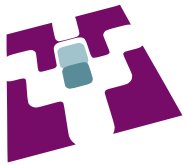
UNIVERSITÉ
CAEN
NORMANDIE

Autre Paradigme

Patrice Boizumault

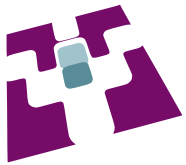
Université de Caen - Normandie

Lundi 13 janvier 2020



Partie #2 : Premier aperçu des types simples

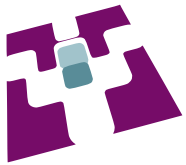
1. les nombres entiers : type `Int` et type `Integer`
2. les booléens : type `Bool`
3. les caractères : type `Char`
4. les nombres flottants : type `Float` et type `Double`
5. chaînes de caractères : `[Char]` et `String` sont des types synonymes



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

- opérateurs infixes : `+`, `-`, `*`
- opérateurs préfixes : `div`, `mod`
- précédence et associativité
- comparateurs : `<`, `<=`, `==`, `/=`, `>`, `>=`
- type `Int` (précision limitée) vs type `Integer` (précision infinie)
- fonctions `sum` et `product`

```
> sum [1, 5, 7, 13]   ==> 26
> sum []   ==> 0

> product [1, 5, 7, 13]   ==> 455
> product []   ==> 1
```



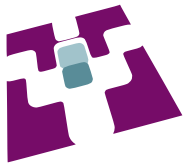
Les nombres entiers : types Int et Integer

- script

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

factBis :: Integer -> Integer
factBis n = if (n == 0) then 1 else n*(factBis (n-1))
```

- session

```
> fact 25     ==> 7034535277573963776
> fact 26     ==> -1569523520172457984

> factBis 26     ==> 403291461126605635584000000
> factBis 40
    ==> 815915283247897734345611269596115894272000000000
> factBis 50
   ==> 30414093201713378043612608166064768844377641568960512000000000000
```

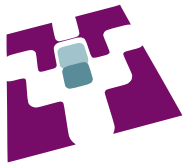


Traitement récursif des entiers

Factorielle

- deux définitions équivalentes :

```
fact, fact2 :: Integer -> Integer

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

-- en utilisant les gardes

fact2 n
   | n == 0    = 1
   | otherwise = n * fact2 (n - 1)
```

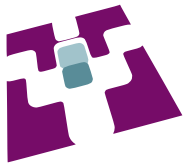


Traitement récursif des entiers

Factorielle

- une 3-ème définition elle aussi équivalente :

```
fact, fact3 :: Integer -> Integer

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

-- en utilisant le pattern matching sur les entiers

fact3 0 = 1
fact3 n = n * fact3 (n - 1)
```

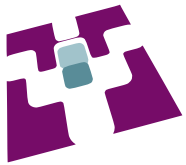


Traitement récursif des entiers

Nombres de Fibonacci

- définitions équivalentes :

```
fibo, fibo2 :: Integer -> Integer

fibo n = if (n == 0) || (n == 1)
         then 1
         else fibo (n-1) + fibo (n-2)

-- en utilisant les gardes

fibo2 n
   | n == 0    = 1
   | n == 1    = 1
   | otherwise = fibo2 (n-1) + fibo2 (n-2)
```

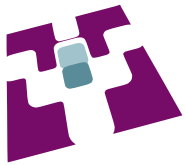


Traitement récursif des entiers

Nombres de Fibonacci

- une troisième définition équivalente :

```
fibo2, fibo3 :: Integer -> Integer

fibo2 n
   | n == 0    = 1
   | n == 1    = 1
   | otherwise = fibo2 (n-1) + fibo2 (n-2)

-- en utilisant le pattern matching sur les entiers

fibo3 0 = 1
fibo3 1 = 1
fibo3 n = fibo3 (n-1) + fibo3 (n-2)
```

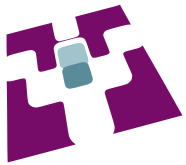


Traitement récursif des entiers

Elever un nombre à la puissance `n`

- Première définition :

```
> power 2 10   ==> 1024
> power 2 11 ==> 2048

power :: Integer -> Integer -> Integer

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

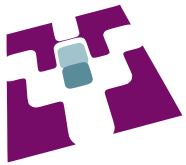


Traitement récursif des entiers

Elever un nombre à la puissance `n`

- Deux autres définitions équivalentes :

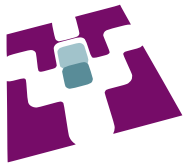
```
> power 2 10   ==> 1024
> power 4 5 ==>1024
> power 2 11 ==> 2048
> 2 * (power 2 10) ==> 2048

-- en utilisant le pattern matching sur les entiers
power2 x 0 = 1
power2 x n = x * (power2 x (n-1))

-- quid de cette definition ?
power3 x n
   | n == 0    = 1
   | odd n     = x * (power3 x (n-1))
   | otherwise = power3 (x * x) (div n 2)
```



Les booléens : type `Bool`

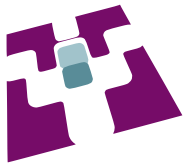
- seulement 2 valeurs : `True` et `False`
- opérateurs infixes : `&&`, `||`
- opérateurs préfixe : `not`
- exemple de pattern matching sur les booléens

```
negation :: Bool -> Bool

negation True = False

negation False = True
```



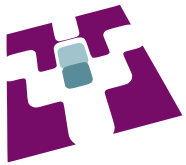
Les caractères : type `Char`

- 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 : `<`, `<=`, `==`, `/=`, `>`, `>=`



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

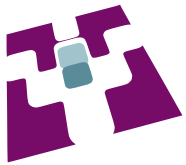
- opérateurs infixes : `+`, `-`, `*`, `/`,
- précédence et associativité
- comparateurs : `<`, `<=`, `==`, `/=`, `>`, `>=`
- type `Float` (simple précision) vs type `Double` (double précision)
- les nombres flottants sont des **approximations** des nombres réels



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
```

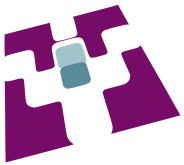
En conséquence, **ne jamais tester l'égalité de réels** représentés par des nombres flottants.



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
```



Traitement récursif de listes

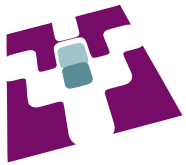
Longueur d'une liste

- polymorphisme grâce aux variables de type
- elle s'applique sur `[Int]`, `[Bool]`, `[Char]`, listes de listes, listes de ce que l'on veut ...

```
> length [3..12]   ==> 10
> length []   ==> 0
> length "hello wordl"   ==> 11
> length [[2,5], [1,-3,-12], [7..23]]   ==> 3
> length ["un", "deux", "trois"]   ==> 3
```

- son type est `([a] -> Int)` où `a` est une variable de type[1].

[1] une variable de type permet de désigner n'importe quel type.



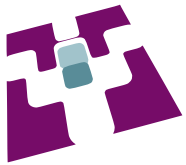
Définitions

- Primitives d'accès *head*, *tail* et de construction *(:)*

```
(:) :: a -> [a] -> [a]

head :: [a] -> a

tail :: [a] -> [a]
```

- **Exemples**

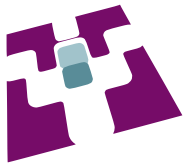
```
> head [1, 7, 2, 5]    ==> 1

> tail [1, 7, 2, 5]    ==> [7,2,5]

> head (tail [1, 7, 2, 5])    ==> 7

> tail (tail [1, 7, 2, 5])    ==> [2,5]

> head (tail (tail [1, 7, 2, 5]))    ==> 2
```



Définitions

- Propriété : si `xs` est une liste d'éléments de même type que celui de `x`, alors on a toujours :
  - `head (x:xs) = x`
  - `tail (x:xs) = xs`
  - `(head xs):(tail xs) = xs`
- **Exemples**

```
head [7, 4, 8] = head (7 : [4, 8]) = 7
tail [7, 4, 8] = tail (7 : [4, 8]) = [4, 8]

head "hello" = head ('h' : "ello") = 'h'
tail "hello" = tail ('h' : "ello") = "ello"
```

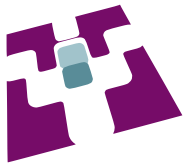


Pattern matching

- `[]` est le pattern (modèle) de la liste vide
- `(x:xs)` est le pattern (modèle) des listes non vides

Exemples

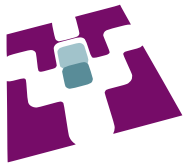
- `[1,3,2]` vs `(x:xs)` $\rightarrow$ `x=1`, `xs=[3,2]` car `[1,3,2]=1:[3,2]`
- `[3]` vs `(x:xs)` $\rightarrow$ `x=3`, `xs=[]` car `[3]=3:[]`
- `[1,3,2]` vs `(x:y:xs)` $\rightarrow$ `x=1`, `y=3`, `xs=[2]` car `[1,3,2]=1:3:[2]`
- `[1,1,2]` vs `(x:y:xs)` $\rightarrow$ `x=1`, `y=1`, `xs=[2]` car `[1,1,2]=1:1:[2]`
- `[3]` ne satisfait pas le pattern `(x:y:xs)`



Pattern matching

- `[]` est le pattern de la liste vide
- `(x:xs)` est le pattern des listes non vides
- `(x:xs)` est le pattern des listes ayant au moins un élément
- `(x:y:xs)` est le pattern des listes ayant au moins deux éléments

- Pour déterminer si une liste possède 2 premiers éléments identiques, peut-on utiliser le pattern `(x:x:xs)` ?
- Quelles listes représente le pattern `[x]` ?
- Quelles listes représente le pattern `[x, y]` ?

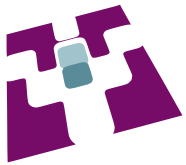


Traitement récursif de listes

Somme d'une liste d'entiers

- type de cette fonction : `[Int] -> Int`
- deux définitions équivalentes :

```
sum1 :: [Int] -> Int
sum1 l = if (l == [])
         then 0
         else (head l) + (sum1 (tail l))

-- pattern matching

sum3 :: [Int] -> Int

sum3 [] = 0
sum3 (x:xs) = x + (sum3 xs)
```

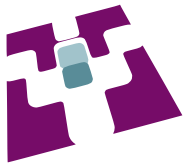


Traitement récursif de listes

Longueur d'une liste

- son type : `[a] -> Int`
- deux définitions équivalentes :

```
lg1 liste = if (liste == [])
            then 0
            else 1 + lg1 (tail liste)

-- pattern matching

lg3 [] = 0
lg3 (x:xs) = 1 + (lg3 xs)
```



Traitement récursif de listes

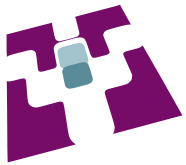
Longueur d'une liste

- son type : `[a] -> Int`
- pattern matching en utilisant 2 modèles de listes

```
lg3 [] = 0
lg3 (x:xs) = 1 + (lg3 xs)
```

- la variable `x` n'apparaît pas dans la partie droite de la 2nde équation, on peut remplacer `x` par l'attrape-tout noté `_`

```
lg4 [] = 0
lg4 (_:xs) = 1 + (lg4 xs)
```



Traitement récursif de listes

Concaténation de deux listes : opérateur infixe (++)

- exemples

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

- définition

```
append :: [a] -> [a] -> [a]

append []     ys = ys
append (x:xs) ys = x : (append xs ys)
```