Лекция 5

Сопоставление с образцом в языке Scala

Удобным средством распространения возможности сопоставления с образцом на пользовательские типы являются case-классы. Объявление case-класса предваряется ключевым словом `case`. Например,

```
case class Progression(initial: Int, diff: Int)
```

Компилятор автоматически порождает для case-класса:

1. метод-фабрику, позволяющий создавать объекты без вызова `new`;

   ```
   val p = Progression(10, 2)
   ```

2. поля, соответствующие формальным параметрам конструктора;

3. реализацию метода `toString`;

4. реализацию методов `hashCode` и `equals`, соответствующую структурному сравнению объектов case-класса.

Для case-классов автоматически работает сопоставление с образцом, которое можно использовать как для декомпозиции объектов:

```
val Progression(a0, d) = p
```

так и для написания частичных функций:

```
def sign(p: Progression): Int = p match {
  case Progression(_, 0) => 0
  case Progression(_, d) if d > 0 => 1
  case _ => -1
}
```

Сочетание case-классов и наследования позволяет эмулировать так называемые «алгебраические типы», которые используются в некоторых языках (таких, как Standard ML) для представления деревьев

Например, абстрактное синтаксическое дерево формулы логики высказываний можно представить как

```
abstract class Formula
case class Var(name: String) extends Formula
case class Not(a: Formula) extends Formula
case class And(a: Formula, b: Formula) extends Formula
case class Or(a: Formula, b: Formula) extends Formula
```

Учитывая, что case-классы, конструкторы которых не имеют параметров, суть синглетоны, в Scala предусмотрена возможность объявлять case-объекты:

```
case object True extends Formula
case object False extends Formula
```

Пример (перевод формулы в нормальную форму отрицания, в которой операция НЕ применяется только к переменным и константам):

```
1  object Test {
2    def nnf(f: Formula): Formula = f match {
3      case Not(True)       => False
4      case Not(False)      => True
5      case Not(Not(a))     => nnf(a)
6      case Not(And(a, b)) => Or(nnf(Not(a)), nnf(Not(b)))
7      case Not(Or(a, b))  => And(nnf(Not(a)), nnf(Not(b)))
8      case And(a, b)       => And(nnf(a), nnf(b))
9      case Or(a, b)        => Or(nnf(a), nnf(b))
10     case other           => other
11   }

13   def main(args: Array[String]): Unit = {
14     val f = Not(And(Not(Var("a")), Or(Var("b"), True)))
15     println(nnf(f)) // Or(Var(a),And(Not(Var(b)),False))
16   }
17 }
```

К слову, на Standard ML пример выглядел бы следующим образом:

```
1  datatype formula
2    = Var of string
3    | Not of formula
4    | And of formula * formula
5    | Or of formula * formula
6    | True
7    | False

9  fun nnf (Not True)           = False
10   | nnf (Not False)          = True
11   | nnf (Not (Not a))        = nnf a
12   | nnf (Not (And (a, b))) = Or (nnf (Not a), nnf (Not b))
13   | nnf (Not (Or (a, b)))   = And (nnf (Not a), nnf (Not b))
14   | nnf (And (a, b))         = And (nnf a, nnf b)
15   | nnf (Or (a, b))          = Or (nnf a, nnf b)
16   | nnf other                = other
```

Компилятор Scala считает, что если имя переменной в образце начинается с маленькой буквы, то с этой переменной должно быть связано значение, выделяемое образцом из аргумента функции. Если нужно изменить это поведение, идентификатор придётся заключить в обратные кавычки

Пример (замена подформулы):

```
1 def replace(f: Formula, what: Formula, by: Formula): Formula =
2   f match {
3     case `what`     => by
4     case Not(a)     => Not(replace(a, what, by))
5     case And(a, b) =>
6       And(replace(a, what, by), replace(b, what, by))
7     case Or(a, b)  =>
8       Or(replace(a, what, by), replace(b, what, by))
9     case other      => other
10  }
```

Образцы в Scala допускают связывание переменных и подвыражений, соответствующих фрагментам образца. Для этого используется запись

```
имя_переменной @ образец
```

Пример (канонизация бинарной операции – переменная или константа становится вторым операндом):

```
1 def canonize(f: Formula): Formula = f match {
2   case And(a @ (Var(_) | True | False), b) => And(b, a)
3   case Or(a @ (Var(_) | True | False), b) => Or(b, a)
4   case other => other
5 }
```

Пример также демонстрирует использование альтернативных образцов:

```
Var(_) | True | False
```

Внутри альтернативных образцов не допускается связывание переменных.

Обратим внимание на то, что конструкторы case-классов в образцах можно записывать в инфиксной форме, т.е вместо

```
case And(a, b) => ...
```

использовать запись

```
case a And b => ...
```

К сожалению, для записи вызовов конструкторов case-классов в выражениях инфиксная форма недоступна, но нужного эффекта можно достичь добавлением методов-операций в класс `formula`:

```
1 abstract class Formula {
2   def And(b: Formula) = new And(this, b)
3   def Or(b: Formula) = new Or(this, b)
4 }
```

(Явный вызов операции `new` понадобился, чтобы отличать создание объекта от рекурсивного вызова метода)

Использование инфиксной формы записи образцов в сочетании с инфиксным вызовом методов-операций позволяет переписать метод `nnf` следующим образом:

```
1 def nnf(f: Formula): Formula = f match {
2   case Not(True)       => False
3   case Not(False)      => True
4   case Not(Not(a))     => nnf(a)
5   case Not(a And b)    => nnf(Not(a)) Or nnf(Not(b))
6   case Not(a Or b)     => nnf(Not(a)) And nnf(Not(b))
7   case a And b         => nnf(a) And nnf(b)
8   case a Or b          => nnf(a) Or nnf(b)
9   case other           => other
10 }
```

(В принципе, можно добиться ещё большей выразительности, дав case-классам `And` и `Or`, а также соответствующим методам-операциям имена, состоящие из спецсимволов, например: `&&` и `||`. Однако, удовольствие будет неполным, потому что оформить `Not` в образце в виде унарной операции не удастся)

Примером использования case-классов в стандартной библиотеке Scala является обобщённый тип `Option`, представляющий необязательное значение, возвращаемое функцией:

```
sealed abstract class Option[+A]
  extends Product with Serializable {
  ...
}

final case class Some[+A](x: A) extends Option[A] {
  ...
}

case object None extends Option[Nothing] {
  ...
}
```

Case-классы не являются единственной возможностью организовать сопоставление с образцом. На самом деле, они являются частным случаем *экстракторов*

Экстрактор – это объект, в котором реализован метод `unapply`, предназначенный для выполнения попытки декомпозиции значения на части

Метод `unapply` принимает декомпозируемое значение и возвращает `Option` от кортежа составных частей выражения. Подразумевается, что `unapply` возвращает `None` в случае неудачи

В вырожденном случае, когда выделение составных частей значения не требуется, а требуется лишь проверка соответствия значения некоторому критерию, метод `unapply` может возвращать `Boolean`

Экстрактор и фабрику для порождения значений часто совмещают в одном объекте. Фабрика реализуется в методе `apply`

Пример (фабрика и экстрактор для формул логики высказываний, на верхнем уровне которых находится бинарная операция):

```
1  object Binary {
2    def apply(op: String, a: Formula, b: Formula): Formula =
3      op match {
4        case "&" => And(a, b)
5        case "|" => Or(a, b)
6      }

8    def unapply(f: Formula): Option[(String,Formula,Formula)] =
9      f match {
10       case And(a, b) => Some("&", a, b)
11       case Or(a, b)  => Some("|", a, b)
12       case _         => None
13     }
14 }
```

Использование объекта `Binary` позволяет переписать метод `nnf` следующим образом:

```
1 def neg = Map("&" -> "|", "|" -> "&")

3 def nnf(f: Formula): Formula = f match {
4   case Not(True)                => False
5   case Not(False)               => True
6   case Not(Not(a))              => nnf(a)
7   case Not(Binary(op, a, b)) =>
8     Binary(neg(op), nnf(Not(a)), nnf(Not(b)))
9   case Binary(op, a, b)         => Binary(op, nnf(a), nnf(b))
10  case other                    => other
11 }
```

Можно заметить, что использование экстракторов в сочетании с фабриками позволяет отделить представление значения (`Binary`) от его реализации (`And` и `Or`)

Отметим, что case-классы – всего лишь синтаксический сахар. С помощью экстракторов можно добиться ровно такой же функциональности:

```
1  class And(val a: Formula, val b: Formula) extends Formula {
2    override def equals(obj: scala.Any): Boolean =
3      obj match {
4        case and: And => a == and.a && b == and.b
5        case _        => false
6      }
7    override def hashCode(): Int = ...
8    override def toString: String = "And(%s,%s)".format(a, b)
9  }

11 object And {
12   def apply(a: Formula, b: Formula): Formula = new And(a, b)
13   def unapply(f: Formula): Option[(Formula, Formula)] =
14     f match {
15       case and: And => Some(and.a, and.b)
16       case _        => None
17     }
18 }
```