

6 ケースクラスとパターンマッチング

プログラムによく出てくるデータ構造の一つにツリーがあります。例えばインタプリタやコンパイラは通常、プログラムを内部的にツリーで表現しています。XML 文書はツリーです。ある種のコンテナは赤黒木のようなツリーに基づいています。

小さな電卓プログラムを通して、ツリーが Scala でどのように表現され操作されるのを見てみましょう。このプログラムの目的は、加法と整数定数と変数からなる非常に簡単な数式を操作することです。その例を 2 つ挙げると、 $1+2$ や $(x+x)+(7+y)$ などです。

最初にそのような数式をどのように表現するか決めましょう。最も自然な方法はツリーです。ノードが演算（ここでは加法）で、リーフが値（ここでは定数か変数）です。

Java ではそういったツリーは、ツリーのための抽象スーパークラスと、ノードやリーフ毎に 1 つの具象サブクラスを用いて表現されるでしょう。関数型プログラミング言語では、同じ目的のために代数的データ型を用います。Scala には、両者の中間的なものである **ケースクラス** があります。それをどうやって私たちのツリーの型を定義するのに用いるかを示します。

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Sum, Var, Const クラスがケースクラスとして宣言されていることは、いくつかの点で普通のクラスとは違うということを意味しています。

- それらのクラスのインスタンスを作るのにキーワード **new** は必須ではありません。（すなわち、**new Const(5)** の代わりに **Const(5)** と書けます。）
- コンストラクタのパラメータ用の getter 関数は自動的に定義されます。（すなわち、Const クラスのインスタンス **c** のコンストラクタのパラメータ **v** の値は、単に **c.v** と書けば取得できます。）
- メソッド **equals** と **hashCode** はデフォルトで定義され、それらは同一性ではなくインスタンスの構造に基づいています。
- **toString** メソッドはデフォルトで定義され、値を「ソース形式」で表示します（例えば、式 $x+1$ の式のツリーは **Sum(Var(x), Const(1))** と表示されます）
- これらのクラスのインスタンスは以下で見るように **パターンマッチング** を通して分解されます

数式を表現するデータ型を定義したので、つぎにそれを操作する演算を定義しましょう。ある **環境** で式を評価する関数から始めることにします。環境の目的は変数に値を与えることです。例えば式 $x+1$ の評価を、変数 x を値 5 に関連づけるような環境 $\{x \ 5\}$ の元で行うと、結果 6 を得ます。

ここで環境を表現する方法を見つける必要があります。もちろんハッシュ表のような連想データ構造を使うこともできますが、直接に関数を使うこともできます！環境はまさに、値を（変数）名と関連付ける関数に他なりません。上で述べた環境 $\{x \ 5\}$ は Scala では下記のように簡単に書けます。

```
{ case "x" => 5 }
```

この書き方で、引数として文字列 "x" が与えられたなら整数 5 を返し、他の場合には例外で失敗させる関数を定義できます。

評価する関数を書く前に、環境の型に名前を付けましょう。もちろん、型 **String => Int** を環境のために使うこともできますが、この型に名前を付ければ、プログラムがシンプルになり将来の変更も容易になります。Scala では下記のように書けばよいのです。

```
type Environment = String => Int
```

以後、Environment 型は String から Int への関数の型の別名として使えます。

では評価する関数の定義を行いましょ。概念としてはとても簡単です。2 つの式の和の値は単にそれぞれの式の値の和です。変数の値は環境から直接得られます。そして定数の値は定数自身です。Scala でこれを表現するのは同じくらい簡単です。

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n) => env(n)
  case Const(v) => v
}
```

この評価関数はツリー **t** に対して **パターンマッチング** を実行します。直感的には上記の定義は明確なはずですが、

1. まずツリー **t** が Sum であるかチェックし、もしそうなら左部分木を新しい変数 **l** に、右部分木を変数 **r** に束縛します。そして

矢印に従って評価を進めます。矢印の左側のパターンによって束縛された変数 l と r を使用します。

- もし最初のチェックが成功しなければ、すなわちツリーは `Sum` でなければ、続いて t は `Var` かチェックします。もしそうなら `Var` の含まれる名前を変数 n に束縛し、右辺の式を用います。
- もし2番目のチェックにも失敗したなら、つまり t は `Sum` でも `Var` でもなければ、`Const` であるかチェックします。もしそうなら `Const` ノードに含まれる値を変数 v に束縛し、右辺へ進みます。
- 最後に、全てのチェックに失敗したなら、式のパターンマッチングの失敗を伝えるために例外が上げられます。ここで `Tree` のサブクラスが他に宣言されない限り、それは起きません。

ある値を一連のパターンに順に当てはめ、マッチしたら直ちにその値の様々なパーツを取り出して名前をつけ、名付けられたパーツを用いてコードを評価する、というパターンマッチングの基本的なアイデアを見てきました。

年期の入ったオブジェクト指向プログラマは、なぜ `eval` を `Tree` クラスとそのサブクラスの `メソッド` にしなかったのか、不思議に思うかもしれませんが、実はそのようにもできます。Scalaではケースクラスのメソッド定義は普通のクラスのようにできるからです。それゆえパターンマッチングとメソッドのどちらを使うかは趣味の問題ですが、拡張性にも密接に関係します。

- メソッドを用いれば、新しい種類のノードを追加することは `Tree` のサブクラスを定義することで簡単に行えます。しかしツリーを操作する新しい演算を追加するのは、`Tree` の全てのサブクラスの修正が必要なため面倒です
- パターンマッチングを用いれば状況は逆転します。新しい種類のノードを追加するには、ツリーのパターンマッチングを行う全ての関数で、新しいノードを考慮するための修正が必要です。その一方で新しい演算を追加するのは簡単で、単に独立した関数を定義するだけです。

パターンマッチングをもっと調べるために、別の数式への演算である微分シンボルを定義してみましょう。読者はこの演算に関する下記の規則を覚えているでしょうか。

- 和の導関数は、導関数の和。
- 変数 v の導関数は、 v が微分を行う変数なら 1 、さもなければ 0 。
- 定数の微分は 0 。

これらの規則はほとんど字句通り Scala のコードに変換でき、下記の定義を得ます：

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

この関数ではパターンマッチングに関する新しい概念を2つ導入します。最初に、変数に関する `case` 式には `ガード`、すなわち `if` キーワードに続く式があります。ガードは、式が真でなければパターンマッチングを失敗させます。ここでは、微分される変数名が微分する変数 v と等しい場合のみ定数 1 を返すように使われています。2つめにここで使われているパターンマッチングの特徴は、`_` で示される `ワイルドカード`、どんな値にもマッチするパターンで、値に名前をつける必要はありません。

パターンマッチングの能力を全て調べてはいませんが、この文書が長くなりすぎないようにここで止めておきましょう。上記2つの関数が実際の例でどう動くか見たいと思います。この目的のため、簡単な `main` 関数を書いて、式 $(x+x)+(7+y)$ に対する演算を幾つか行ってみましょう。最初に環境 $\{x=5, y=7\}$ に対する値を計算し、次いで x と y とで微分した導関数を求めましょう。

```
def main(args: Array[String]) {
  val exp: Tree = Sum(Sum(Var("x"), Var("x")), Sum(Const(7), Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x: \n " + derive(exp, "x"))
  println("Derivative relative to y: \n " + derive(exp, "y"))
}
```

プログラムを実行すると、期待した結果が得られます。

```
Expression: Sum(Sum(Var(x), Var(x)), Sum(Const(7), Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
Sum(Sum(Const(1), Const(1)), Sum(Const(0), Const(0)))
Derivative relative to y:
Sum(Sum(Const(0), Const(0)), Sum(Const(0), Const(1)))
```

結果を見ると、導関数の結果はユーザに見せる前に簡略化すべきであることが判ります。パターンマッチングを用いて基本的な簡約関数を定義することは興味深い（しかし驚く程に巧みな）問題です。読者の練習問題としておきます。

- 最終的なプログラムを載せてもらえませんか？ -- 名無しさん (2009-12-01 14:01:45)

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