## Week 7: Symbolic Computation

In the previous weeks we got to know essential elements of modern functional programming.

- (First-class) functions
- (Parameteric) types
- Pattern matching
- Lists

In this week we will apply some of these elements in an extended example: symbolic differentiation.

But before, we still need to explain how functions and objects are linked in Scala.

## Functions and Objects

Scala is a functional language.

$\Rightarrow$ This implies that functions are (first-class) values.

Scala is also a (pure) object oriented language:

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

*Ergo*, functions in Scala are objects.

Indeed, functions with $n$ parameters are instances of the standard trait *scala.Functionn*, which is defined as follows.

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

In particular:

- The function type

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

  is simply a shorthand for the class type

  $\mathit{Functionn}\,[T_1, ..., T_n, U]$

- The anonymous function expression

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

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

  ```
  new Functionn [T_1, ..., T_n, U] {
      def apply (x_1 : T_1, ..., x_n : T_n): U = E
  }
  ```

- On the other hand, every time an object value is applied to some arguments, an implicit *apply* method is inserted.

  $x\,(y_1, ..., y_n)$ *is a shorthand for* $x.\mathit{apply}\,(y_1, ..., y_n)$

  If $x$ is not a method.

**Example:** Consider the Scala code

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

This is expanded into the following object code.

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

## Anonymous Pattern Matching Functions

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

So far, ***case*** always appeared in conjunction with *match*.

But it is also possible to use ***case*** by itself.

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

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

In fact, this expression is equivalent to:

```
$xss.flatMap$ {
    $y \Rightarrow y$ $match$ {
        **case** $x :: xs \Rightarrow [x]$
        **case** $[\,] \Rightarrow [\,]$
    }
}
```

So the part in braces is in fact an anonymous function.

## Extended Example: Symbolic Differentiation

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

We will write a function *derive*, which should ideally work as follows.

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

We first fix the *language* of expressions which we want to differentiate.

Let's consider for the moment just numbers, variables, and operators + and ∗.

This leads to the following class hierarchy:

```
abstract class 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 declared abstract, because we do not want to create *Expr* values directly.

Now, let's define a *derive* function in class *Expr*.

```
abstract class 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 all, for now. We can already test our derivation program.

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

## Implicit Case Class Members

Note that case classes implicitly define access functions for their constructor parameters. I.e. the definition

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

is augmented to

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

Note that case classes also implicitly define a *toString* function; that's why we see

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

rather than something like

```
Sum@6547859495
```

as output.

However, for our example this is not good enough; we would like to display result expressions in more legible form.

This is achieved by overwriting the *toString* function 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 in *Prod* will place parentheses around a factor of a product only if that factor is a sum.

Therefore, the minimal number of parentheses are inserted.

Now we get:

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

This is better, but it immediately calls for the next improvement:

We also would like to use $+$ and $*$ in *input expressions*.

How can this be achieved?

The same way as $+$ and $*$ can be defined as operators for *int*'s: Simply define methods $+$ and $*$ in the *Expr* class.

```
abstract class 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)
    }
}
```

Now we can write:

```
> 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
```

It seems we are not done yet.

The resulting output is correct but unsimplified.

This can be annoying for larger expressions.

Solution: We need to *simplify* expressions.

As in the case of rational numbers, there are several choices where to simplify:

1. When constructing an expression.
2. When displaying an expression.
3. By an explicit operation which is called by the client.

We choose (1.). We want to simplify expressions as they are constructed.

```
abstract class 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 */
}
```

Many simplifications are possible, including:

|  |  |  |
|---|---|---|
| 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.

Here, for instance is how the simplifications of products can be implemented:

```
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)
}
```

**Exercise:**

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

**Exercise:** Add a case class $Power(e1: Expr, e2: Expr)$ for exponentiation and modify your program accordingly.

## Two Forms of Decomposition

We have seen two fundamental forms of organizing class hierarchies.

1. In the usual object-oriented way, by declaring all operations as methods, which are implemented separately in each subclass, or
2. By having subclasses with no (or few) methods and using pattern matching to access an object.

In languages without pattern matching, the *Visitor* design pattern can be used instead.

It depends on the situation which one of the two solutions is better.

The most important consideration in choosing an organization is to decide what needs to be extensible in the future.

## Object-Oriented Decomposition

Consider our original class *Expr* again. If all we ever want to do is evaluate expressions, we can very well implement an *eval* method in every expression form.

In this case, it is easy to add new forms of expressions. For instance:

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

But to add a new operation (such as pretty-print, or derive), requires the addition of a new method to every subclass constructed so far.

## Pattern-Matching Decomposition

On the other hand, if we choose decomposition by pattern matching, it is very easy to add new operations.

For instance, to add *eval* to our differentiation class, simply write:

```
def eval(e: Expr) = 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)
}
```

However, adding a new form of expression requires that we go back to every pattern matching decomposition and add the new case.

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

- ... for a Java compiler, where the class hierarchy represents Java's syntax constructs?
- for a Window-manager, where the class hierarchy represents display objects?