

# Comp 311

# Functional Programming

Eric Allen, PhD  
Vice President, Engineering  
Two Sigma Investments, LLC

# Purely Functional Random Number Generation

```
trait RandomNumberGenerator {  
  def nextInt: (Int, RandomNumberGenerator)  
}
```

# Purely Functional Random Number Generation

```
case class SimpleRNG(seed: Int) extends RandomNumberGenerator {  
  val a = 48271  
  val b = 0  
  val m = Int.MaxValue  
  
  def nextInt: (Int, RandomNumberGenerator) = {  
    val newSeed = (a*seed + b) % m  
    val newRNG = SimpleRNG(newSeed)  
    (newSeed, newRNG)  
  }  
}
```

# Threading State Through a Sequence of Statements

```
val rng = SimpleRNG(2)
val (n1, rng1) = rng.nextInt
val (n2, rng2) = rng1.nextInt
```

# Transforming Stateful APIs to Functional APIs

```
trait Foo {  
  private var s: State = MyState  
  def bar: Bar  
  def baz: Int  
}
```

*becomes*

```
trait Foo {  
  def bar: (Bar, Foo)  
  def baz: (Int, Foo)  
}
```

# A Better API for State Actions

- Explicitly threading state from one function application to the next is tedious and error prone
- We would like to define combinators that pass the state from one application to the next automatically
- For now, we consider the state of our program to be defined entirely by the state of our random number generator

# Defining a Type Alias for State Actions

```
type StateAction[+A] =  
  RandomNumberGenerator => (A, RandomNumberGenerator)
```

# A Simple State Action

```
val nextInt: StateAction[Int] = _.nextInt
```



# A “No-Op” Abstraction Over State Actions

```
def unit[A](a: A): StateAction[A] =  
  rng => (a, rng)
```

# A “Compound” State Action

```
def nonNegativeInt(rng: RandomNumberGenerator):  
  (Int, RandomNumberGenerator) = {  
    val (n, rng2) = rng.nextInt  
    if (n == Int.MinValue) 0  
    else if (n < 0) (-n, rng2)  
    else (n, rng2)  
  }
```

# Constructing a List of Random Numbers

```
def randomInts(count: Int): StateAction[List[Int]] = { rng =>
  if (count == 0) (Nil, rng)
  else {
    val (n, rng2) = rng.nextInt
    val (ns, rngN) = randomInts(count - 1)(rng2)
    (n :: ns, rngN)
  }
}
```

# Transforming State Actions

- It is often convenient to form one state action from another by:
  - Performing the given state action
  - Applying a function to the resulting value
- We will define a combinator that constructs state actions in this way
- For no immediately obvious reason, we will name this combinator **map**

# Transforming State Actions With the Map Combinator

```
def map[A,B](s: StateAction[A])(f: A => B): StateAction[B] =  
  rng => {  
    val (a, rng2) = s(rng)  
    (f(a), rng2)  
  }
```

# Using Map

```
def nonNegativeEven: StateAction[Int] =  
  map(nonNegativeInt)(i => i - (i % 2))
```

# Random Non-Negative Numbers in a Range (Attempt 1)

// INCORRECT

```
def nonNegativeLessThan(n: Int): StateAction[Int] =  
  map(nonNegativeInt)(_ % n)
```

*This definition skews the results because  
Int.MaxValue might not be divisible by n.*

# Random Non-Negative Numbers in a Range (Attempt 2)

*// INCORRECT*

```
def nonNegativeLessThan(n: Int): StateAction[Int] =  
  map(nonNegativeInt) { i =>  
    val mod = i % n  
    if (i + (n - 1) - mod >= 0) mod  
    else nonNegativeLessThan(n)  
  }
```

*But this version does not pass type checking!*



# Random Non-Negative Numbers in a Range (Attempt 2)

- The problem with our Attempt 2 is that the recursive call to `nonNegativeLessThan` than produces a `StateAction[Int]`
- Our map combinator expects an Int result from the mapped function, not a `StateAction[Int]`
- To get a better idea as to how to define `nonNegativeLessThan`, let us try defining it without combinators

# Random Non-Negative Numbers in a Range (Attempt 3)

```
def nonNegativeLessThan(n: Int): StateAction[Int] = { rng =>
  val (i, rng2) = nonNegativeInt(rng)
  val mod = i % n
  if (i + (n - 1) - mod >= 0) (mod, rng2)
  else nonNegativeLessThan(n)(rng)
}
```

*This version works, but now we are back to threading  
state explicitly.*

*We need a new combinator.*

# Defining FlatMap on State Actions

```
def flatMap[A,B](s: StateAction[A])  
                (f: A => StateAction[B]):  
StateAction[B] = { rng =>  
  val (a, rng2) = s(rng)  
  f(a)(rng2)  
}
```

# Random Non-Negative Numbers in a Range (Attempt 4)

```
def nonNegativeLessThan(n: Int): StateAction[Int] = {  
  flatMap(nonNegativeInt) { i =>  
    val mod = i % n  
    if (i + (n - 1) - mod >= 0) (mod, _)  
    else nonNegativeLessThan(n)  
  }  
}
```

*We have almost completely eliminated state threading,  
except for one underscore.*

# Random Non-Negative Numbers in a Range (Attempt 4)


- We now have the inverse of our earlier problem:
  - Our flatMap combinator expects an **StateAction[Int]** result from the mapped function, not an Int
- We can address this problem by wrapping part of the flatMapped function in an application of the unit constructor for **StateActions**

# Random Non-Negative Numbers in a Range (Attempt 5)

```
def nonNegativeLessThan4point5(n: Int):  
StateAction[RandomNumberGenerator, Int] = {  
  nonNegativeInt.flatMap { i =>  
    val result = i % n  
    if (i + (n - 1) - result >= 0) unit(result)  
    else nonNegativeLessThan5(n)  
  }  
}
```

# Random Non-Negative Numbers in a Range (Attempt 5)

```
def nonNegativeLessThan4point5(n: Int):  
StateAction[RandomNumberGenerator, Int] = {  
  nonNegativeInt.flatMap { i =>  
    val result = i % n  
    if (i + (n - 1) - result >= 0) unit(result)  
    else nonNegativeLessThan5(n)  
  } map (j => j)  
}
```



*A trailing map of the identity function defines  
an equivalent function.*

# Using For-Expression Syntax

- Our final attempt at `nonNegativeLessThan` involved a `flatMap` of a `map`
  - This is exactly the form of expression that `for`-expression syntax can be used for
- Let's redefine `StateAction` as a class with `map` and `flatMap` methods so we can use `for`-syntax
- We can also generalize `StateActions` to work over arbitrary state, not just `RandomNumberGenerators`



# A General StateAction Class

```
case class StateAction[S,+A](run: S => (A,S))
extends Function1[S,(A,S)] {
  def apply(s:S) = run(s)

  def map[B](f: A => B): StateAction[S,B] = StateAction { s =>
    val (a, s2) = run(s)
    (f(a), s2)
  }

  def flatMap[B](f: A => StateAction[S,B]): StateAction[S,B] =
    StateAction { s =>
      val (a, s2) = run(s)
      f(a)(s2)
    }
}
```

# Every Partial Application of the StateAction Type Defines a Monad

```
type RNGStateAction[A] =  
  StateAction[RandomNumberGenerator, A]
```

# The Unit Constructor for StateActions

```
def unit[S,A](a: A): StateAction[S,A] =  
  StateAction[S,A](s => (a, s))
```

# The Unit Constructor for RNGStateActions

```
def rngUnit[A](a: A): RNGStateAction[A] =  
  StateAction(s => (a, s))
```

# Reformulating nextInt as a State Action

```
val nextInt =  
  StateAction {  
    (rng: RandomNumberGenerator) => rng.nextInt  
  }
```

# Reformulating nonNegativeInt as a State Action

```
def nonNegativeInt: RngStateAction[Int] =  
  StateAction {  
    rng =>  
      val (n, rng2) = rng.nextInt  
      if (n == Int.MinValue) nonNegativeInt(rng2)  
      else if (n < 0) (-n, rng2)  
      else (n, rng2)  
  }
```

# Revisiting nonNegativeLessThan

```
def nonNegativeLessThan(n: Int):  
  StateAction[RandomNumberGenerator, Int] = {  
    nonNegativeInt.flatMap { i =>  
      val result = i % n  
      if (i + (n - 1) - result >= 0) rngUnit(result)  
      else nonNegativeLessThan(n)  
    } map (j => j)  
  }
```

# Using For-Expression Syntax

```
def nonNegativeLessThan(n: Int): RngStateAction[Int] = {  
  for {  
    rand <- nonNegativeInt  
    result <- {  
      val randN = rand % n  
      if (rand + (n - 1) - randN >= 0) rngUnit(randN)  
      else nonNegativeLessThan(n)  
    }  
  }  
  yield result  
}
```



# Revisiting RollDie

```
def rollDie: StateAction[Int] = nonNegativeLessThan(6)
```

# Revisiting RollDie

```
def rollDie: StateAction[Int] =  
  map(nonNegativeLessThan(6))(_ + 1)
```

# Revisiting RollDie

```
def rollDie =  
  for {  
    i <- nonNegativeLessThan(6)  
  }  
  yield (i + 1)
```

# Mechanical Proof Checking

# Syntax of Propositional Logic

$$\begin{array}{l} S ::= x \\ | S \wedge S \\ | S \vee S \\ | S \rightarrow S \\ | \neg S \end{array}$$

# Factory Methods for Construction

```
case object Formulas {  
  def var(name: String): Formula  
  def and(left: Formula, right: Formula): Formula  
  def or(left: Formula, right: Formula): Formula  
  def implies(left: Formula, right: Formula): Formula  
  def not(body: Formula): Formula  
}
```

# Sequents

$$S * \vdash S$$

# Inference Rules

$$\frac{Q^*}{Q}$$



# Example Inference Rule

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{\Gamma \cup \Delta \vdash p \wedge q} \text{ [And-Intro]}$$

# More Inference Rules

$$\frac{\Gamma \vdash p \wedge q}{\Gamma \vdash p} \quad [\text{And-Elim-Left}]$$

$$\frac{\Gamma \vdash q \wedge p}{\Gamma \vdash p} \quad [\text{And-Elim-Right}]$$

# Rule Application

```
case object Rules {  
  def identity(p: Formula): Sequent  
  def assumption(s: Sequent): Sequent  
  def generalization(p: Formula)(s: Sequent): Sequent  
  def andIntro(left: Sequent, right: Sequent): Sequent  
  def andElimLeft(s: Sequent): Sequent  
  def andElimRight(s: Sequent): Sequent  
  def orIntroLeft(p: Formula)(s: Sequent): Sequent  
  def orIntroRight(p: Formula)(s: Sequent): Sequent  
  def orElim(s0: Sequent, s1: Sequent, s2: Sequent): Sequent  
  def negIntro(p: Formula)(s0: Sequent, s1: Sequent): Sequent  
  def negElim(s: Sequent): Sequent  
  def impliesIntro(s: Sequent): Sequent  
  def impliesElim(p: Formula)(s: Sequent): Sequent  
}
```