Decomposition

Decomposition

Suppose you want to write a small interpreter for arithmetic expressions.

To keep it simple, let's restrict ourselves to numbers and additions.

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

To treat an expression, it's necessary to know the expression's shape and its components.

This brings us to the following implementation.

Expressions

isVar
isProd

name

```
trait Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}
```

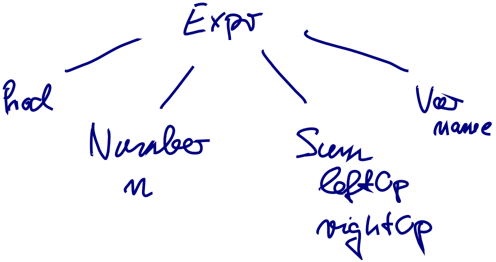
Classification

Accessor

8

```
class Number(n: Int) extends Expr {
  def isNumber: Boolean = true
  def isSum: Boolean = false
  def numValue: Int = n
  def leftOp: Expr = throw new Error(”Number.leftOp”)
  def rightOp: Expr = throw new Error(”Number.rightOp”)
}
```

8



Expressions (2)

```
  class Sum(e1: Expr, e2: Expr) extends Expr {
     def isNumber: Boolean = false
     def isSum: Boolean = true
     def numValue: Int = throw new Error(”Sum.numValue”)
     def leftOp: Expr = e1
     def rightOp: Expr = e2
}
```

8

$\text{new Sum}(e_1, e_2) \approx e_1 + e_2$

Evaluation of Expressions

You can now write an evaluation function as follows.

```
def eval(e: Expr): Int = {
  if (e.isNumber) e.numValue
  else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
  else throw new Error(”Unknown expression ” + e)
}
```

Problem: Writing all these classification and accessor functions quickly becomes tedious!

eval (Sum (Num (1), Num (2)) = 3

Adding New Forms of Expressions

So, what happens if you want to add new expression forms, say

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

You need to add methods for classification and access to all classes defined above.

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

○ 9

○ 10

○ 19

● 25

○ 35

○ 40

quadratic increase of methods

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

○ 9

○ 10

○ 19

○ 25

○ 35

○ 40

Non-Solution: Type Tests and Type Casts

A “hacky” solution could use type tests and type casts.

Scala let’s you do these using methods defined in class `Any`:

```
def isInstanceOf[T]: Boolean  // checks whether this object’s type conforms to ‘T‘
def asInstanceOf[T]: T        // treats this object as an instance of type ‘T‘
                              // throws ‘ClassCastException‘ if it isn’t.
```

These correspond to Java’s type tests and casts

```
Scala                   Java

x.isInstanceOf[T]       x instanceof T
x.asInstanceOf[T]       (T) x
```

But their use in Scala is discouraged, because there are better alternatives.

Eval with Type Tests and Type Casts

Here’s a formulation of the eval method using type tests and casts:

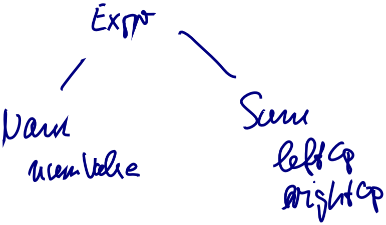
```
def eval(e: Expr): Int =
  if (e.isInstanceOf[Number])
    e.asInstanceOf[Number].numValue
  else if (e.isInstanceOf[Sum])
    eval(e.asInstanceOf[Sum].leftOp) +
    eval(e.asInstanceOf[Sum].rightOp)
  else throw new Error(”Unknown expression ” + e)
```

Assessment of this solution:

Eval with Type Tests and Type Casts

Here's a formulation of the eval method using type tests and casts:

```
def eval(e: Expr): Int =
  if (e.isInstanceOf[Number])
    e.asInstanceOf[Number].numValue
  else if (e.isInstanceOf[Sum])
    eval(e.asInstanceOf[Sum].leftOp) +
    eval(e.asInstanceOf[Sum].rightOp)
  else throw new Error(”Unknown expression ” + e)
```



Assessment of this solution:

+ no need for classification methods, access methods only for classes where the value is defined.
– low-level and potentially unsafe.

Solution 1: Object-Oriented Decomposition

For example, suppose that all you want to do is *evaluate* expressions.

You could then define:

```
trait Expr {
  def eval: Int ;  def show : String
}
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
}
```

But what happens if you'd like to display expressions now?

You have to define new methods in all the subclasses.

Limitations of OO Decomposition

And what if you want to simplify the expressions, say using the rule:

```
a * b + a * c   ->   a * (b + c)
```

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

You are back to square one; you need test and access methods for all the different subclasses.