The Expression Problem

Nancy Hitschfeld
Matías Toro

Why the Visitor Pattern?

The visitor pattern exists mainly because some languages don't support *pattern matching.*

Pattern matching is a mechanism for checking a value against a pattern. A successful match can also deconstruct a value into its constituent parts. It is a more powerful version of the switch statement and it can likewise be used in place of a series of if/else statements.

Pattern matching is a concept that goes "against" OOP, and is a feature used in the FP paradigm.

Pattern Matching: Syntax

```
e match {
   case … => …
   case … => …
   case … => …
   …
}
```

Matching Literals and Variables

```
import scala.util.Random

val x: Int = Random.nextInt(10)

x match {
  case 0 => "zero"
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
```

"Any other case"

Pattern Matching on classes

Case classes are especially useful for pattern matching.

```
sealed trait Notification

case class Email(sender: String, title: String, body: String) extends Notification

case class SMS(caller: String, message: String) extends Notification

case class VoiceRecording(contactName: String, link: String) extends Notification
```

```
def showNotification(notification: Notification): String = {
  notification match {
    case Email(sender, title, _) =>
      s"You got an email from $sender with title: $title"
    case SMS(number, message) =>
      s"You got an SMS from $number! Message: $message"
    case VoiceRecording(name, link) =>
      s"You received a Voice Recording from $name! Click the $link"
  }
}

showNotification(SMS("12345", "Are you there?"))
```

You got an SMS from 12345! Message: Are you there?

Pattern Matching: Pattern guards

Case classes are especially useful for pattern matching.

```
def showImportantNotification(
                notification: Notification,
                impPple: Seq[String]): String = {
  notification match {
    case Email(sender, _, _) if impPple.contains(sender) =>
      "You got an email from special someone!"
    case SMS(number, _) if impPple.contains(number) =>
      "You got an SMS from special someone!"
    case other =>
      //nothing special, delegate to our
      //original showNotification function
      showNotification(other)
  }
}
val importantPeopleInfo = Seq("867-5309", "jenny@gmail.com")
val someSms = SMS("867-5309", "Are you there?")
showImportantNotification(someSms, importantPeopleInfo)
```

You got an SMS from special someone!

Matching on lists

```
def sum(l: List[Int]): Int = {
  l match{
    case Nil => 0
    case head :: tail => head + sum(tail)
  }
}

println(sum(List(1,2,3,4,5)))
```

15

Pattern Matching on types

```
sealed trait Device

case class Phone(model: String) extends Device {
  def screenOff = "Turning screen off"
}

case class Computer(model: String) extends Device {
  def screenSaverOn = "Turning screen saver on..."
}

def goIdle(device: Device): String = device match {
  case p: Phone => p.screenOff
  case c: Computer => c.screenSaverOn
}
```

Pattern Matching: Extractor objects

```scala
import scala.util.Random

object CustomerID {

  def apply(name: String) = s"$name--${Random.nextLong()}"

  def unapply(customerID: String): Option[String] = {
    val stringArray: Array[String] = customerID.split("--")
    if (stringArray.tail.nonEmpty) Some(stringArray.head)
    else None
  }
}

val customer1ID = CustomerID("Sukyoung") // Sukyoung--23098234908
customer1ID match {
  case CustomerID(name) => println(name) // prints Sukyoung
  case _ => println("Could not extract a CustomerID")
}
```

Revisiting the FileSystem problem with Pattern Matching

```
class FileSystem() {
  …
  def getNumberOfFiles(): Int = {
    NumberOfFile(root)
  }

  def getNumberOfDirectory(): Int = {
    NumberOfDirectory(root)
  }

  def listing(): String = {
    Listing(root)
  }
}
```

Revisiting the FileSystem problem with Pattern Matching

```
object NumberOfFile {
  def apply(item: Item): Int = {
    item match{
      case d: Directory =>
        d.getItems().map(apply).sum
      case _ => 1
    }
  }
}
```

Revisiting the FileSystem problem with Pattern Matching

```
object NumberOfDirectory {
  def apply(item: Item): Int = {
    item match{
      case d: Directory =>
        1+d.getItems().map(apply).sum
      case _ => 0
    }
  }
}
```

Revisiting the FileSystem problem with Pattern Matching

```
object Listing {
  def apply(item: Item): String = {
    item match{
      case d: Directory => {
        /*d.getItems().foldLeft(d.getName()+"\n"){
          (acc, item) => acc + apply(item)
        }*/
        var tmp = d.getName()+"\n"
        for(item <- d.getItems()){
          tmp += apply(item)
        }
        tmp
      }
      case _ => item.getName()+"\n"
    }
  }
}
```

The Expression Problem

What happen if we want to **add a new operation** such as **SizeOf**?.

Easy and **low cost**:

```
object SizeOf{
  def apply(item: Item): Int = {
    item match {
      case d: Directory =>
        d.getItems().map(apply).sum
      case _ => item.getSize()
    }
  }
}
```

The Expression Problem

What happen if we want to **add a new variant** such as **SymbolicLink**?.

This might be **expensive**, as we may have to modify all of our operations and add an extra case.

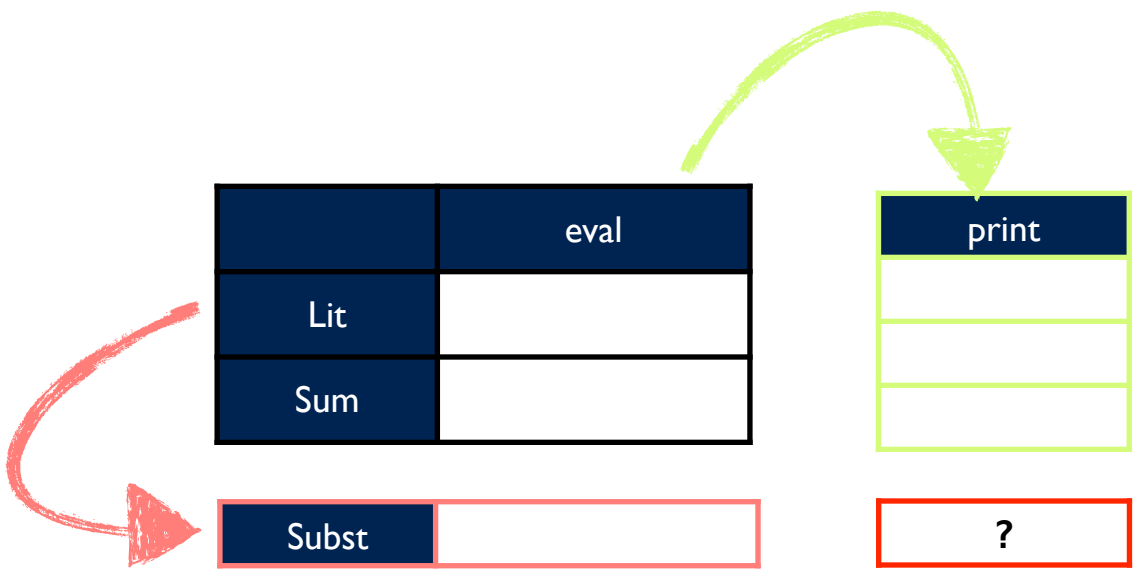
```
object SizeOf{
  def apply(item: Item): Int = {
    item match {
      case d: Directory =>
        d.getItems().map(apply).sum
      case i: SymbolicLink => ...
      case _ => item.getSize()
    }
  }
}
```

The Expression Problem

This extensibility problem is usually presented as follows: Given a language for arithmetic expressions, we want to extend it with two possible evolutions:

New variant: a new type of expression, e.g. **subtraction**

New operation: a new method, e.g., **pretty printing**



Expression Problem: OOP approach

Given a language for arithmetic expressions, we want to extend it with two possible evolutions:

New variant: a new type of expression, e.g. **subtraction**

New operation: a new method, e.g., **pretty printing**

```
trait Exp {
  def eval(): Int
}

class Lit(x: Int) extends Exp {
  def eval() = x
}

class Add(e1: Exp, e2: Exp) extends Exp {
  def eval() = e1.eval + e2.eval
}
```

Expression Problem: OOP approach.

Adding a **new variant** to the domain.

```
…
class Subs(e1: Exp, e2: Exp) extends Exp {
  def eval() = e1.eval - e2.eval
}
```

Low cost!

Expression Problem: OOP approach.

Adding a **new operation**.

```
trait Exp {
  def eval(): Int
  def print(): String
}

class Lit(x: Int) extends Exp {
  def eval() = x
  def print() = ""+x
}

class Add(e1: Exp, e2: Exp) extends Exp {
  def eval() = e1.eval + e2.eval
  def print() = "("+ e1.print + "+" + e2.print +")"
}
```

Expensive!

Expression Problem: FP approach

Given a language for arithmetic expressions, we want to extend it with two possible evolutions:

New variant: a new type of expression, e.g. **subtraction**

New operation: a new method, e.g., **pretty printing**

```
trait Exp {
}
case class Lit(x: Int) extends Exp
case class Add(e1: Exp, e2: Exp) extends Exp

def eval(e: Exp) = e match{
  case Lit(x) => x
  case Add(e1, e2) => eval(e1)+eval(e2)
}
```

Expression Problem: FP approach.

Adding a **new operation**.

```
…
def print(e: Exp) = e match{
   case Lit(x) => ""+x
   case Add(e1, e2) =>
        “(“+println(e1) + "+" + println(e2)+”)”
}
```

Low cost!

Expression Problem: FP approach.

Adding a **new variant** to the domain.

```
def eval(e: Exp) = e match{
  case Lit(x) => x
  case Add(e1, e2) => eval(e1)+eval(e2)
  case Subs(e1, e2) => println(e1)-println(e2))
}
def print(e: Exp) = e match{
  case Lit(x) => ""+x
  case Add(e1, e2) =>
    "("+print(e1) + "+" + print(e2)+")"
  case Subs(e1, e2) =>
    "("+print(e1) + "-" + print(e2)+")"
}
```

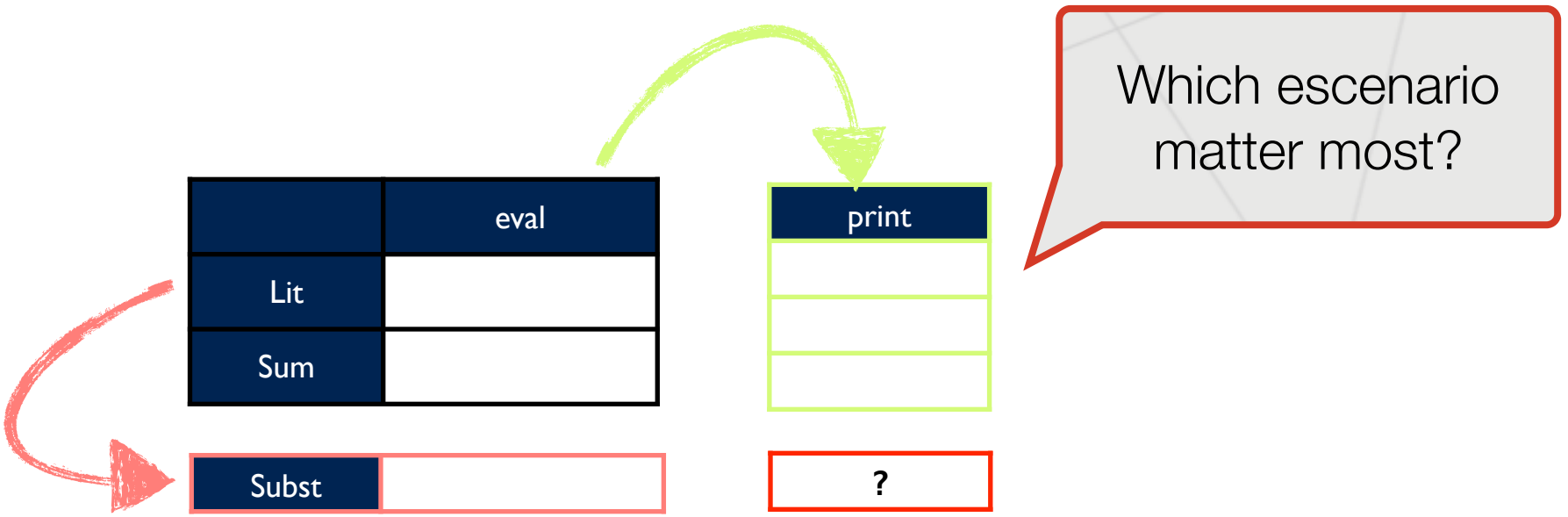
Expensive!

Expression Problem

Given a language for arithmetic expressions, we want to extend it with two possible evolutions:

New variant: a new type of expression, e.g. **subtraction**

New operation: a new method, e.g., **pretty printing**



Solutions to the Expression Problem

There are multiple solutions on the internet to this problem.

We are going to describe live, one fairly simple approach for Scala

License

dcc

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

www.dcc.uchile.cl

/ DCCUCHILE