

Chapitre 2

1. Listes : fonctions primitives
2. Fonctions : « gardes »
3. Fonctions : appel par filtrage
4. Récursivité sur les listes

1. Listes

- Une liste est une suite homogène de valeurs (même type)
- Représentation:

[]	Liste vide
[x1 , x2 , ... , xn]	Liste à n éléments

- Type

[a] est le type des listes d'éléments de type a

l1 = [1,2,3] :: **[Int]**

l2 = ['a','b','c'] :: **[Char]**

l3 = [('x',12),('y',26),('z',9)] :: **[(Char,Int)]**

ages = [("Jean",23),("Aline",24),("Berthe »,19)] :: **[(String,Int)]**

Remarques

1) **String** = [Char]

```
> :t "abc"
```

```
"abc" :: [Char]
```

2) Segments

```
> [1..5]
```

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

```
> ['a'..'d']
```

```
"abcd"
```

3) Liste singleton : [x]

Ne pas confondre avec x

```
> :t 'a'
```

```
'a' :: Char
```

```
> :t ['a']
```

```
['a'] :: [Char]
```

- Fonctions primitives sur les listes

- Accès aux éléments

head l = « tête » de l = 1er élément

tail l = « queue » de l = le reste

```
> head [1..5]
```

```
1
```

```
> tail [1..5]
```

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

```
> head "abc"
```

```
'a'
```

```
> tail "abc"
```

```
"bc »
```

```
> (head "Sarko") == (head "Snake")
```

```
True
```

NB. Suppose l non vide

```
> head []
```

```
*** Exception: Prelude.head: empty list
```

- Fonctions primitives sur les listes

- Construction d'une liste

x : l = liste de premier élément (head) x
et de reste (tail) l

```
> 'a' : "bcd"
```

```
"abcd"
```

```
> 1 : [ 2 .. 5 ]
```

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

```
> (head [1..5]) : (tail [1..5])
```

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

NB. Associe à droite

```
> 1:2:[3..5]
```

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

- Fonctions primitives sur les listes

- Concatenation (append)

l1 ++ l2 = liste constituée des éléments de l1 suivie des éléments de l2

```
> [1..3]++[4..6]
```

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

```
> "I"++"love"++"Sarko"
```

```
"IloveSarko »
```

- Égalité : ==

```
>(tail [1..5]) == 2:3:4:[5]
```

```
True
```

- Premières fonctions (récursives)

```
-- deuxième élément, troisième
-- suppose l de taille adéquate
deuse, troise :: [a] -> a
deuse l = head (tail l)
troise l = head (tail (tail l))
```

```
-- longueur
long :: [Int] -> Int
long l = if l == [] then 0 else 1+(long (tail l))
```

```
-- Somme des éléments
somliste :: [Int] -> Int
somliste l = if l == [] then 0
             else (head l) + (somliste (tail l))
```

```
-- éléments supérieurs à N
lesSup :: Int -> [Int] -> [Int]
lesSup n l = if l == [] then []
             else if (head l) > n
                   then (head l):(lesSup n (tail l))
                   else (lesSup n (tail l))
```

2. Une autre conditionnelle = gardes

```
-- éléments supérieurs à N : version 2
lesSup2 :: Int -> [Int] -> [Int]
lesSup2 n l
  | l == []      = []
  | head l > n   = (head l):(lesSup n (tail l))
  | otherwise    = lesSup n (tail l)
```

- Problème : prendre les k premiers éléments d'une liste

```
> take 4 [1..10]  
[1,2,3,4]
```

- Solution 1 : avec if

```
-- take avec if  
mytake1 k l =  
    if k == 0 || l == [] then []  
    else (head l) : (mytake1 (k-1) (tail l))
```

- Solution 2 : avec Gardes

-- Take avec Gardes

```
mytake2 k l
  | k == 0      = []
  | l == []     = []
  | otherwise = (head l) : (mytake2 (k-1) (tail l))
```

- Gardes : Forme générale

```
fonc x1...xn
  | garde1 = Expr1
  | ...
  | garde k = Exprk
```

gardei = prédicat (fonction booléenne)
 Eventuellement : garde k == otherwise (tjs True)
 La première garde == True est sélectionnée

- Autre exemple : test d'appartenance

-- Avec if

```
cherche1 x l = if l == [] then False
               else if x == head(l) then True
               else cherche1 x (tail l)
```

-- Avec gardes

```
cherche2 x l
  | l == []      = False
  | x == head(l) = True
  | otherwise    = cherche2 x (tail l)
```

- Remarque : typage

```
cherche2 x l
  | l == []      = False
  | x == head(l) = True
  | otherwise    = cherche2 x (tail l)
```

```
> :t cherche2
(Eq a) => a -> [a] -> Bool
```

Il faut que l'égalité (==) sur

- (les éléments de type) **a**

soit définie. Voir plus loin. Ce sera le cas avec les types « de base » pour **a** : **Int**, **Bool**, **Char**...

- Sur des nombres

-- PGCD avec gardes

pgcd n p

| n > p = pgcd (n-p) p

| n == p = n

| n < p = pgcd p n

3. Appel par filtrage

- `cherche` : encore une solution

```
cherche3 x [] = False
cherche3 x (y:ys) = if x == y then True
                    else cherche3 x ys
```

```
-- ou, avec le joker « _ »
cherche4 _ [] = False
cherche4 x (y:ys) = if x == y then True
                    else cherche3 4 ys
```

On définit un résultat en fonction de la « forme des arguments d'appel »

- Notion de *pattern matching* (*filtrage*)

Entre :

Une expression	Un patron (pattern)	Liaisons
[1,2,3,4]	x : l	x = 1 l = [2,3,4]
[1,2,3,4]	x : y : l	x = 1 y = 2 l = [3,4]
[1,2,3,4]	1 : y : l	y = 2 l = [3,4]
[1,2,3,4]	2 : l	Echec
[1,2,3,4]	[]	Echec
[]	[]	Succes

Avec le « joker »: _

[1,2,3,4]	x : _	x=1
[1,2,3,4]	_ : _ : l	l = [3,4]

```
> let x:l = [1..5] in l
[2,3,4,5]
```

```
> let x:3:l = [1..5] in l
```

```
*** Exception: <interactive>:1:4-17: Irrefutable
      pattern failed for pattern x : 3 : l
```

- Fonctionnement : détail

```
foo pat1 pat2 ... patn = Expr
```

```
foo pat1' pat2' ... patn' = Expr'
```

```
foo pat1'' pat2'' ... patn'' = Expr''
```

...

- Les `pati` sont des « patrons » (*patterns*).
- Pour le moment, `listes` et `t-uples`. Ultérieurement : tous types définis par des constructeurs (chapitre 4)
- Un patron peut « matcher » un argument d'appel :

```
foo arg1 arg2 ... argn          pat1 filtre arg1 ....
```

Les variables du patron prennent alors leur valeur selon la structure de l'argument d'appel

- Le matching peut `échouer`
- La première expression pour laquelle les patrons matchent les arguments est sélectionnée

- Autres exemples

```
-- mytake par filtrage
```

```
mytake3 :: Int -> [a] -> [a]
```

```
mytake3 0 _ = []
```

```
mytake3 _ [] = []
```

```
mytake3 k (x : xs) = x : (mytake3 (k-1) xs)
```

```
-- recherche par filtrage et gardes
```

```
cherche5 _ [] = False
```

```
cherche5 x (y:ys)
```

```
    | x == y      = True
```

```
    | otherwise = cherche5 x ys
```

Exercices. Autre exemple : tuples et listes

`prem :: (a,b,c) -> a`

`??`

`deuxe :: (a,b,c) -> b`

`??`

`troixe :: (a,b,c) -> c`

`??`

4. Récursion sur les listes : principe

- Exemple : somme des éléments d'une liste

```
-- Version 1 (if)
somliste :: [Int] -> Int
somliste l = if l == [] then 0
             else (head l) + (somliste (tail l))
```

```
-- Version 2 (filtrage)
Somliste' :: [Int] -> Int
Somliste' [] = 0
Somliste' x:l' = x + (somme1 l')
```

```
> somliste [1..10]
55
```

Principe de récursion structurelle

- Une « liste de a » ($l :: [a]$) est :
 - Soit la liste vide $[]$
 - Soit la donnée $x : l'$
 - D'un élément x élément de type a (tête/head)
 - Et d'une liste l' liste d'éléments de type a (queue/tail)
- Une fonction $f :: [a] \rightarrow b$ est définie par
 - Une condition d'arrêt $f [] = v0$
 - Une clause récursive $f (x:l') = g\ x\ (f\ l')$

Combinant selon une certaine opération g la tête x et le résultat de f sur la queue $(f\ l')$

- Fonctionnement : suite des appels récursifs

```
somliste' [] = 0
```

```
somliste' x:l' = x + (somliste' l')
```

```
somliste' [1,2,3]          -- décomposition x:l' = [1,2,3]
```

⇓

```
1 + (somliste' [2,3])      -- décomposition x:l' = [2,3]
```

⇓

```
2 + (somliste' [3])        -- décomposition x:l' = [3]
```

⇓

```
3 + (somliste' [])        -- reconnaissance []
```

⇓

0

Résultat : $1 + 2 + 3 + 0 = 6$

A chaque étape, une opération à réaliser après l'appel récursif est
« empilée »

- Raisonnement

$$f [] = v0$$

$$f (x : l') = g \ x \ (f \ l')$$

- Si la donnée est [], la valeur de f est : v0
- Supposons que l'on sache calculer

$$\begin{array}{ll} f \ l' & = v' \\ \text{Alors } f (x : l') & = g \ x \ v' \end{array} \quad \text{-- donnée + petite}$$

$$\text{sommeliste}' \ [] = 0$$

$$\text{sommeliste}' (x : l') = x + (\text{somme1} \ l')$$

- Si la donnée est [], la somme vaut 0
- Supposons que l'on sache calculer

$$\begin{array}{ll} \text{sommeliste}' \ l' & = v' \\ \text{Alors } \text{sommeliste}' (x : l') & = x + v' \end{array} \quad \text{-- donnée + petite}$$