Solution 1 : Introduction d'une interface de haut niveau

Admettons que l'on désire uniquement évaluer les expressions.

On pourrait alors définir :

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

Mais que se passe-t-il si on désire imprimer les expressions ?

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

Problème : cela devient vite fastidieux !

D'autre part, que se passe-t-il si on désire ajouter un nouveau type d'expression ? P.ex.

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

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

Comment peut-on résoudre ce problème ?


```
abstract class Expr with {
  abstract def isNumber: Boolean;
  abstract def isSum: Boolean;
  abstract def numValue: Int;
  abstract def leftOp: Expr;
  abstract def rightOp: Expr;
}
class Number(n: Int) extends Expr with {
  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 with {
  def isNumber: Boolean = False;
  def isSum: Boolean = True;
  def numValue: Int = error("Sum.numValue");
  def leftOp: Expr = e1;
  def rightOp: Expr = e2;}
```


Le filtrage de motifs (*pattern matching*)

Imaginons que nous désirions écrire un petit interpréteur pour des expressions arithmétiques.

Pour rester simple, nous nous restreignons 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, nous devons connaître sa forme, et ses composantes.

Cela nous amène à l'implantation suivante.

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 de base *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*).

Classes cas (case classes)

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

Exemple :

```
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*.

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

Solution 2 : utiliser le filtrage de motifs

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.

Et si l'on désire simplifier les expressions, p.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 intégrée à une 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.

Filtrage et méthodes

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

Exemple :

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


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

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
```


Signification du filtrage

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 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 constant c filtre les valeurs qui sont égales à c (au sens de $==$).

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

- Un motif consiste en :
  - des constructors, p.ex. *Number*, *Sum*,
  - des variables, p.ex. *n*, *e1*, *e2*,
  - des constantes, p.ex. *1*, *True*.
- Les variables commencent toujours par une lettre minuscule.
- Les noms de constructeurs et de constantes commencent par une lettre majuscule. (C'est pour cela que l'on écrit *True* et pas *true*.)
- Les constructeurs sont des noms de classes cas.

Fonctions de filtrage anonymes

Jusqu'à présent, ***case*** est toujours apparu en même temps que *match*.

Mais il est aussi possible d'utilise ***case*** tout seul.

Exemple : Soit une liste de listes *xss*, l'expression suivante retourne les têtes de toutes les listes non-vides de *xss*:

```
xss flatMap {
  case x :: xs ⇒ [x]
  case [] ⇒ []
}
```

En fait, cette expression est équivalente à :

```
xss.flatMap {
  y ⇒ y match {
    case x :: xs ⇒ [x]
    case [] ⇒ []
  }
}
```


Motifs multiples

Il est également possible d'avoir plusieurs motifs pour un seul cas. Un cas du type

case $p_1, \ldots, p_n \Rightarrow e$

filtre le sélecteur si un des motifs $p_1, \ldots, p_n$ le filtre.

La fonction *zipFun* peut dès lors s'écrire comme suit :

```
def zipFun[a,b](xs: List[a], ys: List[b]): List[(a,b)] = (xs, ys) match {
  case ([], _), (_, []) ⇒ []
  case (x :: xs1, y :: ys1) ⇒ (x, y) :: zipFun(xs1, ys1)
}
```

Attention : pas encore implanté !

Motifs de tuples

Les tuples peuvent également être filtrés par des motifs. Un motif de tuple s'écrit $(p_1, \ldots, p_n)$, où $p_1, \ldots, p_n$ sont des motifs.

Exemple : Voici une nouvelle formulation de *zip*.

```
def zipFun[a,b](xs: List[a], ys: List[b]): List[(a,b)] = (xs, ys) match {
  case ([], _) ⇒ []
  case (_, []) ⇒ []
  case (x :: xs1, y :: ys1) ⇒ (x, y) :: zipFun(xs1, ys1)
}
```

Dans cet exemple,

- _ est un *motif générique* (*wildcard* en anglais). Il filtre n'importe quelle valeur.

  Un motif générique est donc équivalent à un motif variable x dont la variable x n'est jamais utilisée.

Motifs de listes

La classe *List* permet également le filtrage de motifs. Les motifs de listes sont semblables aux constructeurs. P.ex.

```
def concat[a](xss: List[List[a]]): List[a] = xss match {
  case [] ⇒ []
  case xs :: xss1 ⇒ xs ::: concat(xss1)
}
```

Le compilateur transforme chaque motif [] en *Nil*.

Il transforme de plus les motifs de construction $a :: b$ en $::_class(a)(b)$.

(Ici, $::_class$ est le nouveau nom de *Cons*).

Exemple plus conséquent : différentiation symbolique

Utilisons maintenant le filtrage dans un programme de différentiation symbolique.

Notre but est d'écrire une fonction *derive*, qui puisse idéalement s'utiliser comme suit :

```
> val x = Var("x")
> val expr = Number(7) * x * x * x + Number(3) * x
> expr derive x
21 * x * x + 3
```

Commençons par modifier le *langage* d'expressions que l'on désire traiter.

Considérons pour commencer les expressions composées uniquement de nombres, de variables et des opérateurs + et ∗.

Cela mène à la hiérarchie de classe suivante :


```
abstract class Expr with { ... }
case class Number(x: Int) extends Expr;
case class Var(name: String) extends Expr;
case class Sum(e1: Expr, e2: Expr) extends Expr;
case class Prod(e1: Expr, e2: Expr) extends Expr;
```

Notez que *Expr* est déclarée abstraite, car on ne veut pas créer de valeurs de ce type directement.

Définissons ensuite la fonction *derive* dans la classe *Expr*.

```
abstract class Expr with {
  def derive(v: Var): Expr = this match {
    case Number(_) ⇒ Number(0)
    case Var(name) ⇒ if (name == v.name) Number(1) else Number(0)
    case Sum(e1, e2) ⇒ Sum(e1 derive v, e2 derive v)
    case Prod(e1, e2) ⇒ Sum(Prod(e1,e2 derive v),Prod(e2,e1 derive v))
  }
}
```


Et voilà ! Nous pouvons déjà tester notre programme de dérivation.

```
> val x = Var("x")
> val expr = Prod(x, x)
> expr derive x
Sum(Prod(Var(x), Number(1)), Prod(Var(x), Number(1)))
```


Membres implicites des classes cas

Notez que les classes cas définissent implicitement des fonctions d'accès pour les paramètres de leur constructeur. C'est-à-dire que la définition suivante :

```
case class Var(name: String) extends Expr;
```

est augmentée ainsi :

```
case class Var(name: String) extends Expr with {
  def name: String = outer.name;
  override def toString() = "Var(" + name + ")";
}
```

Ici, *outer.name* fait référence à la définition de *name* en dehors du corps de la classe courante.

C'est-à-dire que, dans notre exemple, *outer.name* retourne l'argument du constructeur.

```
abstract class Expr with {
  def + (that: Expr) = Sum(this, that);
  def * (that: Expr) = Prod(this, that);
  def derive(v: Var): Expr = this match {
    case Number(_) ⇒ Number(0)
    case Var(name) ⇒ if (name == v.name) Number(1) else Number(0)
    case Sum(e1, e2) ⇒ (e1 derive v) + (e2 derive v)
    case Prod(e1, e2) ⇒ e1 * (e2 derive v) + e2 * (e1 derive v)
  }
}
```

On peut désormais écrire :

```
> val x = Var("x")
> val expr = x * x
> expr derive x
x * 1 + x * 1
> val expr1 = Number(2) * x * x + Number(3) * x
> expr1 derive x
2 * x * 1 + x * (2 * 1 + x * 0) + 3 * 1 + x * 0
```


On insère ainsi un nombre minimal de parenthèses.

On obtient maintenant :

```
> val x = Var("x")
> val expr = Prod(x, x)
> expr derive x
x * 1 + x * 1
```

Cela est mieux, mais bien vite on se rend compte qu'on désirerait également utiliser + et * dans les expressions que l'on entre.

Comment peut-on y arriver ?

Au moyen de la même technique utilisée pour définir + et * comme opérateurs sur les entiers : il suffit de définir des méthodes nommées + et * dans la classe *Expr*.


```
case class Number(x: Int) extends Expr with {
  override def toString() = x.toString()
}
case class Var(name: String) extends Expr with {
  override def toString() = name;
}
case class Sum(e1: Expr, e2: Expr) extends Expr with {
  override def toString() = e1.toString() + " + " + e2.toString();
}
case class Prod(e1: Expr, e2: Expr) extends Expr with {
  override def toString() = {
    def factorToString(e: Expr) = e match {
      case Sum(_, _) ⇒ "(" + e.toString() + ")"
      case _ ⇒ e.toString()
    }
    factorToString(e1) + " * " + factorToString(e2);
  }
}
```

La fonction *factorToString* de *Prod* place des parenthèses autour du facteur d'un produit uniquement si ce facteur est une somme.

Notez que les classes cas définissent implicitement une fonction *toString*; c'est pourquoi on voit

```
Sum(Prod(Var(x), Number(1)), Prod(Var(x), Number(1)))
```

et pas quelque chose comme

```
Sum@6547859495
```

s'afficher à l'écran.

Toutefois, pour notre exemple, ce n'est pas suffisant ; nous voudrions afficher les expressions sous une forme plus lisible.

Pour ce faire, redéfinissons la fonction *toString* dans chaque classe cas :

Résumé

Les classes cas et le filtrage de motifs sont des outils puissants pour décomposer les structures de données.

Au niveau syntaxique, les motifs sont similaires aux constructeurs d'objets.

Les motifs peuvent être composés de constructeurs, de variables et de constantes.

Filtrer un sélecteur avec un motif signifie :

- Si le motif filtre la valeur, alors le filtrage *réussit* et toutes les variables du motifs sont *liées* au composants correspondants de la valeur.
- Sinon, le filtrage *échoue*.

Exercice :

- Implantez ces simplifications dans la classe *Expr*.
- Testez votre programme en dérivant quelques expressions.
- Existe-t-il d'autres simplifications utiles ?

Exercice : Ajoutez une classe cas *Power*(*e1*: *Expr*, *e2*: *Int*) pour l'élévation à la puissance, et modifiez votre programme en fonction.


```
abstract class Expr with {
  def + (that: Expr) =
    /* retourne la version simplifiée de this + that */
  def * (that: Expr) =
    /* retourne la version simplifiée de this * that */
  def derive(x: Var): Expr =
    /* comme avant */
}
```

Un grand nombre de simplifications sont possibles, parmi lesquelles :

$Number(0) * e$	$\rightarrow$	$Number(0)$
$Number(1) * e$	$\rightarrow$	e
$Number(0) + e$	$\rightarrow$	e
$Number(n) * Number(m)$	$\rightarrow$	$Number(n * m)$
$Number(n) + Number(m)$	$\rightarrow$	$Number(n + m)$
$Var(x) * Number(n)$	$\rightarrow$	$Number(n) * Var(x)$
$e * Var(x) + e' * Var(x)$	$\rightarrow$	$(e + e') * Var(x)$

etc.

Il semble qu'il nous reste encore du travail.

Les expressions retournées sont correctes, mais pas simplifiées.

Cela peut être ennuyant pour les expressions complexes.

Solution : il faut *simplifier* les expressions.

Comme pour l'exemple des rationnels, on peut simplifier à différents moments :

1. lors de la construction d'une expression,
2. lors de l'affichage d'une expression, ou
3. lorsque le client le demande explicitement.

Nous choisissons la première solution : nous simplifions les expressions au moment de leur construction.

Question : Laquelle des deux décompositions choisiriez-vous pour :

- un compilateur Java, dans lequel la hiérarchie de classes représente les constructions syntaxiques du langage Java ?
- un gestionnaire de fenêtres, dans lequel la hiérarchie de classes représente les objets affichés ?

Décomposition utilisant le filtrage de motifs

D'un autre côté, si on décide d'effectuer la décomposition en utilisant le filtrage, il devient très simple d'ajouter de nouvelles opérations.

Par exemple, pour ajouter l'évaluation (*eval*) à notre classe d'expressions, il suffit d'écrire :

```
def eval(e: Expr) = e match {
  case Number(n) ⇒ n
  case Var(_) ⇒ error("impossible d'évaluer une variable")
  case Sum(e1, e2) ⇒ eval(e1) + eval(e2)
  case Prod(e1, e2) ⇒ eval(e1) * eval(e2)
}
```

Mais maintenant, c'est l'ajout d'un nouveau type d'expressions qui pose problème, car il implique de modifier tous les motifs pour prendre en compte le nouveau cas.

Décomposition orientée-objet

Reprenons notre classe *Expr* originale. Si tout ce que l'on désire faire est évaluer des expressions, on peut très bien implanter la méthode *eval* dans chaque sous-classe.

Ainsi, il est simple d'ajouter de nouveaux types d'expressions. P.ex.

```
class Prod(e1: Expr, e2: Expr) extends Expr {
  def eval = e1.eval * e2.eval;
}
```

Toutefois, l'ajout de nouvelles opérations (comme l'affichage, ou la dérivation) implique l'ajout d'une nouvelle méthode dans chaque sous-classe.

Deux formes de décomposition

Nous avons vu deux manières fondamentales d'organiser des hiérarchies de classes :

1. de manière orientée-objet classique, en déclarant les opérations en tant que méthodes, qui sont implantées séparément dans chaque sous-classe, ou
2. en ayant des sous-classes avec peu de méthodes (voire aucune), et en utilisant le filtrage pour accéder à l'objet.

Dans des langages sans filtrage, on peut utiliser le *design pattern* *Visiteur*.

La solution à choisir dépend de la situation.

Le facteur le plus important influençant ce choix est l'extensibilité.