

## 第7章 ケースクラスとパターンマッチング

ところで、数式のインタプリタを書きたいとしましょう。はじめは話を単純にするために、単に数と + 演算だけに制限します。そのような式はあるクラス階層、ルートの抽象基底クラス `Expr` と、2つのサブクラス `Number` と `Sum` を用いて表現できます。すると、式  $1 + (3 + 7)$  は次のように表現されます。

```
new Sum(new Number(1), new Sum(new Number(3), new Number(7)))
```

さて、このような式評価器は、それがどの形式であるか (`Sum` か `Number` か) を知る必要があります。式の要素にアクセスする必要もあります。次は必要なメソッドすべての実装です。

```
abstract class Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}
class Number(n: Int) extends Expr {
  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 {
  def isNumber: Boolean = false
  def isSum: Boolean = true
  def numValue: Int = error("Sum.numValue")
  def leftOp: Expr = e1
  def rightOp: Expr = e2
}
```

これらのクラス化とアクセスメソッドによって、評価器関数は簡単に書けます。

```
def eval(e: Expr): Int = {
  if (e.isNumber) e.numValue
  else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
  else error("unrecognized expression kind")
}
```

しかし、これらすべてのメソッドをクラス `Sum` と `Number` に定義するのは、かなり退屈です。さらに、式の新しい型を追加したくなった時に問題は悪化します。たとえば乗算のために新しい式の形式 `Prod` を追加することを考えてみましょう。既存のクラス化とアクセスメソッドに加えて、新しいクラス `Prod` を実装しなくてはならないだけでなく、クラス `Expr` に新しい抽象メソッド `isProduct` を導入する必要があります。そのメソッドをサブクラス `Number`, `Sum`, `Prod` に実装する必要があります。システムを拡張する時に、既存コードを修正しなくてはならないのは昔からの問題です。なぜならバージョン化と保守の問題を引き起こすからです。

オブジェクト指向プログラミングの約束することは、「そのような修正は不要です。なぜなら、継承によって既存の未修整のコードを再利用できるから」というものです。実際、問題をよりオブジェクト指向的に分解すれば問題は解決します。そのアイデアは「ハイレベルな」操作である `eval` を、前に我々がやったように、式クラス階層の外の関数として実装するのではなく、それぞれの式クラスのメソッドにすることです。そうすれば、`eval` はすべての式ノードのメンバなので、クラス化とアクセスメソッドはすべて不要となり、実装はかなり簡単になります。

```
abstract class Expr {
  def eval: Int
}
class Number(n: Int) extends Expr {
  def eval: Int = n
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
}
```

さらに、新しい Prod クラスの追加は既存コードに何も変化を引き起こしません。

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

この例から導かれる結論は、データ型の拡張可能なシステムを構築する際には、オブジェクト指向分解は選択すべきテクニックである、ということです。しかし他にも式の例を拡張したくなる方法があるかもしれませんが、式に対して新しい **操作** を追加したくなるかもしれません。たとえば、式の木を標準出力に整形して表示する操作を追加したくなるかもしれません。

もしすべてのクラス化とアクセスメソッドを定義してあれば、そういった操作は簡単に外部の関数として書けます。こんな風です。

```
def print(e: Expr) {
  if (e.isNumber) Console.print(e.numValue)
  else if (e.isSum) {
    Console.print("(")
    print(e.leftOp)
    Console.print("+")
    print(e.rightOp)
    Console.print(")")
  } else error("unrecognized expression kind")
}
```

しかし、オブジェクト指向分解を選んでいたら、新しい手続き print を各クラスに追加する必要があるでしょう。

```
abstract class Expr {
  def eval: Int
  def print
}
class Number(n: Int) extends Expr {
  def eval: Int = n
  def print { Console.print(n) }
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
  def print {
    Console.print("(")
    print(e1)
    Console.print("+")
    print(e2)
    Console.print(")")
  }
}
```

したがって、システムに新しい操作を入れて拡張する時には、古典的なオブジェクト指向分解では、既存のすべてのクラスの修正が必要になります。

インタプリタの一つの拡張として、式を単純化したいとしましょう。たとえば、式の形式を  $a * b + a * c$  から  $a * (b + c)$  へ書き換える関数が欲しいとします。この操作のためには、一つ以上の式木のノードを同時に調べる必要があります。しかし、メソッドが他のノードを調べることができなければ、式の種類ごとのメソッドでは実装できません。ですからこの場合には、クラス化とアクセスメソッドを強いられます。冗長さと拡張性の問題に満ちた四角四面なやり方に逆戻りのようです。

詳しく調べてみると、クラス化とアクセス関数はデータの構成プロセスを **逆転** させるだけが目的で分かります。それによって最初に、抽象クラスのどのサブクラスが使われたのか、その次にコンストラクタ引数が何であったのか、が決定されます。このような状況はかなり一般的なもので、Scala にはそれをケースクラスによって自動化する方法があります。

- [7.1 ケースクラスとケースオブジェクト](#)
- [7.2 パターンマッチング](#)

[前ページ](#) [7章](#) [目次](#) [次ページ](#)

- ちょっと意識なのだと思いますが、「オブジェクト指向プログラミングの約束することは、そのような修正は不要です」を「オブジェクト指向プログラミングが約束するのは、そのような修正は不要になるということです」な感じはいかがでしょうか・・・ -- ryugate (2008-05-24 00:58:29)

- そこまでやるんなら「そのような修正は不要であるとオブジェクト指向は約束してくれます」くらいまでやればいいじゃない -- 名無しさん (2008-07-07 17:26:36)

名前:	<input type="text"/>
コメント:	<input type="text"/>

投稿