Semaine 7 : Calcul symbolique

Les semaines passées nous avons vu les éléments essentiels de la programmation fonctionnelle moderne.

- les fonctions (de première classe)
- les types (paramétriques)
- le filtrage de motif
- les listes

Cette semaine nous allons appliquer certains de ces éléments à un exemple plus conséquent : la différentiation symbolique.

Mais avant, il nous reste à expliquer quel est le lien entre les fonctions et les objets en Scala.

Fonctions et objets

Scala est un langage fonctionnel.

⇒ Cela implique que les fonctions sont des valeurs (de première classe).

Scala est aussi un langage orienté objet (pur) :

⇒ Cela signifie que chaque valeur est un objet.

Donc, les fonctions sont des objets en Scala.

En fait, les fonctions avec n paramètres sont des instances du trait standard *scala.Functionn*, qui est défini ainsi

```
trait Functionn [a1, ..., an, b] {
  def apply (x1 : a1, ..., xn : an ): b
}
```

En particulier :

- Le type fonctionnel

  $(T_1, ..., T_n) \Rightarrow U$

  est juste un raccourci pour le type classe

  $Functionn[T_1, ..., T_n, U]$

- L'expression de fonction anonyme

  $(x_1 : T_1, ..., x_n : T_n) \Rightarrow E$

  où E a le type U est un raccourci pour une expression de création d'objet

```
new Functionn[T1, ..., Tn, U] {
    def apply(x1 : T1, ..., xn : Tn): U = E
}
```

- D'autre part, chaque fois qu'une valeur d'objet est appliquée à des arguments, une méthode *apply* implicite est insérée.

  $x(y_1, ..., y_n)$ *est un raccourci pour* $x.apply(y_1, ..., y_n)$

  Si x n'est pas une méthode.

Exemple : Considérons le code Scala

```
val plus1: int => int = x: int => x + 1;
plus1(2)
```

Il se développe en le code objet suivant.

```
val plus1: Function1[int, int] = new Function1[int, int] {
    def apply(x: int): int = x + 1
}
plus1.apply(2)
```

Fonctions de filtrage anonymes

Les fonctions anonymes peuvent également être construites à partir d'expressions ***case***.

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

Mais il est aussi possible d'utiliser ***case*** seul.

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

```
xss flatMap {
    case x :: xs => List(x)
    case List() => List()
}
```

En fait, cette expression est équivalente à :


```
xss.flatMap {
    y => y match {
        case x :: xs => List(x)
        case List() => List()
    }
}
```

La partie entre accolades est donc bien une fonction anonyme.

Exemple plus conséquent : la différentiation symbolique

On va maintenant utiliser le filtrage de motif dans un programme qui fait de la différentiation symbolique d'expressions.

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
```

On commence par définir le *langage* des expressions que l'on désire dériver.

On considère pour commencer les expressions composées uniquement de nombres, de variables et des opérateurs + et *.

Cela nous amène à la hiérarchie de classe suivante :


```
trait Expr { ... }
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 veut ne pas pouvoir créer de valeurs de ce type directement.

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

```
trait Expr {
  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. Autrement dit, la définition

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

est augmentée ainsi

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

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

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

et pas quelque chose comme

```
Sum@6547859495
```

à l'écran.

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

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


```
case class Number(x: int) extends Expr {
    override def toString() = x.toString()
}
case class Var(name: String) extends Expr {
    override def toString() = name;
}
case class Sum(e1: Expr, e2: Expr) extends {
    override def toString() = e1.toString() + " + " + e2.toString();
}
case class Prod(e1: Expr, e2: Expr) extends {
    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.

On insère ainsi un nombre minimum de parenthèses.

On obtient maintenant :

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

C'est mieux, mais cela appelle aussitôt une nouvelle amélioration :

On aimerait aussi pouvoir utiliser + et * dans les expressions *d'entrée*.

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*.


```
trait Expr {
  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
```

Il semble qu'il y ait encore du travail.

L'expression retournée est correcte, mais non simplifiée.

Cela peut être ennuyeux pour des expressions plus longues.

Solution : il nous faut *simplifier* les expressions.

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

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

Ici, on choisit la première solution : on veut simplifier les expressions au moment de leur construction.


```
trait Expr {
  def + (that: Expr) =
    /* return result of simplifying this + that */
  def * (that: Expr) =
    /* return result of simplifying this * that */
  def derive(x: Var): Expr =
    /* as before */
}
```

Un grand nombre de simplifications sont possibles, parmi lesquelles :

```
Number(0) * e               →  Number(0)
Number(1) * e               →  e
Number(0) + e               →  e
Number(n) * Number(m)       →  Number(n * m)
Number(n) + Number(m)       →  Number(n + m)
Var(x) * Number(n)          →  Number(n) * Var(x)
e * Var(x) + e' * Var(x)    →  (e + e') * Var(x)
```

etc.

Voici par exemple comment les simplifications liées au produit peuvent être implantées.

```
def * (that: Expr) = Pair(this, that) match {
    case Pair(Number(0), _) ⇒ Number(0)
    case Pair(_, Number(0)) ⇒ Number(0)
    case Pair(Number(1), e) ⇒ e
    case Pair(e, Number(1)) ⇒ e
    case Pair(Number(x), Number(y)) ⇒ Number(x * y)
    case Pair(Var(x), Number(y)) ⇒ Prod(Number(y), Var(x))
    case Pair(x, y) ⇒ Prod(x, y)
}
```

Exercice :

- Implantez les simplifications liées à la somme dans la classe *Expr*.
- Existe-t-il d'autres simplifications utiles ?

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

Deux formes de décomposition

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

1. de la 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 de motif pour décomposer un objet.

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

Le choix de la meilleure solution dépend de la situation.

Le facteur le plus important influençant ce choix est de décider ce qui doit pouvoir être étendu dans le futur.

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.

Dans ce cas, il est très simple d'ajouter de nouveaux types d'expressions. Par 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 existante.

Décomposition utilisant le filtrage de motifs

D'un autre côté, si l'on décide d'effectuer la décomposition en utilisant le filtrage de motif, il devient très facile 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): int = e match {
  case Number(n) ⇒ n
  case Var(_) ⇒ error("cannot evaluate 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 retrouver toutes les expressions utilisant le filtrage de motifs pour ajouter le nouveau cas.

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 à afficher ?

Exercice :

- Ajouter à la fonction *eval* un paramètre *env* de type *Map[String, Expr]* de manière à pouvoir aussi évaluer des expressions contenant des variables.