Memento Hakell

1 Notations

1.1. Notation préfixée

```
Prelude> div 13 4
3
Prelude> mod 11 2
1
Prelude> (++) 2 3
5
```

1.2. Notation infixée

```
Prelude> 13 `div` 4
3
Prelude> 11 `mod` 2
1
Prelude> 2 + 3
5
```

1.3. Lambda fonction

```
(\x -> x + 2)-- ajoute 2
```

2 Fonctions de base

2.1. Arithmétique :

`+` : addition

`-` : soustraction

`*` : multiplication

`/` : division (pour les types fractionnaires)

`div` : division entière

`mod` : modulo

2.2. Comparaison :

`==` : égalité

`/=` : différence

`<` : inférieur

`<=` : inférieur ou égal

`>` : supérieur

`>=` : supérieur ou égal

2.3. Listes

2.3.1 Listes en compréhension sur les entiers :

`[val1 .. val2]` : permet d'obtenir la liste des entiers compris, au sens large, entre `val1` et `val2`

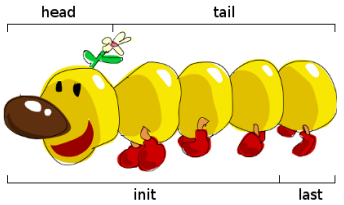
`[val..]` : permet d'obtenir la liste infinie des entiers en commençant à la valeur `val`

`[val1, val2.. val3]` : permet d'obtenir la liste des entiers compris entre `val1` et `val3` par pas de (`val2-val1`)

`[val1, val2..]` : permet d'obtenir la liste infinie des entiers compris en commençant à `val1` par pas de (`val2-val1`)

`enumFrom val` : permet d'obtenir la liste infinie des entiers à partir de la valeur `val`

2.3.2 Fonctions simples



`head` : retourne le premier élément d'une liste

```
Prelude> head [1, 2, 3]
1
```

`tail`: retourne tous les éléments d'une liste sauf le premier

```
Prelude> tail [1, 2, 3]
[2, 3]
```

`init` : retourne tous les éléments d'une liste sauf le dernier

```
Prelude> init [1, 2, 3]
[1, 2]
```

`last` : retourne le dernier élément d'une liste

```
Prelude> last [1, 2, 3]
3
```

`length` : retourne la longueur d'une liste

```
Prelude> length [True, False, True]
3
```

`null` : vérifie si une liste est vide

```
Prelude> null []
True
```

`reverse` : renverse une liste

```
Prelude> reverse [1, 2, 3]
[3, 2, 1]
```

`!!` : accès à l'élément d'une liste par son index

```
Prelude> [1, 2, 3] !! 2
3
```

`take` : prend les *n* premiers éléments d'une liste

```
Prelude> take 3 [1..10]
[1, 2, 3]
```

`drop` : enlève les n premiers éléments d'une liste

```
Prelude> drop 7 [1..10]
[8, 9, 10]
```

`concat` : concatène une liste de listes

```
Prelude> concat [[1, 2, 3], [6, 5, 4]]
[1, 2, 3, 6, 5, 4]
```

`++` : concatène deux listes

```
Prelude> [1, 2, 3] ++ [6, 5, 4]
[1, 2, 3, 6, 5, 4]
```

`zip` : combine deux listes en une liste de paires

```
Prelude> zip [1, 2, 3] [6, 5, 4]
[(1,6),(2,5),(3,4)]
```

`unzip`: décompose une liste de paires en deux listes. Opération réciproque de zip.

`elem` : test l'appartenance d'un élément dans une liste

```
Prelude> elem 2 [1, 2]
True
Prelude> elem 3 [1, 2]
False
```

2.3.3 Fonctions d'ordre supérieur de base

`takeWhile` : prend les premiers éléments d'une liste selon un prédicat.

```
Prelude> takeWhile (\x -> x < 4) [0..]
[0, 1, 2, 3]
```

`dropWhile` : enlève les premiers éléments d'une liste selon un prédicat.

```
Prelude> dropWhile (\x -> x < 4) [0..6]
[4, 5, 6]
```

`map` : applique une fonction à chaque élément d'une liste

```
Prelude> map succ [1, 4, 9]
[2, 5, 10]
```

`filter` : filtre les éléments d'une liste selon un prédicat

```
Prelude> filter (\x -> x > 4) [12, 2, 8, 9, 3]
[12, 8, 9]
```


2.3.4 Fonctions d'accumulation

`foldl` : réduction à gauche avec valeur initiale

```
Prelude> foldl (+) 0 [1, 2, 3, 4]
10
```

Calcule `(((0 + 1) + 2) + 3) + 4`

`foldl1` : réduction à gauche sans valeur initiale

```
Prelude> foldl1 (+) [1, 2, 3, 4]
10
```

Calcule `(((1 + 2) + 3) + 4`

`foldr` : réduction à droite

```
Prelude> foldr (+) 0 [1, 2, 3, 4]
10
```

Calcule `1 + (2 + (3 + (4 + 0)))`

2.4. Fonctions de conversion :

`show` : convertit une valeur en chaîne de caractères

```
Prelude> show 12
"12"
```

`read` : convertit une chaîne de caractères en une valeur

```
Prelude> True && read "False"
False
```

2.5. Fonctions sur les tuples :

`fst` : retourne le premier élément d'un tuple

```
Prelude> fst (4, 5)
4
```

`snd` : retourne le deuxième élément d'un tuple

```
Prelude> snd (4, 5)
5
```

2.6. Fonctions logiques

2.6.1 Fonctions simples

`&&` : et logique

`||` : ou logique

`not` : négation logique

2.6.2 Fonctions d'ordre supérieur

`any` : `any` *`predicat liste`*. Renvoie vrai si au moins un des éléments de la liste respecte le prédicat.

`all` : `all` *`predicat liste`*. Renvoie vrai si tous les éléments de la liste respectent le prédicat.

2.7. Entrée/Sortie (I/O)

2.7.1 Fonctions de base

`putStr` : affiche une chaîne de caractères

`putStrLn` : affiche une chaîne de caractères avec un retour à la ligne

`getLine` : lit une ligne de l'entrée standard

`print` : affiche une valeur (utilise show pour la convertir en chaîne de caractères)

2.7.2 Début de programme en Haskell :

```
main :: IO ()
main =  do
    instructions
```

3 Typage en Haskell

3.1. Types

On peut demander le type d'une donnée avec `:t`.

```
Prelude>:t True
Bool
```

3.1.1 Types simples

`Int`, `Integer` : types entiers

`Float`, `Double`: types flottants

`Bool` : type booléen

3.1.2 Types génériques

`[a]` : une liste d'éléments de type a

`(a, b)`: un tuple (premier élément de type a, deuxième de type b)

3.2. Classes de type

Sorte d'interface (ensemble de fonctions) que peuvent implanter certains types.

On peut demander des informations sur un type ou une classe de type avec `:i`.

```
Prelude>:i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’
```


`Num` : nombres

`Integral` : entiers

`Real`: flottants

`Show` : affichable (avec show)

`Enum` : énumérable (avec succ et pred)

`Ord`: muni d'une relation d'ordre (<, <=, >, >=)

`Eq` : comparable (==,/=)

3.3. Liens entre type et classe de type

un type peut implanter des classes de type (voir `instance Num Int` dans l'exemple précédent).

une classe de type peut hériter d'autres classes de type

`Ord` hérite de `Eq` : `class Eq a => Ord a`

`Integral` hérite de `Num` et `Real` : `class (Real a, Enum a) => Integral`

3.4. Typage d'une fonction

3.4.1 Forme de base :

`type param1 -> type param2 -> … type param n -> type résultat`

Exemple :

```
(&&) :: Bool -> Bool -> Bool
```

3.4.2 Forme avec type paramétré :

Exemple :

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

Ici, `a` est une variable de type : `head` prend une liste d'éléments d'un type `a` quelconque, et renvoie une donnée du même type `a`.

3.4.3 Forme avec type paramétré contraint :

Exemple :

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

Ici, `a` est une variable de type, pouvant être un type quelconque, mais devant impanter la classe de type `Num`.

4 Typages des fonctions évoquées

Fonction	Type
+, -, *	Num a => a -> a -> a
/	Fractional a => a -> a → a
div, mod	Integral a => a -> a -> a
==, /=	Eq a => a -> a -> Bool
<, <=, >, >=	Ord a => a -> a -> a
head, last	[a] -> a
tail, init	[a] -> [a]
length	Foldable t => t a -> Inr
null	Foldable t => t a -> Bool
reverse	[a] -> [a]
!!	[a] -> Int -> a
take, drop	Int -> a -> a
drop	Int -> a -> a
concat	Foldable t => t [a] -> [a]
(++)	[a] -> [a] -> [a]
zip	[a] -> [b] -> [(a, b)]
Unzip	[(a, b)] -> ([a], [b])
elem	(Foldable t, Eq a) => a -> t a -> Bool
takeWhile, dropWhile	(a -> Bool) -> [a] -> [a]
map	(a -> b) -> [a] -> [b]
filter	(a -> Bool) -> [a] -> [a]
foldl, foldr	Foldable t => (b -> a -> b) -> b -> t a -> b
foldl1, foldr1	Foldable t => (a -> a -> a) -> t a -> a
show	Show a => a -> String
read	Read a => String -> a
fst	(a, b) -> a
snd	(a, b) -> b
&&, ||	Bool -> Bool -> Bool
not	Bool -> Bool
any, all	Foldable t => (a -> Bool) -> t a -> Bool
putStr, putStrLn	String -> IO ()
getLine	IO String
print	Show a => a -> IO ()