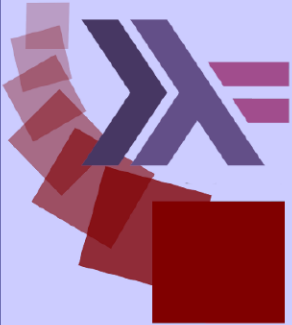
PROGRAMMATION FONCTIONNELLE

via le langage Haskell

OUALI Abdelkader

L2 au département Mathématique-Informatique
UFR des sciences
Université de Caen Normandie



GHC et GHCi

UNIVERSITÉ CAEN NORMANDIE

- À partir d'un terminal, le compilateur est appelé en tant que

```
> ghc monfichier.hs
> runghc monfichier.hs (sans créer le fichier binaire)
> ghci (ou comme >ghc --interactive)
```

- GHCi opère sur une boucle évaluer-afficher

```
Prelude>sqrt(3^2 + 4^2)
5.0
Prelude>
```

- Possibilité d'invoquer un éditeur dans le terminal

GHCi

Commandes de bases utiles dans GHCi :

- > :? – Help ! Afficher toutes les commandes
- > :load test – Ouvrir le fichier test.hs
- > :reload – Recharge le fichier précédemment chargé
- > :main a1 a2 – Invoque main avec les arguments de ligne de commande a1 a2
- > :! – Exécuter une commande shell
- > :edit nom – Modifier le script nom.hs
- > :edit – modifier le script actuel
- > :type expr – Afficher le type d'une expression
- > :quit –Quitter GHCi

Les commandes peuvent être **abrégées**. Par exemple, :r pour ;reload
Au démarrage, les définitions du "**Standard Prelude**" sont chargées.

Syntaxe de Base

Pour mieux coder, il faut retenir :

- Valeur : donnée concrète p.ex. "toto", 10, . . .
- Type : ensemble de valeurs ayant des propriétés (opérations) communes : Int, Char, . . .
- Expression : un morceau de code pouvant être calculé $21 * 2$
- Fonction : expression avec des arguments en entrée
- Quand on parle d'une variable $\rightarrow$ nom associé à une expression

Les nombres entiers : types `Int` et `Integer`

UNIVERSITÉ CAEN NORMANDIE

- opérateurs infixes : `+, -, *`
- opérateurs préfixes : `div, mod`
- précédence et associativité
- comparateurs : `<, <=, ==, /=, >, >=`
- type `Int` (précision limitée) vs type `Integer` (précision infinie)
- le "-" représente la soustraction ou le signe de négativité
  ➠ Pour éviter la confusion utiliser les parenthèse quand il s'agit d'un nombre négatif

Types élémentaires : entiers `Int`

- **Nombres Entiers (types `Int` et `Integer`)**

```
> 6*7   ==> 42

> 7*4 + 3    ==> 31

> div 13 4    ==> 3

> mod 13 4    ==> 1
```

Les nombres flottants : types `Float` et `Double`

UNIVERSITÉ CAEN NORMANDIE

- opérateurs infixes : `+`, `-`, `*`, `/`,
- précédence et associativité
- comparateurs : `<`, `<=`, `==`, `/=`, `>`, `>=`
- type `Float` (simple précision) vs type `Double` (double précision)
- les nombres flottants sont des **approximations** des nombres réels

Types élémentaires : flottants `Float`

- **Nombres Flottants (types `Float` et `Double`)**

```
> sqrt 65   ==> 8.06225774829855

> pi    ==> 3.141592653589793

> sin (pi/2)    ==> 1.0

> cos (pi/4)    ==> 0.7071067811865476
```

Les nombres flottants : types `Float` et `Double`

```
--script
mustBeTheSame :: Double -> Bool
mustBeTheSame n = (sqrt n) * (sqrt n) == n

--session
> mustBeTheSame 1   ==> True
> mustBeTheSame 10    ==> False
> mustBeTheSame 100    ==> True
> mustBeTheSame 100000    ==> False

> mustBeTheSame 0.1    ==> True
> mustBeTheSame 0.01    ==> False
> mustBeTheSame 0.0001    ==> True
> mustBeTheSame 0.000001    ==> True
> mustBeTheSame 0.0000001    ==> False
```

En conséquence, **ne jamais tester l'égalité de réels** représentés par des nombres flottants.

Les nombres flottants : types `Float` et `Double`

- ne jamais tester l'égalité de réels représentés par des nombres flottants
- égalité selon une précision voulue

```
estIdentiqueBis :: Double -> Bool
estIdentiqueBis n = abs ((sqrt n) * (sqrt n) - n) < 1/10^10

> estIdentiqueBis 1   => True
> estIdentiqueBis 100   => True
> estIdentiqueBis 1000   => True

> estIdentiqueBis 0.1   => True
> estIdentiqueBis 0.01   => True
> estIdentiqueBis 0.0001   => True
> estIdentiqueBis 0.0000001   => True
```

Les booléens : type `Bool`

UNIVERSITÉ CAEN NORMANDIE

- seulement 2 valeurs : `True` et `False`
- opérateurs infixes : `&&, ||`
- opérateurs préfixe : `not`

Types élémentaires : booléen `Bool`

- **Booléens (type `Bool`)**

```
> True && False   ==> False

> True || False   ==> True

> 2 == 1+1    ==> True

> 5 > 3*4    ==> False
```

Les caractères : type `Char`

UNIVERSITÉ CAEN NORMANDIE

- exemples : 'a', '1', '-', ' ', '.'
- caractères spéciaux : tab = `'\t'`, newline = `'\n'`, backslash = `'\\'`
- relation d'ordre sur les `Char`
  - fonctions `succ` et `pred` de type `(Char -> Char)`
  - ainsi que les comparateurs : `<`, `<=`, `==`, `/=`, `>`, `>=`

Types élémentaires : Caractères `Char`

- **Caractères (type `Char`)**

```
> 'a'   ==> 'a'

> 'A' < 'a'  ==> True

> succ 'b'  ==> 'c'

> pred 'z'  ==> 'y'
```

Listes

- **Liste d'entiers : type** `[Int]`

```
> [1..5]   ==> [1,2,3,4,5]
> [1,4..10] ==> [1,4,7,10]
> [2+3, 7-4, div 72 7, 23]   ==> [5,3,10,23]
```

- **Liste de booléens : type** `[Bool]`

```
> [1 < 2, True, 'a'=='b'] => [True,True,False]
```

Listes

- **Liste de caractères : type `String`**

```
> "hello " ++ "world"   ==> "hello world"

> ['h', 'e', 'l', 'l', 'o']   ==> "hello"
```

- **`String` et `[Char]` sont des types synonymes**

```
> "hello" == ['h', 'e', 'l', 'l', 'o']   ==> True

> "hello " == ['h', 'e', 'l', 'l', 'o', ' ']   ==> True
```

Tuples

- **Tuples**

```
> (2^5, True || False, 'a')   ==> (32, True, 'a')

> (pi/2, 'A' < 'a')   ==> (1.5707963267948966, True)

> ("Lundi", 9, "Janvier", 2018)

==> ("Lundi", 9, "Janvier", 2018)
```

Définir une fonction

- Les noms de fonctions et de paramètres doivent commencer par une lettre minuscule, par exemple myFun1, arg_x, personName, etc.
  - ➠ par convention, un argument de type liste contient "s" un suffixe s dans son nom, p.ex. xs, ns, nss, etc.)
- Une fonction est définie par une égalité :

```
carré x = x * x
plus x y = x + y
```

- Dès que la fonction est définie, on peut l'appliquer par appel de son nom et les arguments :
  - ➠ dans d'autres langage p.ex. C : carré(5) ; et plus(3,2) ;

```
> carré 5
25
> plus 3 2
5
```

- Les parenthèses sont aussi nécessaires dans Haskell, p.ex.

```
> plus (carré 5) (carré 3)
34
```

Définition de fonctions

- L'application de fonction a la priorité la plus élevée par rapport à tout

  ➠ **carré 5 + 3 signifie (carré 5) + 3**

- L'appel de fonction s'associe à gauche et se fait par **pattern-matching**

  ➠ le **premier qui correspond** est utilisé

- L'opérateur d'application de fonction $ a la priorité la plus faible

  ➠ utilisé pour supprimer les parenthèses

```
sum ([1..5] ++ [6..10]) sum $ [1..5] ++ [6..10]
```

Définir une fonction

UNIVERSITÉ CAEN NORMANDIE

- **Factorielle d'un entier**

```
fact :: Int -> Int

fact n = if (n == 0) then 1 else n*(fact (n-1))
```

- **Nombre de chiffres représentant un nombre entier**

```
nbDigits :: Int -> Int

nbDigits n = if (n <= 9) then 1 else 1 + (nbDigits (div n 10))
```

Définir un opérateur

- Des combinaisons de la plupart des symboles sont autorisées comme opérateur

```
4 $ % ^ & * - + # 23
```

```
x +/- y = (x+y, x-y)
> 22 +/- 90
(112,-68)
```

Application de fonctions

UNIVERSITÉ CAEN NORMANDIE

- En mathématique,
  - l'application de fonction est utilisé avec des parenthèses
  - la multiplication est dénoté par juxtaposition ou un espace
- `f(a,b) + c d`
- Dans Haskell,
  - l'application de la fonction est dénoté par un espace
  - la multiplication est dénoté par *

Exemples

Mathématiques	Haskell
f(x)	f x
f(x,y)	f x y
f(g(x))	f (g y)
f(x, g(y))	f x (g y)
f(x) g(y)	(f x) * (g y)

Évaluation d'une fonction

Évaluer par une substitution et une réduction :

```
plus x y = x + y; carré x = x * x
plus (carré 2) (plus 2 3)
-- appliquer carré
plus (2 * 2) (plus 2 3)
-- appliquer *
plus 4 (plus 2 3)
-- appliquer inner plus
plus 4 (2 + 3)
-- appliquer +
plus 4 5
-- appliquer plus
4+5
-- appliquer +
9
```

Évaluation d'une fonction

Plusieurs façon d'appliquer les fonctions :

```
head (1:(reverse [2,3,4,5]))
-- appliquer reverse
-- ... many steps omitted here
head ([1,5,4,3,2])
-- appliquer head
1
```

```
head (1:(reverse [2,3,4,5]))
-- appliquer head
1
```

- Ordre d'évaluation ne modifie pas le résultat
- Affecte le calcul (si calcul termine, avec ou sans échec)
- Évaluation paresseuse

Syntaxe de Base

- Commentaires :

```
-- commentaire de fin de ligne (sur une seule ligne)
{- commentaire
en multiligne {- et imbricable -} -}
```

- les parenthèses servent uniquement à indiquer des priorités :

```
f x = 2 * (x - 3)
```

- et non pour évaluer une fonction :

```
Prelude> f 4
2
```

Quelques mots réservés

Quelques mots réservés à retenir pour le premier TP :
Liaison d'une expression à un nom ("définir les variables")

- `let ... in ...` :

```
f = let y = 1+2
        z = 4+6
    in  y+z
```

- `where` :

```
f = y+z
 where y = 1+2
       z = 4+6
```

- `let ... in ...` : est une expression contrairement au `where` qui est réservé pour faire une construction syntaxique

Quelques mots réservés

Quelques mots réservés à retenir pour le premier TP :

- `if expression then expression else expression` : contrairement au langage impératif le `else` est obligatoire

```haskell
plusPetit x y = if (x < y) then x
                else y
```

- `case of` :

```haskell
troisPremiers x = case x of
          1 ->  "A"
          2 ->  "B"
          3 ->  "C"
          _ ->  "*" -- pattern-matching pour le choix pardéfaut (voir plus tard)
```

Prelude : quelques fonctions utiles

- Haskell est diffusé avec un grand nombre de fonctions dans la bibliothèque standard
- Fonctions numériques familières telles que + et *
- De nombreuses fonctions utiles sur les listes
- Sélectionner le premier élément d'une liste :

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

- Supprimer le premier élément d'une liste :

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

- Sélectionner le nème élément d'une liste :

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

Prelude : quelques fonctions utiles

- Sélectionner les n premiers éléments d'une liste :

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

- Supprimer les n premiers éléments d'une liste :

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

- Inverser une liste :

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

- Calculer la longueur d'une liste :

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

- Calculer la somme d'une liste :

```
Prelude> sum [1,2,3,4,5]
15
```

- Calculer le produit d'une liste :

```
Prelude>product [1,2,3,4,5]
120
```