## Récursivité terminale

- Une fonction est récursive terminale si son dernier appel récursif ne contient *que l'appel récursif et ses paramètres*.
- L'intérêt de la récursion terminale est qu'elle ne comporte pas de phase de « remontée » : elle est donc plus efficace.
- L'autre intérêt est que certains compilateurs sauront optimiser un appel récursif terminal en le traduisant par une boucle.
- Le passage à la récursion terminale passe généralement par l'emploi d'une *fonction auxiliaire* et d'un *accumulateur*.
- Notre fonction factorielle n'était pas récursive terminale...

### Appel de facto 4 avec sa définition actuelle

```
facto 4
=> facto 4
=> 4 * facto 3
=> 4 * (3 * facto 2)
=> 4 * (3 * (2 * facto 1))
=> 4 * (3 * (2 * (1 * facto 0)))
=> 4 * (3 * (2 * (1 * 1)))
=> 4 * (3 * (2 * 1))
=> 4 * (3 * 2)
=> 4 * 6
=> 24
```

## Récursivité terminale

- Le principe va consister à « mémoriser » les étapes intermédiaires entre chaque appel récursif : *mémoïzation*.
- Soit *accu* le résultat du calcul de *facto (n - 1)*, le calcul de *facto n* consistera donc à calculer *accu = n * accu*.
- Il faut donc transmettre la valeur courante de *accu* entre chaque appel récursif.
- À la fin de la récursion, on renvoie simplement *accu* qui contiendra donc la valeur finale.

## Récursivité terminale

- On pose : *facto' n accu = facto n * accu*
- Lorsque *accu = 1*, on remarque que *facto n = facto' n 1*.
- Lorsque *accu* $\neq$ *1*, on remplace *facto n* par sa définition (en distinguant les différents cas) :

### Définition de facto' en fonction de facto pour accu $\neq$ 1

```
facto' 0 accu = facto 0 * accu
              = 1 * accu
              = accu

facto' n accu = facto n * accu
              = (n * facto (n - 1)) * accu
              = facto (n - 1) * (n * accu)
              = facto' (n - 1) (n * accu)
```

## Récursivité terminale

D'où la définition *récursive terminale* de la factorielle en Haskell :

### Définition de facto' en fonction de facto

```
facto :: Int -> Int
facto n = facto' n 1
    where
       facto' :: Int -> Int -> Int
       facto' 0 accu = accu
       facto' n accu = facto' (n - 1) (n * accu)
```

L'appel *facto 4* devient alors :

### Appel de facto 4 (version récursive terminale)

```
facto 4
=> facto 4
=> facto' 4 1
=> facto' 3 4
=> facto' 2 12
=> facto' 1 24
=> facto' 0 24
=> 24
```

# La même chose avec Maybe

## Factorielle terminale avec Maybe

```
facto :: Int -> Maybe Int
facto n
   | n < 0     = Nothing
   | otherwise = facto' n 1
     where
        facto' :: Int -> Int -> Maybe Int
        facto' 0 acc = Just acc
        facto' n acc = facto' (n - 1) (acc * n)
```

## Fonctions auxiliaires

- Pour résoudre un problème, on le décompose en problèmes plus simples.
- En programmation fonctionnelle (comme en programmation impérative), on aura donc souvent besoin de *fonctions auxiliaires* :
  - Parce qu'on ne peut pas faire autrement : il semble difficile d'écrire $f(n,k) = 1^k + 2^k + \cdots + n^k$ sans écrire la fonction $x^y$...
  - Parce qu'on aura besoin de la même fonction ailleurs.
  - *Parce que cela peut beaucoup améliorer les performances.*
- Problème classique : sous sa forme naïve, la fonction de Fibonacci passe son temps à recalculer ce qu'elle a déjà calculé :

```
fibo :: Int -> Int
fibo 0 = 0
fibo 1 = 1
fibo n = fibo (n - 1) + fibo (n - 2)
```

## Fonctions auxiliaires

On définit la fonction $g(n)$ qui renvoie le couple des deux valeurs consécutives de Fibonacci pour $n$ :

$$g(n) = (fibo(n), fibo(n+1)) \tag{1}$$

On a donc :

$$\begin{aligned} g(n+1) &= (fibo(n+1), fibo(n+2)) \\ &= (fibo(n+1), fibo(n+1) + fibo(n)) \end{aligned} \tag{2}$$

Si l'on pose $g(n) = (a, b)$, on a d'après (1) et (2) :

$$g(n+1) = (b,\ a+b) \tag{3}$$

## Fonctions auxiliaires

D'où l'écriture de *fibo*, récursive terminale :

```
g :: Int -> (Int, Int)
g 1 = (1, 1)
g n = (b, a + b)
  where (a, b) = g (n - 1)

fibo n = a
  where (a, _) = g n
```

Désormais, *fibo n* ne s'exécute plus qu'en $n + 1$ étapes au lieu de $2^n$ !

## Fonctions auxiliaires

On peut aussi mémoriser les étapes intermédiaires dans les paramètres de l'appel récursif en utilisant deux accumulateurs que l'on additionne à chaque appel.

```
fibo n = loop n 0 1
  where
   loop :: Int -> Int -> Int -> Int
   loop 0 avder der = avder
   loop n avder der = loop (n - 1) der (der + avder)
```

## Fonctions auxiliaires

Pour éviter le recalcul des valeurs, on peut également utiliser la technique de « mémoïsation » : on stocke dans une liste (voir plus loin) les valeurs déjà calculées et on les récupère lorsqu'on en a besoin :

```
fibo_memo n = liste_fibos !! n
   where liste_fibos = map f [0 ..]
         f 0 = 0
         f 1 = 1
         f x = liste_fibos !! (x - 2) + liste_fibos !! (x - 1)
```

Comparaison des deux fonctions (la version naïve et celle-ci) :

```
ghci> :set +s
ghci> fibo_naive 35
9227465
(19.97 secs, 7,792,052,696 bytes)
ghci> fibo_memo 35
9227465
(0.00 secs, 90,672 bytes)
```

Sans commentaires...

# Tuples et listes

## Tuples

- Un tuple est un objet composé de plusieurs éléments de types quelconques. Il est donc *hétérogène*.
- Un tuple contient un *nombre déterminé* d'éléments.
- Le type d'un tuple est indiqué entre parenthèses, par les types de ses éléments : le tuple `("un", 2, "trois")` est du type `(String, Int, String)` et `(1, ('a', 'b'), 3.14)` est du type `(Int, (Char, Char), Float)`.
- En résumé, le type tuple $(t_1, t_2, \ldots, t_n)$ est formé des valeurs $(v_1, v_2, \ldots, v_n)$, avec $v_1$ `::` $t_1 \ldots v_n$ `::` $t_n$.
- Haskell fournit les fonctions *fst* et *snd* qui renvoient, respectivement, le premier et le second élément d'une *paire* (un tuple de deux éléments).

### Exemple

```
ghci> :t properFraction
properFraction :: (Integral b, RealFrac a) => a -> (b, a)
ghci> properFraction 5.6
(5,0.5999999999999996)
ghci> snd $ properFraction 5.6  -- identique à snd (properFraction 5.6)
0.5999999999999996
```

# Tuples

De nombreuses fonctions Haskell renvoient des tuples ou prennent des tuples en paramètres :

## Exemples

```
ghci> import Data.List
ghci> :t partition
partition :: (a -> Bool) -> [a] -> ([a], [a])
gchi> let (ppetits, pgrands) = partition (< 5) [1..10]
ghci> ppetits
[1,2,3,4]
ghci> pgrands
[5,6,7,8,9,10]
ghci> :t zip
zip :: [a] -> [b] -> [(a, b)]
ghci> zip [1..5] ['a'..'e']
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e')]
```

## Tuples

Pour extraire les valeurs d'un tuple de plus de deux éléments, on peut utiliser le pattern-matching :

```
ghci> type Nom      = String
ghci> type Prenom   = String
ghci> type Age      = Int
ghci> type Personne = (Nom, Prenom, Age)

ghci> let nom (n, _, _)    = n
ghci> let prenom (_, p, _) = p
ghci> let age (_, _, a)    = a

ghci> nom ("Dupont", "Michel", 20)
"Dupont"
```

### Remarque

Dans ce cas précis, il serait plus judicieux d'utiliser des *enregistrements* à la place des tuples (voir plus loin pour une explication de *data*) :

```
ghci> data Personne = Personne { nom :: String, prenom :: String, age :: Int }

ghci> let lui = Personne "Dupont" "Michel" 20
ghci> nom lui
"Dupont"
ghci> age lui
20
```

# Tuples : remarques

Attention à la différence entre :

```
-- Fonction prenant UN tuple de deux Int en paramètre et renvoyant un Int.
truc :: (Int, Int) -> Int
```

Et :

```
-- Fonction prenant deux Int en paramètre et renvoyant un Int.
machin :: Int -> Int -> Int
```

Les appels ne seront pas les mêmes :

```
ghci> truc (4, 6)     -- On ne passe qu'UN paramètre
...
ghci> machin 4 6      -- On passe DEUX paramètres
```

## Axiomatique des listes

- Les listes sont des groupements d'objets qui ne peuvent être accédés que *dans un ordre séquentiel*. La liste est *la* structure fondamentale de la programmation fonctionnelle.
- Pour construire une liste, il faut :
  - Une liste vide
  - Un moyen d'ajouter un élément à une liste.
- On part donc de la liste vide, puis on obtient une liste à un élément, à deux éléments, etc.

## Axiomatique des listes

1. La liste vide est une liste.
2. Si $x$ est un élément de la liste $li$, alors $cons(x, li)$ est une liste.
3. Il n'existe pas de couple $(x, li)$ tel que $cons(x, li) = vide$.
4. *cons* est *injective* : si $cons(x, li) = cons(x', li')$, alors forcément $x = x'$ et $li = li'$.
5. Si une propriété $P$ est vraie pour la liste vide et si $P(li) \rightarrow P(cons(x, li))$ pour tout $x$ de type $T$ et toute liste $li$ d'éléments de type $T$, alors $P$ est vraie pour toute liste d'éléments de type $T$ (*principe d'induction*).

**Remarque**

*La liste notée [2, 1, 5, 6] est donc le résultat d'applications successives de cons à partir de la liste vide :* $cons(2, cons(2, cons(5, cons(6, Vide))))$

## Listes Haskell

- Une liste est *homogène* : tous les éléments doivent être d'un même type connu (y compris tuples et listes).
- Le type d'une liste d'éléments de type *T* se note *[T]*.
- La liste vide, notée *[]*, ne contient aucun élément et appartient à toutes les listes.
- La liste 10, 20, 30, 40 est notée [10, 20, 30, 40].
- Le nombre d'éléments d'une liste est *quelconque*, il peut être connu par la fonction prédéfinie *length* (voir le tableau des fonctions prédéfinies).
- La notation *[a]* est spéciale : elle signifie « une liste d'éléments de type *a* quelconque » (on peut évidemment remplacer *a* par une autre lettre, mais l'on utilise traditionnellement *a* et *b*). Cette *variable de type* sera liée au type réel de la liste concernée.

## Listes Haskell

Une notation spéciale permet de construire des listes d'*éléments consécutifs* d'un type quelconque instance de *Enum* (*nombres*, *caractères ou chaînes*, notamment) :

- *[deb..fin]* est une liste dont les éléments vont de *deb* à *fin* (en utilisant *succ* pour trouver l'élément suivant).
- *[deb, suivant..fin]* est un liste dont les éléments vont de *deb* à *fin* par pas de *suivant - deb* (les deux premiers éléments de la liste sont donc *deb* et *suivant* et les autres respectent cet écart).

**Exemples :**

```
ghci> [2..7]                    <- progresse par pas de 1
[2,3,4,5,6,7]
ghci> [2, 4..7]                 <- progresse par pas de 4-2 = 2
[2,4,6]
ghci> [7..2]                    <- progresse par pas de 1
[]
ghci> [7, 6..2]                 <- progresse par pas de 6-7 = -1
[7,6,5,4,3,2]
ghci> [3.1..7.0]
[3.1,4.1,5.1,6.1,7.1]           <- le dernier est le plus proche de 7.0
ghci> ['a'..'m']
"abcdefghijklm"                 Le type String = [Char]
ghci> ['a', 'e'..'m']
"aeim"
```

## Listes Haskell

- La fonction de construction (appelée *cons* dans le cours) est implémentée par l'opérateur *:* (deux-points). La liste *[1, 3, 6, 2]* est donc une représentation de *1:3:6:2:[]*.
- Cet opérateur sert également d'*opérateur de déconstruction* lorsqu'il est utilisé dans un pattern-matching :
  - *(x:xs)* est un motif représentant une liste ayant au moins un élément qui sera lié à *x*, suivi d'une liste qui sera liée à *xs*.
  - *(x:y:xs)* est un motif représentant une liste ayant au moins *deux* éléments, liés respectivement à *x* puis *y*, suivis d'une liste liée à *xs*.

### Fonction longueur d'une liste quelconque

```
let (x:y:z:t:u) = [1..7]          -- x vaudra 1 et u vaudra [5, 6, 7]

longueur :: [a] -> Int
longueur [] = 0
longueur (_:xs) = 1 + longueur xs
```

### Remarque

*Dans longueur, le premier élément de la liste est noté _ car il n'a aucune importance pour le calcul. C'est une ≪ variable muette ≫.*

## Listes Haskell

- Une liste Haskell peut être *infinie* : *[1..]* est la liste de tous les entiers à partir de 1. Cette liste est évaluée *paresseusement* (les différentes valeurs ne sont calculées que lorsqu'elles sont nécessaires au calcul).
- Le prélude fournit un grand nombre de fonctions sur les listes. D'autres nécessitent l'importation du module *Data.List*.
- Les fonctions les plus connues sont : *:*, *++*, *!!*, *concat*, *head*, *last*, *tail*, *init*, *replicate*, *take*, *takeWhile*, *drop*, *dropWhile*, *splitAt*, *reverse*, *zip* et *unzip*. Toutes ces fonctions sont *polymorphes* (ou *génériques*).
- Les fonctions *and*, *or*, *all* et *any* en revanche, ne s'appliquent qu'à des listes de booléens (*[Bool]*).
- Les fonctions *product* et *sum* ne s'appliquent qu'à des listes de nombres (*Num a =¿ [a])*

**Remarque**

*Attention aux fonctions head et tail car elles ne fonctionnent pas avec des listes vides.*

## Listes Haskell

- Pour importer le module *Data.List* dans `ghci`, il suffit de faire *import Data.List*.
- Pour importer le module *Data.List* dans un fichier script, il faut placer au début de celui-ci la directive *import Data.List*.
- La commande *:browse nomModule* de `ghci` affiche la liste des fonctions du module *nomModule*, avec leurs signatures.
- La commande *:t nomFonction* affiche la signature de la fonction indiquée.
- Si vous avez installé *hoogle*, la commande *hoogle -i nomFonction* affiche la documentation de la fonction (la page `https://hoogle.haskell.org/` offre une interface web à *hoogle*).

## Listes en intension

- Par opposition à *liste en extension*, on parle de *liste en intenSion* : une liste n'est plus décrite par la liste de ses éléments mais en expliquant comment *produire* ses valeurs.
- Les anglais utilisent le terme de *comprehension lists*.

### Exemples

```
[ 2 * n | n <- [1..50] ]              -- liste de tous les entiers entre 2 et 100
[ even n | n <- [2, 11, 3, 4, 12] ]   -- produit [True,False,False,True,True]
[ n | n <- [1..50], even n, n >= 12 ]  -- liste des nombres pairs entre 12 et 50
```

## Listes en intension

- La partie *n ¡- liste* est appelée *partie génératrice* car c'est elle qui produit les éléments du résultat à partir de la liste de départ (*n* prendra successivement toutes les valeurs de *liste*).
- Le symbole *¡-* représente le $\in$ (≪ appartient ≫) des ensembles.
- La partie génératrice peut être suivie d'un ou plusieurs tests que doivent vérifier les éléments produits. Ces tests sont séparés par des virgules.
- La partie à gauche du symbole | (≪ tel que ≫) est ensuite appliquée aux éléments générés pour produire les éléments de la liste résultat.
- La partie à droite de | peut contenir une ou plusieurs parties génératrices, des tests et des liaisons locales (avec *let*).

### Exemples

```
ghci> [m + n | (m, n) <- [(1,2),(3,4),(5,6)]]
[3,7,11]

ghci> [z | x <- [1..10], y <- [1..x], let z = x + y, z > 10]
[6,6,7,8,6,7,8,9,10]
```

# Listes en intension

## Exemples

```
crible :: [Int] -> [Int]
crible [] = []
crible (x:xs) = x : crible [n | n <- xs, n `mod` x /= 0]

premiers :: Int -> [Int]
premiers lim = crible [2..limite]

fibo :: [Int]
fibo = 0 : 1 : [ x + y | (x, y) <- zip fibo (tail fibo) ]
```

Jeu : Comparez avec votre langage favori...