" [1..10]
```

```
[('a',1),('b',2),('c',3),('d',4)]
```

On peut s'en servire pour créer des paires, à partir d'une même liste.

```
paires :: [a] -> [(a,a)]
paires xs = zip xs (tail xs)
```

Ce qui donne

```
paires [1..5]
```

```
[(1,2),(2,3),(3,4),(4,5)]
```

On peut ainsi définir une fonction qui vérifie si une liste et triée.

```
est_triée :: Ord a => [a] -> Bool
est_triée xs = and [x <= y | (x,y)  <- paires xs]
```

Vérifions avec

```
est_triée [1,2,3,10,5]
```

```
False
```

5.3 Exercices

5.3.1 Exercice n°1

En utilisant une liste en compréhension donner l'expression qui calcule la somme des carrés des cent premiers entiers. $1^2 + 2^2 + \cdots + 100^2$

5.3.2 Exercice n°2

Soi un système de coordonnées sur une grille de taille $m \times n$ définie par un couple (x, y) avec $0 \leq x \leq m$ et $0 \leq y \leq n$. En utilisant les fonctions standard définir la fonction 'grille' qui renvoie les coordonnées de tous les points d'une crise dont les dimensions sont passées en paramètre.

```
grille :: Int -> Int -> [(Int,Int)]
```

```
> grille 1 2
```

```
[(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]
```

5.3.3 Exercice n°3

Définir une fonction ‘réplique‘ qui prend en paramètre un entier n et une valeur et qui renvoie une liste où la valeur est répliquée n fois.

```
réplique :: Int -> a -> [a]
```

```
> réplique 3 True
```

```
[True, True, True]
```

6 Fonctions récursives

6.1 Concepts de bases

Une fonction récursive et une fonction qui s'appelle elle-même. On peut définir la fonction ‘factorielle‘ récursivement.

```
factorielle :: Int -> Int
factorielle 0 = 1
factorielle x = x * factorielle (x - 1)
```

Ce n'est pas la façon la plus rapide à calculer mais cela est très lisible.

6.2 Récursion et listes

Voici d'autres exemples de fonctions récursive avec des listes en paramètre :

6.2.1 Produit

```
produit :: Num a => [a] -> a
produit [] = 0
produit (n:ns) = n * produit ns
```

6.2.2 Longueur

```
longueur :: [a] -> Int
longueur [] = 0
longueur (_:xs) = 1 + longueur xs
```

6.2.3 Renverse

```
renverse :: [a] -> [a]
renverse [] = []
renverse (x:xs) = reverse xs ++ [x]
```

6.2.4 Coupler

Les fonctions récursives peuvent aussi avoir plusieurs paramètres.
La fonction 'coupler' fait comme le 'zip' et on la définie comme suit :

```
coupler :: [a] -> [b] -> [(a,v)]
coupler [] _ = []
coupler _ [] = []
coupler (x:xs) (y:ys) = (x,y) : coupler xs ys
```

6.2.5 Décapiter

La fonction 'décapiter' qui supprime les n premiers éléments d'une liste.

```
décapiter :: Int -> [a] -> [a]
décapiter 0 xs = xs
décapiter _ [] = []
décapiter n (_:xs) = décapiter (n-1) xs
```

6.3 Récursions multiples

Une fonction récursive peut s'appeler elle-même plusieurs fois, par exemple, ici, pour construire la suite de fibonacci :

```
fibo :: Int -> Int
fibo 0 = 0
fibo 1 = 1
fibo n = fibo (n-2) + fibo (n-1)
```

et ici pour définir un algorithme de tri_rapide

```
tri_rapide :: Ord a => [a] -> [a]
tri_rapide [] = []
tri_rapide (x:xs) = tri_rapide plus_petits ++ [x] ++ tri_rapide plus_grands
                    where
                      plus_petits = [a | a <- xs, a <= xs]
                      plus_grands = [b | b <- xs, b > xs]
```

6.4 Récursion mutelle

Les fonctions récursives en Haskell peuvent aussi faire appellent à des fonctions qui les rappellent. Par exemple ‘est_pair‘ à besoin de ‘est_impair‘

```
est_pair :: Int -> Bool
est_pair 0 = True
est_pair n = est_impair (n-1)
```

qui a besoin de ‘est_pair‘.

```
est_impair :: Int -> Bool
est_impair 1 = True
est_impair n = est_pair (n-1)
```

Ou encore la fonction ‘indices_pairs‘ qui selection un élément s sur deux dans une liste en partant du premier, à besoin des ‘indices_impairs‘

```
indices_pairs :: [a] -> [a]
indices_pairs [] = []
indices_pairs (x:xs) = x : indices_impairs xs
```

qui a besoin de ‘indice_pairs‘.

```
indices_impairs :: [a] -> [a]
indices_impairs [] = []
indices_impairs (_:xs) = indices_pairs xs
```

6.5 Conseils pour construire des fonctions récursives.

La récursion, c’est comme le vélo. Cela semble simple lorsque l’on voit quelqu’un d’autre en faire mais il faut de la pratique pour bien y arriver.

Voici une démarche à suivre et quelques conseils pour y arrriver.

6.5.1 Définir le type de la fonction

```
produit :: [Int] -> Int
```

6.5.2 Identifier les cas importants

```
produit [] =
produit (n:ns) =
```

6.5.3 Écrire la définition de cas simples

```
produit [] = 1
produit (n:ns) =
```

6.5.4 Écrire la définition des autres cas

```
produit [] = 1
produit (n:ns) = n * produit ns
```

6.5.5 Généraliser et simplifier

Dans notre exemple la fonction 'produit' peut s'appliquer à tous les types de la classe 'Num'.

```
produit :: Num a => [a] -> a
```

On va voir dans le prochain chapitre que la manière dont nous venons de définir la fonction 'produit' est très générique et Haskell permet de coder une multitude de fonction de cette manière à l'aide de la fonction *foldr*.

'produit' se définit alors en une ligne.

```
produit = foldr (*) 1
```

6.6 Exerices

6.6.1 Exercices n°1

Définir une fonction récurcive 'euclide' qui calcule le plus grand diviseur de deux nombres non-négatifs.

```
euclide :: Int -> Int -> Int
```

Si les deux nombre sont égaux, alors ce nombre est le résultat, sinon le plus petit est soustrait au plus grand.

6.6.2 Exercice n°2

Définir de façon récursive les fonctions suivantes :

1. 'et_liste'

   ```
   et_liste :: [Bool] -> Bool
   ```

   décide si tout les valeurs de la liste sont vraie
2. 'concatène'

`concatène :: [[a]] -> [a]`

Concatène une liste de listes

3. 'est_élément'

   `est_élément :: Eq a => a -> [a] -> Bool`

   Décide si un élément fait partie d'une liste.

6.6.3 Exercice n°3

Définie une fonction récursive 'fusionne' qui rassemble deux listes en une seule triée.

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

par exemple

```
>merge [2,5,6] [1,3,4]
```

```
[1,2,3,4,5,6]
```

7 Fonctions de haut-niveau (higher-order)

Une fonction de haut-niveau est une fonction qui **prend d'autre(s) fonction(s) en paramètre**.

7.1 Exemples de base

Les fonctions de ce types les plus courantes sont 'map' et 'filter'. 'map' applique la fonction passé en paramètre sur une liste d'éléments.

```
map (\x -> x^2) [1..5]
```

```
[1,4,9,16,25]
```

'filter' renvoie une liste d'élément sélectionné dans une autre liste selon un prédicat passé sous forme de fonction en paramètre, par exemple :

```
filter (\[x,y] -> x == y)  [ "aa", "ab", "Aa", "AA"]
```

```
["aa","AA"]
```

Nous donnons maintenant des exemples courants de fonctions de haut-niveau.

7.2 fold

— 'foldr' applique un opérateur sur une liste en allant de la droite à la gauche. Elle est souvent appelée *reduce* dans d'autre langage.

```
longueur :: [a] -> Int
longueur = flodr (\_ n -> 1 +n) 0
```

— 'foldl' : applique un opérateur sur une liste en allant de la droite à la gauche.

7.3 Composition

L'opérateur '(.)' permet de faire de la composition de fonction. Cela simplifie et de clarifie le code. À la place de noter

```
impaire n = not (paire n)
```

on note

```
impaire  = not . pair
```

À la place de

```
repéter f x = f (f x)
```

on écrit

```
répeter = f . f
```

et à la place de

```
somme_carré_positif ns = sum (map (^2) (filter paire ns))
```

on écrira

```
somme_carré_positif = sum . map (^2) . filter pair
```

7.4 Exercices

7.4.1 Exercice n°1

Redéfinir la liste en compréhension $[fx|x < -xs, px]$ en utilisant les fonctions de haut niveau 'map' et 'filter'

1. solution

```
-- [f x | x <- xs, p x] ?
exo1 :: (a -> b) -> [a] -> (a -> Bool) -> [b]
exo1 f xs p = [f x | x <- xs, p x]

exo1sol :: (a -> b) -> [a] -> (a -> Bool) -> [b]
exo1sol f xs p = map f (filter p xs)
-- map f filter p xs
```

7.4.2 Exercice n°2

Redéfinir les fonctions ‘map f‘ et ‘filter p‘ en utilisant ‘foldr‘

1. solution

```
-- exo3 redefine map f and filter p using foldr

map_ :: (a -> b) -> [a] -> [b]
map_ f xs = foldr apply_f [] xs
  where apply_f x y = (f x : y)

map__ :: (a -> b) -> [a] -> [b]
map__ f xs = foldr (\x y -> f x : y) [] xs


filter_ :: (a -> Bool) -> [a] -> [a]
filter_ p xs = foldr filter_p [] xs
  where filter_p x y
          | p x = (x:y)
          | otherwise = y
```

7.4.3 Exercice n°3

En utilisant ‘foldl‘ définr la fonction dec2Int qui converti une liste de nombre en entier

```
dec2Int :: [Int] -> Int

dec2int [2,3,4,5]

2345
```

1. sol

```
-- exo 4 p. 105
-- Using foldl, define a function dec2int :: [Int] -> Int that converts a
-- decimal number into an integer. For example:

dec2int :: [Int] -> Int
dec2int xs = foldl (\x y -> 10 * x + y) 0 xs
```

8 Déclaration des Types et des Classes

Les noms de types de classe **commencent avec une majuscule**.

8.1 Comment déclarer de nouveau type

8.1.1 types

On peut les définir en utilisant des types existants

```
type Position = (Int, Int)
type Transistion = Pos -> Pos
```

Les types récursifs sont interdits.

```
type Arbre = (Int, [Tree]) --interdit
```

Par contre, on peut utiliser les classes de type

```
type Paire a = (a,a)
type Dico k v = [(k,v)]
```

8.1.2 data

'data' permet de créer un nouveau type en énumérant les différentes valeurs possibles.

```
data Déplacement = Nord | Sud | Est | Ouest deriving Show

déplace :: Déplacement -> Position -> Position
déplace North (x,y) = (x,y+1)
déplace Sud (x,y) = (x, y-1)
déplace Est (x,y) = (x+1,y)
déplace Ouest (x,y) = (x-1,y)
```

‘data‘ permet d’utiliser des variables et des paramètres. Par exemple :

```
data Forme = Cercle Float | Rectangle Float Float

faire_carré :: Float -> Forme
faire_carré n = Rectangle n n

faire_cercle :: Float -> Forme
faire_cercle r = Cercle r

calculer_aire :: Forme -> Float
calculer_aire (Rectangle long larg) = long * larg
calculer_aire (Cercle r) = pi * r^2
```

Ci-dessus, les constructeurs ‘Cercle‘ et ‘Rectangle‘ sont des fonctions constructeurs, à cause de leurs variables. On le voit en demandant le type de ‘Cercle‘.

```
:t Cercle

Cercle :: Float -> Forme
```

— un autre exemple de type avec une variable

```
data Maybe a = Nothing | Just a

division_securisée :: Int -> Int -> Maybe Int
division_securisée _ 0 :: Nothing
division_securisée m n = Just (m ‘div‘ n)
```

8.1.3 newtype

La troisième façon de créer un type est en utilisant le mot clef ‘newtype‘.

```
newtype Entier_Naturel = N Int
```

8.1.4 types récursifs

Dans certains cas les définitions des types peuvent être récursives, par exemple

```
data Naturel = Zero | Succ Naturel deriving Show
data Arbre a = Feuille a | Noeud (Arbre a) a (Arbre a) deriving Show
```

8.2 Classes de type

Les classes de types ne sont possible que pour les types déclarés avec ‘data‘ et ‘newtype‘.

Voici comment on déclare un classe de type en Haskell

```
class Eq a where
    (==), (/=) :: a -> a -> Bool

    x /= y = not (x==y)
```

et comment on instancie cette classe de type.

```
instance Eq Bool where
    False == False = True
    True  == True  = True
    _ == _ = False
```

8.2.1 instances de type dérivées

Lors de la création d'un type il est possible de les faire dériver de classe existante et il aura alors les méthodes par défaut de ces classes.

La dérivation se fait comme suit :

```
data Bool = False | True
          deriving (Eq, Ord, Show, Read)
```

Cela veut dire que la classe de type Bool hérite automatiquement des méthodes de base des classe ‘Eq‘, ‘Ord‘, ‘Show‘ et ‘Read‘.

8.3 Exercices

9 Annexes