

Monads in Scala

Bernie Pope

Outline

- The story behind monads.
- Monads in action.
- Desugaring Scala's 'for' comprehensions.

The story behind monads

The quest for modular semantics.

Denotational semantics

- The evaluation function:

$\llbracket - \rrbracket : \text{Expression} \Rightarrow \text{Value}$

Denotational semantics

- The evaluation function:

$\llbracket - \rrbracket : \text{Expression} \Rightarrow \text{Value}$

A fancy way to write the name
of the evaluation function.

Denotational semantics

- The evaluation function:

$\llbracket - \rrbracket : \text{Expression} \Rightarrow \text{Value}$

Syntax

Denotational semantics

- The evaluation function:

$\llbracket - \rrbracket : \text{Expression} \Rightarrow \text{Value}$

‘Mathematical’ values

Denotational semantics

- Some simple examples:

$$\llbracket 3 \rrbracket = \textcircled{3}$$

$$\llbracket e_1 + e_2 \rrbracket = \llbracket e_1 \rrbracket \oplus \llbracket e_2 \rrbracket$$

Denotational semantics

- Some simple examples:

$$\llbracket 3 \rrbracket = \textcircled{3}$$

The syntactic symbol 3 has the
'value' $\textcircled{3}$ (the integer three).

$$\llbracket e_1 + e_2 \rrbracket = \llbracket e_1 \rrbracket \oplus \llbracket e_2 \rrbracket$$

Denotational semantics

- Some simple examples:

$$\llbracket 3 \rrbracket = \textcircled{3}$$

$$\llbracket e_1 + e_2 \rrbracket = \llbracket e_1 \rrbracket \oplus \llbracket e_2 \rrbracket$$

$+$ is the syntactic symbol, and
 \oplus is the (mathematical) addition
function on integers.

The value domain

The value domain

- For our simple example, the value domain might be just the set of integers \mathbb{N} .

The value domain

- For our simple example, the value domain might be just the set of integers \mathbb{N} .
- But real programming languages are complex beasts:
 - side-effects (input/output).
 - non-termination.
 - exceptions.
 - recursion.
 - jumps.
 - non-determinism.

The value domain

- For our simple example, the value domain might be just the set of integers \mathbb{N} .
- But real programming languages are complex beasts:
 - side-effects (input/output).
 - non-termination.
 - exceptions.
 - recursion.
 - jumps.
 - non-determinism.

None of these things can be modelled reasonably by a semantic domain containing just integers.

Richer domains

Richer domains

- People invent more embellished domains of values.

Richer domains

- People invent more embellished domains of values.
- The domains get rather complicated.

Richer domains

- People invent more embellished domains of values.
- The domains get rather complicated.
- Combining them is tricky.

Richer domains

- People invent more embellished domains of values.
- The domains get rather complicated.
- Combining them is tricky.
- Hard to see the wood for all the trees.

Richer domains

- People invent more embellished domains of values.
- The domains get rather complicated.
- Combining them is tricky.
- Hard to see the wood for all the trees.
- We want modular semantics!

Abstraction to the rescue!

Abstraction to the rescue!

- Let ' $T\ \alpha$ ' be the type of *computations* yielding values of type α .

Abstraction to the rescue!

- Let ' $T\ \alpha$ ' be the type of *computations* yielding values of type α .
- For example ' $T\ \mathbb{N}$ ' is the type of some computation yielding an integer.

Abstraction to the rescue!

- Let ' $T\ \alpha$ ' be the type of *computations* yielding values of type α .
- For example ' $T\ \mathbb{N}$ ' is the type of some computation yielding an integer.
- We can specify T in different ways:
 - depending on what language features we want to have in the semantics.

Abstraction to the rescue!

- Now we can make the type of the evaluation function more abstract:

$\llbracket - \rrbracket : \text{Expression} \Rightarrow \text{T Value}$

An example model

An example model

- Suppose we want to represent stateful computations.

An example model

- Suppose we want to represent stateful computations.

$$T \alpha \stackrel{\text{def}}{=} \text{State} \Rightarrow (\text{State} \times \alpha)$$

An example model

- Suppose we want to represent stateful computations.

$$T \alpha \stackrel{\text{def}}{=} \text{State} \Rightarrow (\text{State} \times \alpha)$$

- A stateful computation is modelled as a function:
 - It takes a State as input.
 - It produces a State and a value as outputs.

Unit and bind

- We need some way to manipulate ‘ $T \alpha$ ’ things in the evaluation function.

$\text{unit} : \alpha \Rightarrow T \alpha$

$\text{bind} : T \alpha \Rightarrow (\alpha \Rightarrow T \beta) \Rightarrow T \beta$

Unit and bind

- We need some way to manipulate ' $T \alpha$ ' things in the evaluation function.

$\text{unit} : \alpha \Rightarrow T \alpha$

$\text{bind} : T \alpha \Rightarrow (\alpha \Rightarrow T \beta) \Rightarrow T \beta$

embed a value in the space of T

Unit and bind

- We need some way to manipulate ' $T \alpha$ ' things in the evaluation function.

$\text{unit} : \alpha \Rightarrow T \alpha$

$\text{bind} : T \alpha \Rightarrow (\alpha \Rightarrow T \beta) \Rightarrow T \beta$

The input is an ordinary value. The result is a constant computation yielding that value.

Unit and bind

- We need some way to manipulate ‘ $T \alpha$ ’ things in the evaluation function.

$\text{unit} : \alpha \Rightarrow T \alpha$

$\text{bind} : T \alpha \Rightarrow (\alpha \Rightarrow T \beta) \Rightarrow T \beta$

What that means depends on how we define T .

Unit and bind

- We need some way to manipulate 'T α ' things in the evaluation function.

$\text{unit} : \alpha \Rightarrow T \alpha$

$\text{bind} : T \alpha \Rightarrow (\alpha \Rightarrow T \beta) \Rightarrow T \beta$

Construct a new computation by sequential composition.

Unit and bind

- We need some way to manipulate ‘ $T \alpha$ ’ things in the evaluation function.

$\text{unit} : \alpha \Rightarrow T \alpha$

$\text{bind} : T \alpha \Rightarrow (\alpha \Rightarrow T \beta) \Rightarrow T \beta$

Again, what that means depends on
how we define T .

Unit and bind

- We need some way to manipulate 'T α' things in the evaluation function.

$\text{unit} : \alpha \Rightarrow T \alpha$

$\text{bind} : T \alpha \Rightarrow (\alpha \Rightarrow T \beta) \Rightarrow T \beta$

Any instance of T supplied with unit and bind is a monad.

Unit and bind

- We need some way to manipulate 'T α' things in the evaluation function.

$\text{unit} : \alpha \Rightarrow T \alpha$

$\text{bind} : T \alpha \Rightarrow (\alpha \Rightarrow T \beta) \Rightarrow T \beta$

Any instance of T supplied with unit and bind is a monad.

* Conditions apply.
(Monad Laws)

Effect basis

Effect basis

- Particular monads typically provide additional primitive operations on T .

Effect basis

- Particular monads typically provide additional primitive operations on T .
- Recall the state monad: $T \alpha \stackrel{\text{def}}{=} \text{State} \Rightarrow (\text{State} \times \alpha)$

Effect basis

- Particular monads typically provide additional primitive operations on T .
- Recall the state monad: $T \alpha \stackrel{\text{def}}{=} \text{State} \Rightarrow (\text{State} \times \alpha)$
- It needs these primitives to be of any real use:

$\text{get} : T \text{ State}$

$\text{put} : \text{State} \Rightarrow T 1$

Effect basis

- Particular monads typically provide additional primitive operations on T .
- Recall the state monad: $T \propto \stackrel{\text{def}}{=} \text{State} \Rightarrow (\text{State} \times \alpha)$
- It needs these primitives to be of any real use:

$\text{get} : T \text{ State}$

$\text{put} : \text{State} \Rightarrow T \mathbf{1}$

The type which only contains one value. Called Unit in Scala.

Using unit and bind in the evaluation function

- Now we can extend the evaluation function to the monadic style:

$$\llbracket 3 \rrbracket = \text{unit } \textcircled{3}$$

$$\llbracket e_1 + e_2 \rrbracket = \text{bind } \llbracket e_1 \rrbracket (\lambda v_1 \rightarrow \text{bind } \llbracket e_2 \rrbracket (\lambda v_2 \rightarrow \text{unit } (v_1 \oplus v_2)))$$

Using unit and bind in the evaluation function

- Now we can extend the evaluation function to the monadic style:

$\llbracket 3 \rrbracket = \text{unit } 3$

$\llbracket e_1 + e_2 \rrbracket = \text{bind } \llbracket e_1 \rrbracket (\lambda v_1 \rightarrow \text{bind } \llbracket e_2 \rrbracket (\lambda v_2 \rightarrow \text{unit } (v_1 \oplus v_2)))$



This might seem
unwieldy

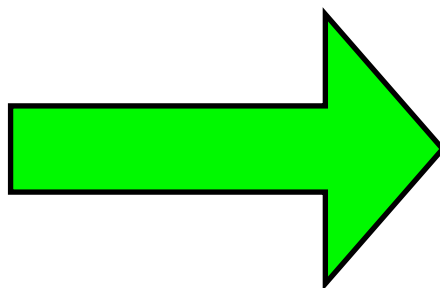
Using unit and bind in the evaluation function

- Now we can extend the evaluation function to the monadic style:

$\llbracket 3 \rrbracket = \text{unit } 3$

$\llbracket e_1 + e_2 \rrbracket = \text{bind } \llbracket e_1 \rrbracket (\lambda v_1 \rightarrow \text{bind } \llbracket e_2 \rrbracket (\lambda v_2 \rightarrow \text{unit } (v_1 \oplus v_2)))$

This might seem
unwieldy



```
for { v1 <- e1 ;  
      v2 <- e2 }  
yield v1 ⊕ v2
```

But it has echoes of something
quite familiar to Scala programmers.

But what does it have to do with programming?

But what does it have to do with programming?

- Originally monads were used in denotational semantics (Eugenio Moggi).

But what does it have to do with programming?

- Originally monads were used in denotational semantics (Eugenio Moggi).
- It was quickly realised that we can program with them too (Philip Wadler).

But what does it have to do with programming?

- Originally monads were used in denotational semantics (Eugenio Moggi).
- It was quickly realised that we can program with them too (Philip Wadler).
- We model T as some generic type constructor (with one parameter).

But what does it have to do with programming?

- Originally monads were used in denotational semantics (Eugenio Moggi).
- It was quickly realised that we can program with them too (Philip Wadler).
- We model T as some generic type constructor (with one parameter).
- `unit` and `bind` are functions which operate on the type chosen for T .

And what does it have to do with Scala?

- The List type is but one of many Scala types which can form a monad:

$$T \alpha \stackrel{\text{def}}{=} \text{List}[\alpha]$$
$$\text{unit } x \stackrel{\text{def}}{=} x :: \text{Nil}$$
$$\text{bind } t \ f \stackrel{\text{def}}{=} t.\text{flatMap}(f)$$

And what does it have to do with Scala?

- The List type is but one of many Scala types which can form a monad:

$$T \alpha \stackrel{\text{def}}{=} \text{List}[\alpha]$$
$$\text{unit } x \stackrel{\text{def}}{=} x :: \text{Nil}$$
$$\text{bind } t \ f \stackrel{\text{def}}{=} t.\text{flatMap}(f)$$

```
def flatMap[B](f : (A) => Iterable[B]) : List[B]
```

flatMap applies the given function *f* to each element of the list *t*, then concatenates the result.

Monads in practice

The quest for modular programs.

Let's write an interpreter for a little language

- Grammar:

$$\text{expr} \stackrel{\text{def}}{=} \text{expr} \text{ '+' expr} \mid \text{expr} \text{ '*' expr} \mid \text{expr} \text{ '-' expr} \mid \text{integer} \mid \text{'(' expr ')'}$$

- Some example expressions and their expected values:

- ▶ $3 \mapsto 3$

- ▶ $2 * 6 \mapsto 12$

- ▶ $10 - 2 * 4 \mapsto 2$

- ▶ $(10 - 2) * 4 \mapsto 32$

Scala types to represent expressions

```
abstract class Expr  
case class BinOp (op:String, left:Expr, right:Expr) extends Expr  
case class Number (number:Int) extends Expr
```

A recursive evaluation procedure

```
def eval(e:Expr) : Int = e match {  
  case Number(n) => n  
  case BinOp(o,l,r) => evalOp(o,eval(l),eval(r))  
}  
  
def evalOp(o:String, l:Int, r:Int) : Int = o match {  
  case "*" => l * r  
  case "-" => l - r  
  case "+" => l + r  
}
```


A recursive evaluation procedure

```
def eval(e:Expr) : Int = e match {  
  case Number(n) => n  
  case BinOp(o,l,r) => evalOp(o,eval(l),eval(r))  
}  
  
def evalOp(o:String, l:Int, r:Int) : Int = o match {  
  case "*" => l * r  
  case "-" => l - r  
  case "+" => l + r  
}
```

No monads here.

Let's add integer division to the language

- Extend the grammar:

$$\text{expr} \stackrel{\text{def}}{=} \dots \mid \text{expr} \text{ '/' } \text{expr} \mid \dots$$

- Some example expressions and their expected values:

▶ $10 / 3 \mapsto 3$

▶ $1 / 2 \mapsto 0$

▶ $1 / 0 \mapsto \text{error "divide by zero"}$

Let's add integer division to the language

- Extend the grammar:

$$\text{expr} \stackrel{\text{def}}{=} \dots \mid \text{expr} \text{ '/' } \text{expr} \mid \dots$$

- Some example expressions and their expected values:

▶ $10 / 3 \mapsto 3$

▶ $1 / 2 \mapsto 0$

▶ $1 / 0 \mapsto$ error “divide by zero”

This is not an integer!

It is common to use exceptions to catch errors

```
try { println(eval(e)) }  
  
catch {  
  case ex: ArithmeticException => println(ex.getMessage)  
}
```

No monads here, either.

Using Option to encode partiality

```
def eval(e:Expr) : Option[Int] = e match {  
  case Number(n) => Some(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

```
def evalOp(o:String, l:Int, r:Int) : Option[Int] = o match {  
  case "*" => Some(l * r)  
  case "-" => Some(l - r)  
  case "+" => Some(l + r)  
  case "/" => if (r == 0) None else Some(l / r)  
}
```

Using Option to encode partiality

```
def eval(e:Expr) : Option[Int] = e match {  
  case Number(n) => Some(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

The failure monad!

```
def evalOp(o:String, l:Int, r:Int) : Option[Int] = o match {  
  case "*" => Some(l * r)  
  case "-" => Some(l - r)  
  case "+" => Some(l + r)  
  case "/" => if (r == 0) None else Some(l / r)  
}
```

Using Option to encode partiality

```
def eval(e:Expr) : Option[Int] = e match {  
  case Number(n) => Some(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

The original result type is embedded
in Option.

```
def evalOp(o:String, l:Int, r:Int) : Option[Int] = o match {  
  case "*" => Some(l * r)  
  case "-" => Some(l - r)  
  case "+" => Some(l + r)  
  case "/" => if (r == 0) None else Some(l / r)  
}
```

Using Option to encode partiality

```
def eval(e:Expr) : Option[Int] = e match {  
  case Number(n) => Some(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

“Some” encodes success (unit).

```
def evalOp(o:String, l:Int, r:Int) : Option[Int] = o match {  
  case "*" => Some(l * r)  
  case "-" => Some(l - r)  
  case "+" => Some(l + r)  
  case "/" => if (r == 0) None else Some(l / r)  
}
```


Using Option to encode partiality

```
def eval(e:Expr) : Option[Int] = e match {  
  case Number(n) => Some(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

“None” encodes failure.

```
def evalOp(o:String, l:Int, r:Int) : Option[Int] = o match {  
  case "*" => Some(l * r)  
  case "-" => Some(l - r)  
  case "+" => Some(l + r)  
  case "/" => if (r == 0) None else Some(l / r)  
}
```

Using Option to encode partiality

```
def eval(e:Expr) : Option[Int] = e match {  
  case Number(n) => Some(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

Sugar for a chain of binds.

```
def evalOp(o:String, l:Int, r:Int) : Option[Int] = o match {  
  case "*" => Some(l * r)  
  case "-" => Some(l - r)  
  case "+" => Some(l + r)  
  case "/" => if (r == 0) None else Some(l / r)  
}
```

Using Option to encode partiality

```
def eval(e:Expr) : Option[Int] = e match {  
  case Number(n) => Some(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

If any individual statement fails,
the whole computation fails.

```
def evalOp(o:String, l:Int, r:Int) : Option[Int] = o match {  
  case "*" => Some(l * r)  
  case "-" => Some(l - r)  
  case "+" => Some(l + r)  
  case "/" => if (r == 0) None else Some(l / r)  
}
```

Let's add multiple solutions to the language

- Extend the grammar:

$$\text{expr} \stackrel{\text{def}}{=} \dots \mid \text{expr} \text{ '?' expr} \mid \dots$$

- Some example expressions and their expected values:

▶ $12 \mapsto \{12\}$

▶ $10 \text{ ? } 3 \mapsto \{10, 3\}$

▶ $2 * (10 \text{ ? } 3) = \{2 * 10, 2 * 3\} \mapsto \{20, 6\}$

▶ $1 / 0 \mapsto \{\}$

▶ $(1 / 0) \text{ ? } 1 \mapsto \{\}$

▶ $1 / (0 \text{ ? } 1) \mapsto \{1\}$

Using Set to encode multiple solutions

```
def eval(e:Expr) : Set[Int] = e match {  
  case Number(n) => Set(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

```
def evalOp(o:String, l:Int, r:Int) : Set[Int] = o match {  
  case "*" => Set(l * r)  
  case "-" => Set(l - r)  
  case "+" => Set(l + r)  
  case "/" => if (r == 0) Set() else Set(l / r)  
  case "?" => Set(l,r)  
}
```

Using Set to encode multiple solutions

```
def eval(e:Expr) : Set[Int] = e match {  
  case Number(n) => Set(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

The non-determinism monad.

```
def evalOp(o:String, l:Int, r:Int) : Set[Int] = o match {  
  case "*" => Set(l * r)  
  case "-" => Set(l - r)  
  case "+" => Set(l + r)  
  case "/" => if (r == 0) Set() else Set(l / r)  
  case "?" => Set(l,r)  
}
```

Using Set to encode multiple solutions

```
def eval(e:Expr) : Set[Int] = e match {  
  case Number(n) => Set(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

Non-empty set encodes success.

```
def evalOp(o:String, l:Int, r:Int) : Set[Int] = o match {  
  case "*" => Set(l * r)  
  case "-" => Set(l - r)  
  case "+" => Set(l + r)  
  case "/" => if (r == 0) Set() else Set(l / r)  
  case "?" => Set(l,r)  
}
```

Using Set to encode multiple solutions

```
def eval(e:Expr) : Set[Int] = e match {  
  case Number(n) => Set(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

Empty set encodes failure.

```
def evalOp(o:String, l:Int, r:Int) : Set[Int] = o match {  
  case "*" => Set(l * r)  
  case "-" => Set(l - r)  
  case "+" => Set(l + r)  
  case "/" => if (r == 0) Set() else Set(l / r)  
  case "?" => Set(l,r)  
}
```


Using Set to encode multiple solutions

```
def eval(e:Expr) : Set[Int] = e match {  
  case Number(n) => Set(n)  
  case BinOp(o,l,r) =>  
    for {  
      x <- eval(l)  
      y <- eval(r)  
      z <- evalOp(o,x,y)  
    }  
    yield z  
}
```

If either l or r fails,
the whole computation fails.
Otherwise take the union of each
result.

```
def evalOp(o:String, l:Int, r:Int) : Set[Int] = o match {  
  case "*" => Set(l * r)  
  case "-" => Set(l - r)  
  case "+" => Set(l + r)  
  case "/" => if (r == 0) Set() else Set(l / r)  
  case "?" => Set(l,r)  
}
```

Substantial changes from Option to Set

```
Set[    ]
```

```
Set[    ]
```

```
case "?" => Set(l,r)
```

What have we achieved?

What have we achieved?

- A separation of concerns.

What have we achieved?

- A separation of concerns.
- Values are distinguished from ‘computations of values’.

What have we achieved?

- A separation of concerns.
- Values are distinguished from ‘computations of values’.
 - ▶ The essential recursive algorithm remains manifest ...

What have we achieved?

- A separation of concerns.
- Values are distinguished from ‘computations of values’.
 - ▶ The essential recursive algorithm remains manifest ...
 - ▶ ... and independent of the underlying computational effects (failure, non-determinism).

Scala's 'for' comprehensions

Binds in disguise

Desugaring

Desugaring

- One generator:

```
for { x <- e1 } yield e2
```

Desugaring

- One generator:

`for { x <- e1 } yield e2`

- Is sugar for:

`e1.map (x => e2)`

Desugaring

- One generator:

```
for { x <- e1 } yield e2
```

- Is sugar for:

```
e1.map (x => e2)
```

- More than one generator:

```
for { x <- e1; generators } yield e2
```

Desugaring

- One generator:

```
for { x <- e1 } yield e2
```

- Is sugar for:

```
e1.map (x => e2)
```

- More than one generator:

```
for { x <- e1; generators } yield e2
```

- Is sugar for:

```
e1.flatMap (x => for { generators } yield e2)
```

Desugaring

- One generator:

```
for { x <- e1 } yield e2
```

- Is sugar for:

```
e1.map (x => e2)
```

and desugar recursively

- More than one generator:

```
for { x <- e1; generators } yield e2
```

- Is sugar for:

```
e1.flatMap (x => for { generators } yield e2)
```

Desugaring

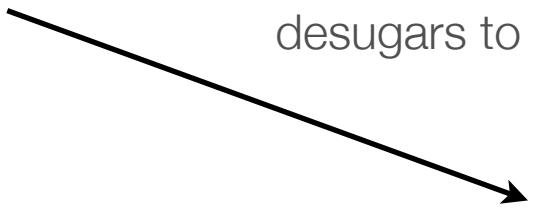
Desugaring

```
for {  
  x <- eval(l)  
  y <- eval(r)  
  z <- evalOp(o,x,y)  
}  
yield z
```


Desugaring

```
for {  
  x <- eval(l)  
  y <- eval(r)  
  z <- evalOp(o,x,y)  
}  
yield z
```

desugars to



```
eval(l).flatMap (x =>  
  eval(r).flatMap (y =>  
    evalOp(o,x,y).map (z => z)))
```

Desugaring

```
for {  
  x <- eval(l)  
  y <- eval(r)  
  z <- evalOp(o,x,y)  
}  
yield z
```

desugars to

```
eval(l).flatMap (x =>  
  eval(r).flatMap (y =>  
    evalOp(o,x,y).map (z => z)))
```

is equivalent to

```
eval(l).flatMap (x =>  
  eval(r).flatMap (y =>  
    evalOp(o,x,y)))
```

One generator is really just a special case

One generator is really just a special case

- One generator:

```
for { x <- e1 } yield e2
```

One generator is really just a special case

- One generator:

```
for { x <- e1 } yield e2
```

- Is *really* sugar for:

```
e1.flatMap (x => unit(e2))
```

One generator is really just a special case

- One generator:

```
for { x <- e1 } yield e2
```

- Is *really* sugar for:

```
e1.flatMap (x => unit(e2))
```

- But for all monads this is equivalent to (what we saw before):

```
e1.map (x => e2)
```

One generator is really just a special case

- One generator:

```
for { x <- e1 } yield e2
```

- Is *really* sugar for:

```
e1.flatMap (x => unit(e2))
```

- But for all monads this is equivalent to (what we saw before):

```
e1.map (x => e2)
```

- Which means Scala avoids the need for an explicit unit operation.

One generator is really just a special case

- One generator:

```
for { x <- e1 } yield e2
```

- Is *really* sugar for:

```
e1.flatMap (x => unit(e2))
```

- But for all monads this is equivalent to (what we saw before):

```
e1.map (x => e2)
```

- Which means Scala avoids the need for an explicit unit operation.
- But you can imagine that ‘yield’ stands for ‘unit’.

flatMap is the monadic bind operator

flatMap is the monadic bind operator

- For Option:

```
def flatMap[B](f: A => Option[B]): Option[B] =  
  if (isEmpty) None else f(this.get)
```

flatMap is the monadic bind operator

- For Option:

```
def flatMap[B](f: A => Option[B]): Option[B] =  
  if (isEmpty) None else f(this.get)
```

```
eval(l).flatMap (x =>  
  eval(r).flatMap (y =>  
    evalOp(o,x,y))
```

If any individual statement fails,
the whole computation fails.

'for' comprehensions are overloaded

'for' comprehensions are overloaded

- Scala's 'for' comprehensions work for any type which supplies:
 - ▶ a map function (for the single generator case)
 - ▶ a flatMap function (for the nested generated case)
 - ▶ a filter function (for guarded generators)

'for' comprehensions are overloaded

- Scala's 'for' comprehensions work for any type which supplies:
 - ▶ a map function (for the single generator case)
 - ▶ a flatMap function (for the nested generated case)
 - ▶ a filter function (for guarded generators)
- Lots of types in the standard library already do:
 - ▶ Any subclass of Iterable, Parser, Option

Conclusions

- There's more to learn about monads:
 - Monad transformers allow the features of different monads to be combined.
 - The monad laws (algebraic requirements).
- Monads are probably less useful in Scala than in Haskell:
 - Scala has more primitive effects built in (I/O, state, exceptions).
- Nevertheless, the monadic style offers a new way of thinking about how to structure code.

Homework

- Look up the implementation of flatMap for Set, and Parser.
- Read James Iry's blog "Monads are Elephants" (parts 1,2,3,4).
- Extend the expression evaluation program to support variables.