Week 7: Symbolic Computation

In the previous weeks, we've seen the essential elements of modern functional programming:

- Functions (first class)
- Types (parametric)
- Pattern Matching
- Lists

This week, we'll apply some of these elements to a more extensive example: symbolic differentiation.

But first, we must cover the relationship between functions and objects in Scala.

Functions and Objects

Scala is a functional language.

$\Rightarrow$ This implies that functions are values (of the first class)

Scala is also an object-oriented language (pure):

$\Rightarrow$ This means that each value is an object.

So, functions are objects in Scala.

In fact, functions with n parameters are instances of the standard trait, $scala.Function_n$, which is defined as

$$\begin{array}{l}\textbf{\textit{trait}}\ Function_n[T_1,\ ...,\ T_n,\ U]\ \{\\ \quad \textbf{\textit{def}}\ apply(x_1 : T_1,\ ...,\ x_n : T_n) : U\\ \}\end{array}$$

In particular:

- The functional type

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

  is just a shorthand for the class type

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

- The expression of anonymous function

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

  where E has type U is a shorthand for an expression of object creation

  new $Function_n[T_1, ..., T_n, U]$ $\{$
  def $apply(x_1 : T_1, ..., x_n : T_n)$: $U = E$
  $\}$

- On the other hand, each time the value of an object is applied to arguments, a method *apply* is implicitly inserted.

  $x(y_1, ..., y_n)$ *is shorthand for* $x.apply(y_1, ..., y_n)$

  If x isn't a method.

Example: Let's consider the following Scala code

```
val plus1 : Int ⇒ Int  =  x : Int ⇒ x + 1
plus1(2)
```

It results in the following code for an object.

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

Anonymous Partial Functions

Anonymous functions can also be constructed from ***case*** expressions.

Until now, ***case*** was always used along with ***match***.

But it is also possible to use ***case*** alone.

Example: Given a list of lists *xss*, the following expression returns the heads of all non-empty lists in *xss*:

$$
\begin{array}{l}
xss\ flatMap\ \{ \\
\quad \textbf{case}\ x :: xs \Rightarrow List(x) \\
\quad \textbf{case}\ List() \Rightarrow List() \\
\}
\end{array}
$$

In fact, this expression is equivalent to:

```
xss flatMap {
    y ⇒ y match {
        case x :: xs ⇒ List(x)
        case List() ⇒ List()
    }
}
```

The part in curly braces is an anonymous function.

More extensive example: symbolic differentiation

We will now use pattern matching in a program that performs symbolic differentiation of expressions.

Our goal is to write a function *derive*, which can ideally be used as follows:

$$
\begin{array}{l}
\mathit{scala}> \textbf{val}\ x = \mathit{Var}("x") \\
\mathit{scala}> \textbf{val}\ \mathit{expr} = \mathit{Number}(7) * x * x * x + \mathit{Number}(3) * x \\
\mathit{scala}> \mathit{expr}\ \mathit{derive}\ x \\
21 * x * x + 3
\end{array}
$$

We begin by defining the *language* of the expressions we'd like to derive.

To start, we consider expressions composed only of numbers, variables, and operators $+$ $*$.

This brings us to the following class hierarchy:

```
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
```

Note that *Expr* is a trait, because we want that it's not possible to create values of this type directly.

Next, we define the function *derive* in the class *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))
  }
}
```

That's it! We can already test our derivation program.

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

Implicit Members of Case Classes

Note that case classes implicitly define access functions for the parameters of their constructor. In other words, the definition

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

is transformed and augmented in the following way

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

Note also that case classes implicitly define a function *toString*; that's why case class instances are printed like

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

and not like

```
Sum@6547859495
```

However, for our example, this isn't enough; we'd like to be able to see the expressions in a more readable form.

To do this, we redefine the function *toString* in each case class:

```
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)
  }
}
```

The *factorToString* function of *Prod* puts parentheses around the factor of a product only if this factor is a sum.

We thus insert a minimum number of parentheses.

We now obtain:

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

It's better, but this immediately calls for a new improvement:

We'd also like to be able to use + and * in the *input* expressions.

How can we do this?

Using the same technique used to define + and * like operators on integers: simply define methods named + and * in the *Expr* class.

```
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)
  }
}
```

We can now write:

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

It seems like there's more work.

The expression returned is correct, but not simplified.

This can be annoying for longer expressions.

Solution: we need to *simplify* the expressions.

Like in the rationals example, we can simplify in different places:

1. during the construction of an expression,
2. when displaying an expression, or
3. when the user explicitly requests.

Here, we chose the first solution: we want to simplify the expressions as soon as they are constructed.

```
trait Expr {
  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 précédemment */
}
```

Many simplifications are possible, including:

$Number(0) * e$	$\rightarrow$	$Number(0)$
$Number(1) * e$	$\rightarrow$	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.

Here is how the simplifications on products can be implemented.

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

Exercise:

- Implement the simplifications on sums in the class *Expr*.
- Are there other useful simplifications?

Exercise: Add a case class $Power(e: Expr, n: Int)$ to represent exponents, and modify your program accordingly.

Two Forms of Decomposition

We've seen two fundamental ways to organize class hierarchies.

1. In the classical object-oriented way, by declaring operations as methods, which are implemented separately in each subclass, or
2. by having subclasses with few methods (if any), and by using pattern matching to decompose an object.

In languages without pattern matching, we can use the *Visitor design pattern*.

The choice of the best solution depends on the situation.

The most important factor influencing this choice depends on what needs to be extended in the future.

Object-Oriented Decomposition

Returning to our original class *Expr*. If all one wants is to evaluate expressions, one can easily implement the method *eval* in each subclass.

In this case, it is very simple to add new expression types, for example:

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

However, the addition of new operations (like printing) implies the addition of a new method in each existing subclass.

Decomposition Using Pattern Matching

On the other hand, if one decides to perform the decomposition by using pattern matching, it becomes very easy to add new operations.

For example, to add evaluation (*eval*) to our class of expressions, just write:

```
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)
}
```

But now, the addition of a new type of expression is problematic, because it involves finding all expressions using pattern matching in order to add the new case.

Question: Which of the the two decompositions would you choose for...

- ... a Java compiler, in which the class hierarchy represents the syntactic constructs of the Java language?
- a window manager, in which the class hierarchy represents the objects to be displayed?

Exercise:

- Add a parameter *env* of type $Map[String, Expr]$ to the *eval* function so it can also evaluate expressions containing variables.