Semaine 4 : Filtrage de motifs (pattern matching)

Supposons que nous voulions écrire un petit interpréteur pour les expressions arithmétiques.

Pour rester simple, nous nous restreindrons aux nombres et aux additions.

Les expressions peuvent être représentées comme une hiérarchie de classe, avec une classe de base *Expr* et deux sous-classes *Number* et *Sum*.

Pour traiter une expression, il est nécessaire de connaître sa forme et ses composantes.

Cela nous amène à l'implantation suivante.


```
abstract class Expr {
    def isNumber: boolean;
    def isSum: boolean;
    def numValue: int;
    def leftOp: Expr;
    def rightOp: Expr;
}
class Number (n: int) extends Expr {
    def isNumber: boolean = true;
    def isSum: boolean = false;
    def numValue: int = n;
    def leftOp: Expr = error ("Number.leftOp");
    def rightOp: Expr = error ("Number.rightOp");
}
class Sum (e1: Expr, e2: Expr) extends Expr {
    def isNumber: boolean = false;
    def isSum: boolean = true;
    def numValue: int = error ("Sum.numValue");
    def leftOp: Expr = e1;
    def rightOp: Expr = e2;}
```

Maintenant on peut écrire une fonction d'évaluation comme suit.

```
def eval(e: Expr): int = {
  if (e.isNumber) e.numValue
  else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
}
```

Problème : Écrire toutes ces fonctions de classification et d'accès devient vite fastidieux !

Aussi, que se passe-t-il si l'on veut ajouter de nouvelles formes d'expressions, disons

```
class Prod(e1: Expr, e2: Expr) extends Expr;     // e1 * e2
class Var(x: String) extends Expr;               // Variable 'x'
```

Il faut alors ajouter des méthodes de classification et d'accès à toutes les classes définies précédemment.

lComment peut-on résoudre ce problème ?

Solution 1 : La décomposition orientée-objet

Par exemple, supposons qu'on veuille uniquement évaluer les expressions.

On pourrait alors définir :

```
abstract class Expr {
  def eval: int;
}
class Number(n: int) extends Expr {
  def eval: int = n;
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: int = e1.eval + e2.eval;
}
```

Mais que se passe-t-il si l'on désire maintenant afficher les expressions ?

- on doit définir de nouvelles méthodes dans toutes les sous-classes.

Et si l'on désire simplifier les expressions, par ex. au moyen de la règle :

$$a * b + a * c \quad \rightarrow \quad a * (b + c)$$

Problème : il s'agit d'une simplification non locale. Elle ne peut être encapsulée dans la méthode d'un seul objet.

Nous voilà de retour à la case départ : il nous faut des méthodes d'accès pour les différentes sous-classes.

Solution 2 : Décomposition fonctionnelle via le filtrage

Constatation : le seul but des fonctions de test et d'accès est de renverser le processus de construction :

- quelle sous-classe a-t-on utilisée ?
- quels étaient les arguments du constructeur ?

Cette situation est si fréquente qu'on l'automatise en Scala.

Classes cas (types algébriques)

Une définition de classe "cas" est similaire à une définition de classe normale, mais précédée du modificateur ***case***. Ex :

```
abstract class Expr;
case class Number(n: int) extends Expr;
case class Sum(e1: Expr, e2: Expr) extends Expr;
```

Comme auparavant, cela définit une classe mère abstraite *Expr* et deux sous-classes concrètes *Number* et *Sum*.

Cela définit aussi implicitement des fonctions de constructions (*factory methods*).

```
def Number(n: int) = new Number(n);
def Sum(e1: int, e2: int) = new Sum(e1, e2)
```

si bien qu'on peut écrire *Number(1)* au lieu de ***new*** *Number(1)*.

Toutefois, ces trois classes sont désormais vides, comment peut-on alors accéder aux membres ?

Le filtrage de motifs

Le filtrage de motifs (*pattern matching*) est une généralisation du *switch* de C/Java aux hiérarchies de classes.

Il s'exprime au moyen de la méthode standard *match*, définie dans la classe racine *Any*, et, dès lors, disponible partout.

Exemple :

```
def eval(e: Expr): int = e match {
  case Number(n) ⇒ n
  case Sum(e1, e2) ⇒ eval(e1) + eval(e2)
}
```

Règles :

- *match* est suivi d'une séquence de *cas*.
- Chaque cas associe une *expression* à un motif (*pattern*).
- Une exception *MatchError* est lancée si aucun motif ne filtre la valeur du sélecteur.

Forme des motifs

- Les motifs sont construits à partir :
  - de constructeurs, par ex. *Number*, *Sum*,
  - de variables, par ex. *n*, *e1*, *e2*,
  - du motif "joker" (wildcard) _,
  - de constantes, par ex. *1*, *true*.
- Les variables commencent toujours par une lettre minuscule.
- Un même nom de variable ne peut apparaître qu'une seule fois dans un motif. Par ex. *Sum*(x, x) n'est pas un motif légal.
- Les constructeurs et les noms de constante commencent par une lettre majuscule, à l'exception des mots réservés ***null***, *true*, *false*.

Signification du filtrage de motif

Une expression du type

e.match { ***case*** $p_1 \Rightarrow e_1$ ... ***case*** $p_n \Rightarrow e_n$ }

filtre la valeur du sélecteur e avec les motifs $p_1, \ldots, p_n$ dans l'ordre dans lequel ils sont écrits.

- Un motif "constructeur" $C(p_1, \ldots, p_n)$ filtre toutes les valeurs de type C (ou d'un sous-type) qui ont été construites avec des arguments filtrés par les motifs $p_1, \ldots, p_n$.
- Un motif "variable" x filtre n'importe quelle valeur et *lie* le nom de la variable à cette valeur.
- Un motif "constante" c filtre les valeurs qui sont égales à c (au sens de ==).

L'expression de filtrage se réécrit en la partie de droite du premier cas dont le motif filtre le sélecteur.

Les références aux variables du motif sont remplacées par les arguments correspondants du constructeur.

Exemple : :

```
     eval(Sum(Number(1), Number(2)))
→    Sum(Number(1), Number(2)) match {
       case Number(n) => n
       case Sum(e1, e2) => eval(e1) + eval(e2)
     }
→    eval(Number(1)) + eval(Number(2))
→    Number(1) match {
       case Number(n) => n
       case Sum(e1, e2) => eval(e1) + eval(e2)
     } + eval(Number(2))
→    1 + eval(Number(2))
→*   1 + 2 → 3
```


Filtrage de motifs et méthodes

Bien entendu, il est également possible de définir la fonction d'évaluation en tant que méthode de la super-classe.

Exemple :

```
abstract class Expr {
  def eval: int = this match {
    case Number(n) => n
    case Sum(e1, e2) => e1.eval + e2.eval
  }
}
```

Exercice

On considère les trois classes suivantes représentant des arbres d'entiers.

Ces classes peuvent être vues comme une représentation alternative de *IntSet* :

```
abstract class IntTree;
case class Empty extends IntTree;
case class Node(elem: int, left: IntTree, right: IntTree) extends IntTree;
```

Complétez l'implantation suivante de la fonction *contains* pour les *IntTrees*.

```
def contains(t: IntTree, v: int): boolean = t match {
  ...
}
```


Les listes

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

Une liste ayant comme éléments $x_1, ..., x_n$ s'écrit *List* $(x_1, ..., x_n)$.

Exemples :

```
val fruit  = List("apples", "oranges", "pears");
val nums   = List(1, 2, 3, 4);
val diag3  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1));
val empty  = List();
```

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 List

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 $List[T]$ (à comparer avec $[\,]T$ pour le type des tableaux d'éléments de type T en C ou Java).

Par ex.

$$\begin{array}{ll}
\textbf{val}\ fruit\ : List[String] & = List(\text{"apples"}, \text{"oranges"}, \text{"pears"}); \\
\textbf{val}\ nums : List[int] & = List(1, 2, 3, 4); \\
\textbf{val}\ diag3 : List[List[int]] & = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1));
\end{array}$$

Constructeurs de listes

Toutes les listes sont construites à partir de :

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

C.-à-d. :

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

Convention : L'opérateur '::' associe à droite. $A :: B :: C$ est interprété comme $A :: (B :: C)$.

On peut donc omettre les parenthèses dans la définition ci-dessus. Par ex.

$$nums \quad = 1 :: 2 :: 3 :: 4 :: Nil$$

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.

```
fruit.head = "apples"
fruit.tail.head = "oranges"
diag3.head = List(1, 0, 0)
empty.head  →  (Exception "head of empty list")
```


Exemple

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

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

Cette idée décrit le *Tri par Insertion* :

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

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

Quelle est la complexité du tri par insertion ?

Motifs de listes

:: et *Nil* sont toutes deux des classes "cas", il est donc aussi possible de décomposer les listes via le filtrage de motifs.

Comme sucre syntaxique, le constructeur *List* (...) peut aussi être utilisé comme un motif, avec la traduction

$List(p_1, ..., p_n) \;=\; p_1 :: ... :: p_n :: Nil.$

Une alternative est alors de réécrire *isort* comme suit.

```
def isort(xs: List[int]): List[int] = xs match {
  case List() => List()
  case x :: xs1 => insert(x, isort(xs1))
}
```

avec

```
def insert(x: int, xs: List[int]): List[int] = xs match {
  case List() => List(x)
  case y :: ys => if (x <= y) x :: xs else y :: insert(x, ys)
}
```


Autres fonctions sur les listes

En utilisant les constructeurs de liste et les motifs 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]): int = xs match {
  case List() => 0
  case x :: xs1 => 1 + length(xs1)
}
> 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 de type* additionnel à *length*.

```
def length[a](xs: List[a]): int =
  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 attendu du résultat (ce qui est généralement le cas).

Dans notre exemple, on aurait aussi pu écrire :

```
length(nums)      /* [int] inféré vu que nums: List[int] */
```

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

```
> def length(x: a) = ...
console:1: not found: a
```

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

Ce mot signifie ”qui a plusieurs formes” en Grec : en effet la fonction peut être appliquée à des différents arguments de types.

Concaténer des listes

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

Il existe aussi l'opérateur ::: (prononcer "concat") qui *concatène* deux listes.

```
> List(1, 2) ::: List(3, 4)
List(1, 2, 3, 4)
```

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

```
def concat[a](xs: List[a], ys: List[a]): List[a] = xs match {
  case List() ⇒
    ?
  case x :: xs1 ⇒
    ?
}
```

Q : Quelle est la complexité de *concat*?

Les 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]): List[a] = xs match {
  case List() ⇒ error("last of empty list")
  case List(x) ⇒ x
  case x :: xs1 ⇒ last(xs.tail)
}
```

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

Petite parenthèse : les exceptions

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

Elle est définie ainsi

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

Notez que *error* est polymorphe – 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.

La Fonction *reverse*

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

```
def reverse[a](xs: List[a]): List[a] = xs match {
  case List() => List()
  case x :: xs => reverse(xs) ::: List(x)
}
```

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).

La classe List

List n'est pas un type primitif en Scala. Il est défini par une classe de base abstraite et deux sous-classes pour :: et *Nil*. Voici une implantation partielle.

```
abstract class List[a] {
    def head: a;
    def tail: List[a];
    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.
- :: pour les listes non vides.

Les classes *Nil* et ::

Ces classes sont définies comme suit.

```
case class Nil[a] extends List[a] {
    def isEmpty = true;
    def head: a = error("Nil.head");
    def tail: List[a] = error("Nil.tail");
}

case class ::[a](x: a, xs: List[a]) extends List[a] {
    def isEmpty = false;
    def head: a = x;
    def tail: List[a] = xs;
}
```

Encore des méthodes de listes

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

```
abstract class List[a] {
  def head: a;
  def tail: List[a];
  def isEmpty: boolean;
  def length = match {
    case Nil => 0
    case x :: xs => 1 + xs.length
  }
  def init: List[a] = match {
    case Nil => error("Nil.init")
    case x :: Nil => List()
    case x :: xs => x :: init(xs)
  }
  ...
}
```


Les opérateurs Cons et Concat

Les opérateurs se terminant par ':' ont un traitement spécial en 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 méthode de leur opérande droite. Par ex.

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

  (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 :

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

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

```
abstract class List[a] {
  ...
  def ::(x: a): List[a] = new scala.::(x, this);

  def :::(prefix: List[a]): List[a] = prefix match {
    case Nil => this
    case p :: ps => p :: ps ::: this   /* or, equivalently: this.:::(ps).::(p) */
  }
```


Toujours plus de méthodes de listes

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 ainsi:

```
abstract class List[a] {
  ...
  def take(n: int): List[int] =
    if (n == 0) List() 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;
}
```

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

- le type des éléments à trier
- comment comparer deux éléments

La conception la plus flexible consiste à rendre la fonction *sort* polymorphe 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] = ...
    merge(msort(less)(xs take n), msort(less)(xs drop n))
  }
}
```

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

```
merge(List(1, 3), List(2, 4)) = List(1, 2, 3, 4)
merge(List(1, 2), List()) = List(1, 2)
```

Voici un exemple d'utilisation de *msort*.

```
> def iless(x: int, y: int) = x < y
> msort(iless)(List(5, 7, 1, 3))
List(1, 3, 5, 7)
```

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

```
> val intSort = msort(iless)
> val reverseSort = msort((x: int, y: int) => x > y)
> intSort(List(6, 3, 5, 5))
List(3, 5, 5, 6)
> reverseSort(List(6, 3, 5, 5))
List(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.

Les couples

Pair est une autre classe commune en Scala. Elle peut être définie ainsi

```
case class Pair[a, b](_1: a, _2: b);
```

Comme exemple d'utilisation, voici une fonction qui retourne le quotient et le reste de deux nombres entiers donnés...

```
def divmod(x: int, y: int) = Pair(x / y, x % y);
```

Et voici comme la fonction peut être utilisée :

```
divmod(x, y) match {
    case Pair(n, d) => System.out.println("quotient: " + n + ", rest: " + d);
}
```

Il est aussi possible d'utiliser le nom des paramètres du constructeur pour accéder aux éléments d'une classe cas directement. Par ex.

```
val p = divmod(x, y);
System.out.println("quotient: " + p._1);
```

L'idée des paires se généralise en Scala aux tuples de plus grande arité. Il existe une classe cas *Tuple_n* pour chaque *n* entre *2* et *9*.

Pair est en fait un alias pour *Tuple2*.

Motifs récurrents de calcul sur les listes

- 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] = xs match {
  case Nil ⇒ xs
  case x :: xs1 ⇒ x * factor :: scaleList(xs1, factor)
}
```

Ce schéma peut se généraliser en la méthode *map* de *List*:


```
abstract class List[a] { ...
  def map[b](f: a ⇒ b): List[b] = this match {
    case Nil ⇒ this
    case x :: xs ⇒ f(x) :: xs.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 élève au carré chaque élément d'une liste et retourne le résultat. Complétez les deux définitions équivalentes suivantes de *squareList*.

```
def squareList(xs: List[int]): List[int] = xs match {
  case List() => ??
  case y :: ys => ??
}
def squareList(xs: List[int]): List[int] =
  xs map ??
```


Filtrage

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

```
def posElems(xs: List[int]): List[int] = xs match {
  case Nil => xs
  case x :: xs1 => if (x > 0) x :: posElems(xs1) else posElems(xs1)
}
```

Ce schéma se généralise en la méthode *filter* de *List* :

```
abstract class List[a] {
  ...
  def filter(p: a => boolean): List[a] = this match {
    case Nil => this
    case x :: xs => if (p(x)) x :: xs.filter(p) else xs.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)
```