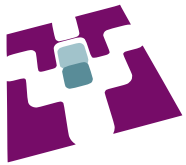
UNIVERSITÉ
CAEN
NORMANDIE

Autre Paradigme

Patrice Boizumault

Université de Caen - Normandie

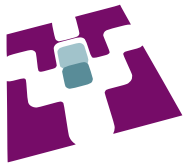
Lundi 27 janvier 2020



Séance #4

TP #2 : Décomposition en facteurs premiers d'un entier naturel

- $20 = 4 \times 5 = 2^2 \times 5^1$
  Représentation `Haskell` $\rightarrow$ `[(2,2),(5,1)]`
- $100 = 2^2 \times 5^2 \rightarrow$ `[(2,2),(5,2)]`
- $1024 = 2^{10} \rightarrow$ `[(2,10)]`
- $10164 = 2^2 \times 3 \times 7 \times 11^2$ `[(2,2),(3,1),(7,1),(11,2)]`
- Quid de la liste vide `[]` ?



Séance #4

Définition de types synonymes

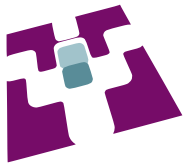
```
type Facteur = Int
type Exposant = Int
type Couple = (Facteur, Exposant)
type Decomposition = [Couple]

int2rep :: Int -> Decomposition

> int2rep 1024   ==> [(2,10)]
> int2rep 10164   ==> [(2,2),(3,1),(7,1),(11,2)]

rep2int :: Decomposition -> Int

> rep2int [(2,10)]   ==> 1024
> rep2int [(2,2),(3,1),(7,1),(11,2)]   ==> 10164
```



Séance #4

Définition de types synonymes

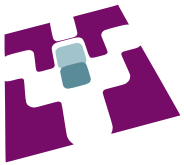
```
type Facteur = Int
type Exposant = Int
type Couple = (Facteur, Exposant)
type Decomposition = [Couple]

int2rep :: Int -> Decomposition

rep2int :: Decomposition -> Int

> :t (rep2int . int2rep)
(rep2int . int2rep) :: Int -> Int

> :t (int2rep . rep2int)
(int2rep . rep2int) :: Decomposition -> Decomposition
```



Séance #4

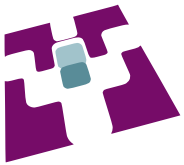
(Extrait de la documentation `Haskell`)

- *(`take n xs`) returns the prefix of `xs` of length `n`,*
  or `xs` itself if `(n > (length xs))`

- *(`drop n xs`) returns the suffix of `xs` after the first `n` elements,*
  or `[]` if `(n > (length xs))`

- ```
  > take 5 [2..20]   ==> [2,3,4,5,6]
  > drop 5 [2..15]   ==> [7,8,9,10,11,12,13,14,15]

  > take 10 [1..6]   ==> [1,2,3,4,5,6]
  > drop 10 [1..6]   ==> []
  ```
  ```



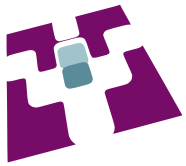
Séance #4

(Extrait de la documentation `Haskell`)

- *`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`*

```
> takeWhile (< 3) [1,2,3,4,1,2,3,4]   ==> [1,2]
> takeWhile (< 9) [1,2,3]   ==> [1,2,3]
> takeWhile (< 0) [1,2,3]   ==> []

> takeWhile (== 'a') "aazertyop"   ==> "aa"
> takeWhile (/= 'a') "aazaertyop"   ==> ""
> takeWhile (/= 't') "aazaertyop"   ==> "aazaer"
```



Séance #4

(Extrait de la documentation `Haskell`)

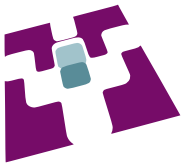
- *`(dropWhile p xs)` returns the suffix remaining after `(takeWhile p xs)`*

```
> takeWhile (< 3) [1,2,3,4,1,2,3,4]   ==> [1,2]
> dropWhile (< 3) [1,2,3,4,5,1,2,3]   ==> [3,4,5,1,2,3]

> takeWhile (< 9) [1,2,3]   ==> [1,2,3]
> dropWhile (< 9) [1,2,3]   == []

> takeWhile (< 0) [1,2,3]   ==> []
> dropWhile (< 0) [1,2,3]   == [1,2,3]

> dropWhile (== 'a') "aazertyop"   ==> "zertyop"
> dropWhile (== 'a') "aazaertyop"   ==> "zaertyop"
> dropWhile (/= 't') "aazaertyop"   ==> "tyop"
```



Séance #4

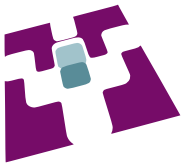
Typage

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

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


takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]

> :t takeWhile
takeWhile :: (a -> Bool) -> [a] -> [a]
```



Séance #4

- **`(pfactors n)` associe, à un entier `n`, la liste de ses facteurs premiers**

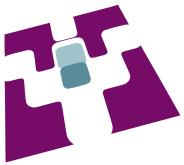
```
> pfactors 8   ==>  [2,2,2]
> pfactors 72   ==>  [2,2,2,3,3]
> pfactors 924   ==>  [2,2,3,7,11]
```

- **schéma de définition**[1]

```
pfactors :: ? -> ?

pfactors n = pfactors' n primes
   where pfactors' x (p:ps)
         | p > x           = ?
         | mod x p == 0    = ?
         | otherwise       = ?
```

[1] `primes` désigne la liste infinie des nombres premiers (cf question #7 du TP #2)



Séance #4

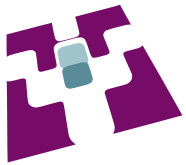
- **Exemples**

```
> pfactors 8   ==>  [2,2,2]
> pfactors 72   ==>  [2,2,2,3,3]
> pfactors 924   ==>  [2,2,3,7,11]
```

- **Définition complète utilisant une fonction auxiliaire**

```
pfactors :: Int -> [Int]

pfactors n = pfactors' n primes
   where pfactors' x (p:ps)
         | p > x           = []
         | mod x p == 0    = p : (pfactors' (div x p) (p:ps))
         | otherwise       = pfactors' x ps
```



Séance #4

- **`(prep xs)` détermine la décomposition d'une liste `xs` de facteurs premiers**

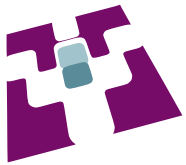
```
> prep [2,2,2]    ==>  [(2,3)]
> prep [2,2,2,3,3]   ==>  [(2,3),(3,2)]
> prep [2,2,3,7,11]    ==>  [(2,2),(3,1),(7,1),(11,1)]
```

- **définition**

```
prep :: [Int] -> Decomposition

prep [] = []

prep (x:xs) = (x, (length (takeWhile (==x) (x:xs))) )
              : (prep (dropWhile (==x) xs))
```



Séance #4

- **`(int2rep n)` détermine la décomposition associée à l'entier `n`**

```
> int2rep 72    ==>  [(2,3),(3,2)]

> pfactors 72    ==>  [2,2,2,3,3]
> prep [2,2,2,3,3]    ==>  [(2,3),(3,2)]

> int2rep 924    ==>  [(2,2),(3,1),(7,1),(11,1)]

> pfactors 924    ==>  [2,2,3,7,11]
> prep [2,2,3,7,11]    ==>  [(2,2),(3,1),(7,1),(11,1)]
```

- **définition**

```
int2rep :: Int -> Decomposition

int2rep = prep . pfactors
```