Listes

La liste est une structure de données fondamentale en programmation fonctionnelle.

Une liste ayant comme éléments $x_1, \ldots, x_n$ s'écrit $[x_1, \ldots, x_n]$.

Exemples :

$$
\begin{aligned}
&\textbf{val } \mathit{fruit} &&= [\text{"apples"}, \text{"oranges"}, \text{"pears"}]\\
&\textbf{val } \mathit{nums} &&= [1, 2, 3, 4]\\
&\textbf{val } \mathit{diag3} &&= [[1, 0, 0], [0, 1, 0], [0, 0, 1]]\\
&\textbf{val } \mathit{empty} &&= [\,]
\end{aligned}
$$

Notez la similarité avec l'initialisation d'un tableau en C ou en Java.

Cependant il y a deux différences importantes entre listes et tableaux.

1. Les listes sont immuables – les éléments d'une liste ne peuvent être changés.
2. Les listes sont récursives alors que les tableaux sont plats.

Le type liste

Comme les tableaux, les listes sont *homogènes* : les éléments d'une liste doivent tous avoir le même type.

Le type d'une liste avec éléments de type T s'écrit $\mathit{List}[T]$ (à comparer avec $[\,]T$ pour le type des tableaux d'éléments de type T en C ou Java).

Par ex.

$$
\begin{aligned}
&\textbf{val } \mathit{fruit} : \mathit{List}[\mathit{String}] &&= [\text{"apples"}, \text{"oranges"}, \text{"pears"}]\\
&\textbf{val } \mathit{nums} : \mathit{List}[\mathit{Int}] &&= [1, 2, 3, 4]\\
&\textbf{val } \mathit{diag3} : \mathit{List}[\mathit{List}[\mathit{Int}]] &&= [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
\end{aligned}
$$

Constructeurs de listes

Toutes les listes sont construites à partir de :

- la liste vide *Nil*.
- l'opération de construction $x :: xs$ qui retourne une nouvelle liste avec premier élément x, suivi des éléments de xs.

C-à-d :

$$
\begin{aligned}
&\mathit{fruit} &&= \text{"apples"} :: (\text{"oranges"} :: (\text{"pears"} :: \mathit{Nil}))\\
&\mathit{nums} &&= 1 :: (2 :: (3 :: (4 :: \mathit{Nil})))\\
&\mathit{empty} &&= \mathit{Nil}
\end{aligned}
$$

Convention : L'opérateur '::' associe à gauche. $A :: B :: C$ s'interprète comme $A :: (B :: C)$.

Il s'ensuit qu'on peut omettre les parenthèses dans la définition ci-dessus. Par ex.

$$\mathit{nums} \quad = \quad 1 :: 2 :: 3 :: 4 :: \mathit{Nil}$$

Opérations sur les listes

Toutes les opérations sur les listes peuvent s'exprimer en termes des trois opérations suivantes :

head	retourne le premier élément d'une liste
tail	retourne la liste composée de tous les éléments sauf le premier
isEmpty	retourne *True* ssi la liste est vide

Ces opérations sont définies comme méthodes des objets de type liste. Par ex.

$$
\begin{aligned}
&\mathit{fruit.head} = \text{"apples"}\\
&\mathit{fruit.tail.head} = \text{"oranges"}\\
&\mathit{diag3.head} = [1, 0, 0]\\
&\mathit{empty.head} \quad \rightarrow \quad (\mathit{Exception} \text{ "head of empty list"})
\end{aligned}
$$

Exemple

Supposons qu'on veuille trier une liste de nombres par ordre croissant :

- Une manière de trier la liste [7, 3, 9, 2] est de trier la queue [3, 9, 2] pour obtenir [2, 3, 9].
- Il s'agit ensuite d'insérer la tête 7 à la bonne place pour obtenir le résultat [2, 3, 7, 9].

Cette idée décrit le *Tri par Insertion* :

```
def isort(xs: List[Int]) =
  if (xs.isEmpty) Nil
  else ins(xs.head, isort(xs.tail))
```

Quelle est une implantation possible de la fonction manquante *ins* ?

Quelle est la complexité du tri par insertion ?

Fonctions sur les listes

En utilisant les deux constructeurs *Nil* et ::, et les trois méthodes *head*, *tail*, et *isEmpty*, on peut maintenant formuler d'autres fonctions communes sur les listes.

La fonction longueur

length(xs) doit retourner le nombre d'éléments de *xs*. Elle est définie comme suit.

```
def length(xs: List[String]) =
  if (xs.isEmpty) 0
  else 1 + length(xs.tail)
```

```
> length(nums)
4
```

Problème : on ne peut appliquer *length* que sur des listes de String.

Comment peut-on formuler la fonction de telle manière qu'elle soit applicable à toutes les listes ?

Polymorphisme

Idée : Passer le type des éléments comme paramètre additionnel (*de type*) à *length*.

```
def length[a](xs: List[a]) =
  if (xs.isEmpty) 0
  else 1 + length(xs.tail)
```

```
> length[Int](nums)
4
```

Syntaxe :

- On écrit les paramètres de type, formels ou effectifs, entre crochets. Par ex. [a], [Int].
- On peut omettre les paramètres de type effectifs quand ils peuvent être inférés à partir des paramètres de la fonction et du type du résultat (ce qui est généralement le cas).

Dans notre exemple, on aurait aussi pu écrire :

```
length(nums)      /* [Int] inferred since nums: List[Int] */
```

Cependant, on ne peut pas omettre les paramètres de type formels :

```
> def length(x: a) = ...
console: not found: a
def length(x: a) = ...
              ^
```

Les fonctions qui prennent des paramètres de type sont dites *polymorphiques*.

Ce mot signifie "plusieurs-formes" en Grec : c-à-d que la fonction peut être appliquée à des arguments de différents types.

Fonction *reverse*

Voici une fonction qui renverse les éléments d'une liste.

```
def reverse[a](xs: List[a]) = {
  if (xs.isEmpty) []
  else reverse(xs.tail) ::: [xs.head]
}
```

Q : Quelle est la complexité de *reverse* ?

R : $n + (n-1) + ... + 1 = n(n+1)/2$ où n est la longueur de *xs*.

Peut-on mieux faire ? (à résoudre plus tard).

Parenthèse : les exceptions

Il existe une fonction prédéfinie *error*, qui met fin au programme avec un message d'erreur donné.

Elle est définie comme suit.

```
def error[a](msg: String): a =
  java.lang.RuntimeException(msg).throw
```

Notez que *error* est polymorphique – elle est déclarée retourner un argument de type quelconque.

Bien sûr, *error* ne retourne en fait pas du tout.

Le polymorphisme de *error* est utilisé ici pour la rendre utilisable dans tous les contextes.

Fonctions *last* et *init*

La méthode *head* retourne le premier élément d'une liste. On peut écrire une fonction qui retourne le dernier élément d'une liste de la manière suivante.

```
def last[a](xs: List[a]) = {
  if (xs.isEmpty) error("last of empty list")
  else if (xs.tail.isEmpty) xs.head
  else last(xs.tail)
}
```

Exercice : Ecrire une fonction *init* qui retourne tous les éléments d'une liste sauf le dernier (autrement dit, *init* et *tail* sont complémentaires).

```
def init[a](xs: List[a]) = {
  if (xs.isEmpty) error("init of empty list")
  else if (xs.tail.isEmpty) ?
  else ?
}
```


Concaténer des listes

L'opérateur :: est asymétrique – il s'applique à un élément de liste et à une liste.

Il existe aussi un opérateur ::: qui *concatène* deux listes.

```
> [1, 2] ::: [3, 4]
[1, 2, 3, 4]
```

::: peut être défini en termes d'opérations primitives. Ecrivons une fonction équivalente

```
def concat[a](xs: List[a], ys: List[a]): List[a] =
  if (xs.isEmpty)
    ?
  else
    ?
```

Q : Quelle est la complexité de *concat* ?

Les opérateurs Cons et Concat

Les opérateurs commençant par ':' ont un traitement spécial dans Scala.

- Tous les opérateurs de ce type sont associatifs à droite. Par ex.

$$x + y + z = (x + y) + z \qquad \textit{mais} \qquad x :: y :: z = x :: (y :: z)$$

- Tous les opérateurs de ce type sont traités comme méthodes de leur opérande droite. Par ex.

$$x + y = x.+(y) \qquad \textit{mais} \qquad x :: y = y.::(x)$$

  (Notez cependant que les expressions opérandes continuent à être évaluées de gauche à droite. Donc, si d et e sont des expressions, alors leur expansion est :

$$d :: e = (\textbf{val}\ x = d;\ e.::(x))$$

Autres méthodes de List

Les fonctions couvertes jusqu'ici existent toutes comme méthodes de la classe *List*. Par exemple :

```
abstract class List[a] with {
  abstract def head: a
  abstract def tail: List[a]
  abstract def isEmpty: Boolean

  def length = if (isEmpty()) 0 else 1 + tail.length

  def init: List[a] =
    if (isEmpty) error("init of empty list")
    else if (tail.isEmpty) Nil
    else head :: tail.init
  ...
}
```


Classes *Nil* et *Cons*

Ces classes sont définies comme suit.

```
class Nil[a] extends List[a] with {
  def head: a = error("head of empty list")
  def tail: List[a] = error("tail of empty list")
  def isEmpty: Boolean  = True
}

class Cons[a](x: a, xs: List[a]) extends List[a] with {
  def head: a = x
  def tail: List[a] = xs
  def isEmpty: Boolean = False
}
```


La classe List

List n'est pas un type primitif dans Scala ; il peut être défini par une classe. Voici une implantation possible.

```
abstract class List[a] with {
  abstract def head: a
  abstract def tail: List[a]
  abstract def isEmpty: Boolean
}
```

Notez que *List* est une classe paramétrique.

Toutes les méthodes dans la classe *List* sont abstraites. Les implantations de ces méthodes se trouvent dans deux sous-classes concrètes :

- *Nil* pour les listes vides.
- *Cons* pour les listes non vides.

La conception la plus flexible consiste à rendre la fonction *sort* polymorphique, et à passer l'opération de comparaison désirée comme paramètre additionnel. Par ex. :

```
def msort[a](less: (a, a)Boolean)(xs: List[a]): List[a] = {
  val n = xs.length/2
  if (n == 0) xs
  else {
    def merge(xs1: List[a], xs2: List[a]): List[a] = ...
    val xs1 = xs.take(n)
    val xs2 = xs.drop(n)
    merge(msort(less)(xs1), msort(less)(xs2))
  }
}
```

Exercice : Définir la fonction *merge*. Voici deux cas de test.

```
merge([1, 3], [2, 4]) = [1, 2, 3, 4]
merge([1, 2], []) = [1, 2]
```


Trier les listes plus rapidement

Comme exemple non trivial, concevons une fonction de tri des éléments d'une liste qui soit plus efficace que le tri par insertion.

Un bon algorithme pour cela est le *tri fusion*. L'idée est la suivante.

- Si la liste est composée de zéro ou un élément, elle est déjà triée.
- Sinon,
  1. Séparer la liste en deux sous-listes chacune contenant environ la moitié des éléments de la liste originale.
  2. Trier les deux sous-listes.
  3. Fusionner les deux sous-listes triées en une seule liste triée.

Pour implanter cela, nous devons encore spécifier

- Quel est le type des éléments à trier ?
- Comment comparer deux éléments ?

Encore des méthodes pour List

La méthode *take*(n) retourne les n premiers éléments de sa liste (ou la liste elle-même si elle est plus courte que n).

La méthode *drop*(n) retourne sa liste sans les n premiers éléments.

La méthode *at*(n) retourne le n-ième élément d'une liste.

Elles sont définies comme suit :

```
abstract class List[a] with {
  ...
  def take(n: Int): List[Int] =
    if (n == 0) [] else head :: tail.take(n - 1)

  def drop(n: Int): List[Int] =
    if (n == 0) this else tail.drop(n - 1)

  def at(n: Int) = drop(n).head
}
```


La définition de :: et ::: est maintenant triviale :

```
abstract class List[a] with {
  ...
  def ::(x: a) = Cons(x, this)

  def :::(prefix: List[a]) : List[a] =
    if (prefix.isEmpty) this
    else prefix.head :: (prefix.tail ::: this)

    /* or, equivalently:
         prefix.tail.:::(this).::(prefix.head)
    */
```

Voici un exemple d'utilisation de *msort*.

```
> def less(x: Int, y: Int) = x < y
> mort(less)([5, 7, 1, 3])
[1, 3, 5, 7]
```

La définition de *msort* est currifiée, pour faciliter sa spécialisation par des fonctions de comparaison particulières.

```
> val intSort = msort(less)
> val reverseSort = msort(x: Int, y: Int ⇒ x > y)
> intSort([6, 3, 5, 5])
[3, 5, 5, 6]
> reverseSort([6, 3, 5, 5])
[6, 5, 5, 3]
```

Complexité:

La complexité de *msort* est $O(n \log n)$.

Cette complexité ne dépend pas de la distribution initiale des éléments dans la liste.

Schémas récurrents de calcul

- Les exemples ont montré que les fonctions sur les listes ont souvent des structures similaires.
- On peut identifier plusieurs schémas récurrents comme
  - Transformer chaque élément d'une liste d'une certaine façon.
  - Extraire d'une liste tous les éléments satisfaisant un critère.
  - Combiner les éléments d'une liste en utilisant un opérateur.
- Les langages fonctionnels permettent aux programmeurs d'écrire des fonctions génériques qui implantent des schémas comme ceux-ci.
- Ces fonctions sont des *fonctions d'ordre supérieur* qui prennent en argument une transformation ou un opérateur.

Application d'une fonction aux éléments d'une liste

Une opération commune est de transformer chaque élément d'une liste et de retourner ensuite la liste des résultats.

Par exemple, pour multiplier chaque élément d'une liste par un même facteur.

```
def scaleList(xs: List[Double], factor: Double): List[Double] =
  if (xs.isEmpty) []
  else xs.head * factor :: scaleList(xs.tail, factor)
```

Ce schéma est généralisable par la méthode *map* dans *List*:

```
abstract class List[a] with {
  ...
  def map[b](f: (a)b): List[b] =
    if (isEmpty) []
    else f(head) :: tail.map(f)
}
```


En utilisant *map*, *scaleList* peut s'écrire de manière plus concise.

```
def scaleList(xs: List[Double], factor: Double) =
  xs map (x ⇒ x * factor)
```

Exercice : On considère une fonction qui calcule le carré de chaque élément d'une liste et retourne le résultat. Compléter les deux définitions équivalentes suivantes de *squareList*.

```
def squareList(xs: List[Int]): List[Int] =
  if (xs.isEmpty) ??
  else ??

def squareList(xs: List[Int]): List[Int] =
  xs map ??
```

Filtrage

Une autre opération commune sélectionne dans une liste tous les éléments remplissant une condition donnée. Par exemple :

```
def posElems(xs: List[Int]): List[Int] =
  if (xs.isEmpty) []
  else if (xs.head > 0) xs.head :: posElems(xs.tail)
  else posElems(xs.tail)
```

Ce schéma est généralisable par la méthode *filter* dans *List* :

```
abstract class List[a] with {
  ...
  def filter(p: (a)Boolean): List[a] =
    if (isEmpty) []
    else if (p(head)) head :: tail.filter(p)
    else tail.filter(p)
```


En utilisant *filter*, *posElems* peut s'écrire de manière plus concise :

```
def posElems(xs: List[Int]): List[Int] =
  xs filter (x ⇒ x > 0)
```


Reduce et Fold

Une autre opération commune est de combiner les éléments d'une liste avec un opérateur donné.

Par exemple :

$$sum([x_1, ..., x_n]) = 0 + x_1 + ... + x_n$$
$$product([x_1, ..., x_n]) = 1 * x_1 * ... * x_n$$

On peut implanter cela par le schéma récursif habituel :

```
def sum(xs: List[Int]): Int =
  if (xs.isEmpty) 0 else xs.head + sum(xs.tail)

def product(xs: List[Int]): Int =
  if (xs.isEmpty) 1 else xs.head * product(xs.tail)
```


La méthode générique *reduce* insère une opérateur binaire donné entre deux noeuds adjacents.

Par ex.

$$[x_1, ..., x_n].reduce(op) = (...(x_1\ op\ x_2)\ op\ ...)\ op\ x_n$$

Maintenant on peut écrire plus simplement :

```
def sum(xs: List[Int])     = (0 :: xs).reduce(x, y ⇒ x + y)
def product(xs: List[Int]) = (1 :: xs).reduce(x, y ⇒ x * y)
```

```
def reduceRight(op: (a, a)a): a =
  if (isEmpty) error("reduce of empty list")
  else if (tail.isEmpty) head
  else op(head, tail.reduceRight(op))

def foldRight[b](op: (a, b)b)(z: b): b =
  if (isEmpty) z
  else op(head, tail.foldRight(op)(z))
```

Pour les opérateurs associatifs *op*, *fold* et *foldRight* sont équivalents (même s'il peut y avoir une différence d'efficacité).

Mais parfois, seul l'un des deux opérateurs est approprié ou a le bon type.

Exemple : Voici une formulation alternative de *concat* :

```
def concat[a](xs: List[a], ys: List[a]): List[a] = {
  def cons(x: a, xs: List[a]): List[a] = x :: xs
  xs.foldRight(cons)(ys)
}
```

Ici il n'est pas possible de remplacer *foldRight* par *fold* (pourquoi ?).

FoldRight et ReduceRight

Les applications de *fold* et *reduce* se développent en arbres qui penchent vers la gauche :

Elles ont deux fonctions duales, foldRight et reduceRight, qui produisent des arbres qui penchent vers la droite. C-à-d :

$$
\begin{aligned}
[x_1, ..., x_n].reduceRight(op) &= x_1 \; op \; (\; ... \; (x_{n-1} \; op \; x_n)...) \\
[x_1, ..., x_n].foldRight(op)(acc) &= x_1 \; op \; (\; ... \; (x_n \; op \; acc)...)
\end{aligned}
$$

Elles sont définies comme suit.

fold prend comme paramètre additionnel un *accumulateur* *acc*, qui est retourné par les listes vides.

C-à-d,

$$[x_1, ..., x_n].fold(op)(acc) = (...(acc \; op \; x_1) \; op \; ...) \; op \; x_n$$

sum et *product* peuvent alors être définies alternativement comme suit.

```
def sum(xs: List[Int])     = xs.fold(x, y => x + y)(0)
def product(xs: List[Int]) = xs.fold(x, y => x * y)(1)
```


Implantation de Reduce

Comment *reduce* peut-il être implanté ?

```
abstract class List[a] with {
  ...
  def reduce(op: (a, a): a): a = {
    if (isEmpty) error("reduce of empty list")
    else tail.fold(op)(head)
  }
  def fold[b](op: (b, a): b)(acc: b): b = {
     if (isEmpty) acc
     else tail.fold(op)(op(acc, head))
  }
}
```

La fonction *reduce* est définie en termes d'une autre fonction souvent utile, *fold*.

Annexe : La classe List

Voici une implantation de la classe *List* dans Scala (en lien avec la discussion qu'on en a faite jusqu'ici dans le cours).

```
import Boolean

def error[a](x: String):a = (new java.lang.RuntimeException(x)).throw

abstract class List[a] with {
  abstract def isEmpty: Boolean
  abstract def head: a
  abstract def tail: List[a]

  def ::(x: a) =
    Cons(x)(this)

  def :::(prefix: List[a]): List[a] =
    if (prefix.isEmpty) this
    else prefix.head :: (prefix.tail ::: this)
```


Résumé

- Nous avons vu que les listes sont une structure de données fondamentale en programmation fonctionnelle.
- Les listes sont définies par des classes paramétrées, et opérées par des méthodes polymorphes.
- Les listes sont l'analogue des tableaux dans les langages impératifs.
- Mais contrairement aux tableaux, on n'accède généralement pas aux éléments d'une liste par leur indice.
- Plutôt, on traverse les listes récursivement ou via des combinateurs d'ordre supérieur tels que *map*, *filter*, *fold*, ou *foldRight*.

Deuxièmement,

```
    x :: []
  = reverse(x :: [])          // par spécification de reverse
  = (x :: []).fold(op)(z)     // par définition de reverse
  = op([], x)                 // par définition de fold
```

Par conséquent, $op([], x) = x :: []$. Cela suggère de prendre pour *op* l'opérateur :: en échangeant ses opérateurs.

On arrive donc à l'implantation suivante de *reverse*.

```
def reverse[a](xs: List[a]): List[a] = {
  def snoc(xs: List[a], x: a): List[a] = x :: xs
  xs.fold(snoc)([])
}
```

Q : Quelle est la complexité de cette implantation de *reverse* ?

Retour sur le renversement de liste

Voici une fonction de renversement de liste avec un coût linéaire.

L'idée est d'utiliser l'opération *fold* :

```
def reverse[a](xs: List[a]): List[a] = xs.fold(op?)(acc?)
```

On a simplement besoin de remplir les parties *op?* et *acc?*.

Essayons de les déduire à partir d'exemples.

Tout d'abord,

```
    []
  = reverse([])        // par spécification de reverse
  = [].fold(op)(z)     // par définition de reverse
  = z                  // par définition de fold
```

Par conséquent, $acc = []$.

```
def length: Int =
  if (isEmpty) 0
  else 1 + tail.length

def init: List[a] =
  if (isEmpty) error("Nil.init")
  else if (tail.isEmpty) Nil
  else head :: tail.init

def last: a =
  if (isEmpty) error("Nil.last")
  else if (tail.isEmpty) head
  else tail.last

def take(n: Int): List[a] =
  if (n == 0) Nil
  else head :: tail.take(n−1)

def drop(n: Int): List[a] =
  if (n == 0) this
  else tail.drop(n−1)

def at(n: Int) = drop(n).head
```



```
def map[b](f: (a)b): List[b] =
  if (isEmpty) Nil
  else f(head) :: tail.map(f)

def filter(p: (a)Boolean): List[a] =
  if (isEmpty) this
  else if (p(head)) head :: tail.filter(p)
  else tail.filter(p)

def reduce(op: (a, a)a): a =
  if (isEmpty) error("reduce of empty list")
  else tail.fold(op)(head)

def reduceRight(op: (a, a)a): a =
  if (isEmpty) error("reduce of empty list")
  else if (tail.isEmpty) head
  else op(head, tail.reduceRight(op))

def fold[b](op: (b, a)b)(z: b): b =
  if (isEmpty) z
  else tail.fold(op)(op(z, head))
```



```
  def foldRight[b](op: (a, b)b)(z: b): b =
    if (isEmpty) z
    else op(head, tail.foldRight(op)(z))

  def reverse: List[a] = {
    if (isEmpty) Nil
    else tail ::: head :: Nil
  }

  override def toString(): String = "[" + mkString(",") + "]"
  def mkString(sep: String): String = {
    if (isEmpty) ""
    else if (tail.isEmpty) head.toString()
    else head.toString().concat(sep).concat(tail.mkString(sep))
  }
}
```



```
final class Nil[a] extends List[a] with {
  def isEmpty = True
  def head: a = error("head of empty list")
  def tail: List[a] = error("tail of empty list")
}

final class Cons[a](x: a)(xs: List[a]) extends List[a] with {
  def isEmpty = False
  def head = x
  def tail = xs
}

def cons[a](x: a, xs: List[a]): List[a] = Cons(x)(xs)
def nil[a]: List[a] = Nil[a]
```