Week 7: Pattern Matching

Say, we want to write a little interpreter for arithmetic expressions.

For simplicity, we restrict ourselves to numbers and + operations.

Expressions can be represented as a class hierarchy, with a base class *Expr* and two subclasses *Number* and *Sum*.

To process an expression, we need to know what form it is, and what its components are.

This leads to the following implementation.


```
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;}
```


Problem: This is becoming boring!

Also, what happens if we want to add new forms of expression, say

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

Then we need to add new test and access methods to all previously defined classes.

How can we solve this dilemma?

Solution 1: Introduce a higher-level interface.

For instance, assume all we want is to evaluate expressions.

Then we could define:

```
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;
}
```

But what happens if we want to pretty-print expressions instead?

- need to define new method in all subclasses.

What if we want to simplify them? Say by applying the rule

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

Problem: This is a non-local simplification. It cannot be encapsulated in a method of a single object.

So, we are back to square 1: Need access methods for all different subclasses.

Solution 2: Use Pattern Matching

Observation: The only purpose of access functions is to reverse the construction process

- what sub-class was used?
- what were the constructor arguments?

This situation is so common that we automate it in Scala.

Case Classes

A case class definition is like a normal class definition, prefixed with the modifier ***case***. E.g.

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

As before, this defines an abstract superclass *Expr* with two concrete subclasses *Number* and *Sum*.

However, all three subclasses are now empty, so how can we access the members?

Pattern Matching

Pattern matching is a generalization of C/Java's *switch* statement to class hierarchies.

It is expressed by a standard method *match*, which is defined in the root class *Any* (so it is available everywhere).

Example:

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

Rules:

- *match* is followed by a sequence of *cases*.
- Every case associates a *pattern* with an *expression*.

Mixing Pattern Matching with Methods

Of course, it is also possible to define a pattern matching function as a method of the superclass.

Example:

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


References to pattern variables are replaced by corresponding constructor arguments.

Example: :

```
        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
```


Meaning of Pattern Matching

A pattern matching expression

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

matches the patterns $p_1, \ldots, p_n$ in the order they are written against the selector value e.

- A constructor pattern $C(p_1, \ldots, p_n)$ matches all values that are of type C (or a subtype thereof) and that have been constructed with C-arguments matching patterns $p_1, \ldots, p_n$.
- A variable pattern x matches any value and *binds* the variable name to that value.
- A constant pattern c matches values that are equal (in terms of ==) to c.

The pattern matching expression rewrites to the right-hand-side of the first case whose pattern matches the selector value.

- A pattern consists of
  - Constructors, e.g. *Number*, *Sum*,
  - variables, e.g. *n*, *e1*, *e2*,
  - Constants, e.g. *1*, *True*.
- Variables always start with a lower-case letter;
- Constructors and constant names start with an upper-case letter.

  (That's the reason why we write *True* instead of *true*.)
- Constructors are names of case classes.

Anonymous Pattern Matching Functions

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


Alternative Patterns

It is also possible to have several alternative patterns in a case. A case

case p_1, ..., p_n ⇒ e

matches a selector value if any one of the patterns $p_1, \ldots, p_n$ matches.

The *zipFun* example could therefore also have been written as follows.

```
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)
}
```

Warning : not implemented yet!

Tuple Patterns

Tuples also admit pattern matching. A tuple pattern is written $(p_1, \ldots, p_n)$, where $p_1, \ldots, p_n$ are patterns.

Example: Here's a new formulation of *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)
}
```

Here,

- _ is a *wildcard pattern*. It matches every value.

  A wildcard is hence equivalent to a variable pattern x where the variable x is not used.

List Patterns

The list class also admits pattern matching. List patterns mirror list constructors. Example:

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

The compiler maps every [] pattern to *Nil*.

It maps every cons-pattern a :: b to *::_class* (a) (b)

(Here, *::_class* is the new name for *Cons*).

Implicit Case Class Members

Note that case classes implicitly define access functions for they 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 = outer.name;
  override def toString ( ) = "Var (" + name + " )";
}
```

Here, *outer.name* refers to the definition on *name* outside the current class body.

I.e. in our example, *outer.name* returns the constructor argument.

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

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) ) )
```



```
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;
```

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 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))
  }
}
```


Extended Example: Symbolic Differentiation

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

We will write a function *diff*, 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 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)
  }
}
```

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
```


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.


```
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);
  }
}
```

The *factorToString* function in *Prod* will place parentheses around a factor of a product only if that factor is a sum.


```
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:

Summary

Case classes and pattern matching provide a convenient way for decomposing data structures.

The syntactical form of patterns mirrors the one of object constructors.

Patterns contain constructors, constants and variables.

Matching a pattern with a selector value means:

- If the form of the value matches the form of the pattern, then the match *succeeds* and all variables in the pattern are *bound* to corresponding components in the selector value.
- Otherwise, the match fails.

Exercise:

- Implement these simplifications in class *Expr*.
- Test your implementation by computing the derivations of sample expressions.
- Are there other useful simplifications?

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


```
abstract class Expr with {
  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:

$$
\begin{array}{lcl}
Number(0) * e & \rightarrow & Number(0) \\
Number(1) * e & \rightarrow & e \\
Number(0) + e & \rightarrow & e \\
Number(n) * Number(m) & \rightarrow & Number(n * m) \\
Number(n) + Number(m) & \rightarrow & Number(n + m) \\
Var(x) * Number(n) & \rightarrow & Number(n) * Var(x) \\
e * Var(x) + e' * Var(x) & \rightarrow & (e + e') * Var(x)
\end{array}
$$

etc.

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.

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?

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.

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.

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.