@AndrzejWasowski
Andrzej Wąsowski

# Advanced Programming

## State Monad

IT UNIVERSITY OF COPENHAGEN
PROCESS AND SYSTEM MODELS GROUP

# A Typical Stateful Imperative API

```
def rollDie: Int = {
  val rng = new scala.util.Random
  rng.nextInt(6)
}
```

Returns a random number from 0 to 5

- We call `rollDie` and observe a value `5`
- Mentimeter: What is the result of `rollDie + rollDie`?
- What does it tell us about referential transparency of `rollDie`?
- To make `rollDie` referentially transparent, make the state explicit

# Converting RNG to explicit state

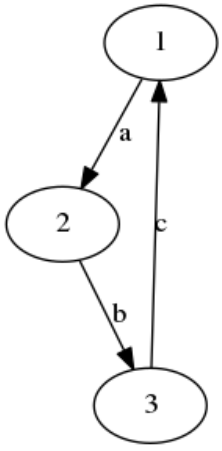
- We had: `RNG.nextInt: () =>Int`
- Let the function return a new state explicitly, instead of modifying the old

```
trait RNG { def nextInt: (Int, RNG) }
object RNG{
  def nextInt (rng: RNG) : (Int, RNG) = rng.nextInt
}
```

- In general we have a function: `State =>(Output,State)`
- The book calls this type `case class State[S,+A] (run: S =>(A,S) )`
- So we can implement `RNG` as `State[RNG,Int] { run =RNG.nextInt }`
- Intuition: perhaps `Automaton` or `Transition` would be better names than `State`
- Intuition: perhaps `step` would be a better name than `run`

# Consider a Simple Automaton

## Stateful **by definition**




```
var state = 1
while (true)
  state match {
    case 1 => { print "a"; state = 2 }
    case 2 => { print "b"; state = 3 }
    case 3 => { print "c"; state = 1 }
  }
```

```
def step (State: Int): (String,Int) = state match {
 case 1 => ( "a", 2 )
 case 2 => ( "b", 3 )
 case 3 => ( "c", 1 )
}
```

- We need a simple recursive loop to run the step like above
- This loop can be general for the `State` type
- This automaton as a state instance: `State[Int,String] (step _)`

# Exercise

Use 5 minutes to write down an instance of `State` implementing this imperative code

```
x = 0
white (true) {
  println (x)
  x += 1
}
```

# States vs Streams

- We can unroll a state machine from an initial state, producing a stream of actions.
- Discuss: What is the stream from our first automaton?
- Discuss: What is the stream from our second automaton?
- Mentimeter: another stream
- We have an exercise implementing this mapping as a function
- Laziness of streams is useful here, why?

# Anything stateful maps to the state pattern

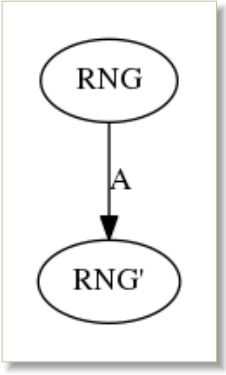
## Recap

- Random number generators (state: RNG seed)
- Websites with modality (session state)
- Database backed applications (DB state)
- Communication protocols (protocol state)
- etc.

# Random Number Generator as an Instance of State

```
type Rand[A] =State[RNG,A]
```

- RNG is the state of the random generator
  (usually some large number encapsulated)
- The textbook gives a simple implementation of RNG
  based on multiplication with large primes module 64 bits
- `Rand[A]` is a **computation** that we can run,
  then it will produce a random `A` and a new state `RNG`
- Another useful intuition: `Rand[A]` is **a generator of random `A`'s**
- Or even just a **"random `A`"**
- **Question:**
  What is the stream we can unfold from `State[Int] (_.nextInt)`?



# How do I use this generator of random numbers?

```
type Rand[A] =State[RNG,A]
val r : Rand[Int] =SimpleRNG (42)
val (r1,i) =r.run
```

- `SimpleRNG` is the book's concrete implementation of the `RNG` trait
- `42` is the initial seed (state)
- `(r1,i)` is a new state and a random number
- **Question:**
  How do I get the next random number?
  What happens if I call `r.run` again?

# What can we do with Automata/State?

State is a monad; mostly same key operations as for `List`, `Option`, and `Stream`

```
def map[S,A,B] (s: State[S,A]) (f: A =>B): State[S,B]
```

We can use this to generate even numbers:

```
val even: Rand[Int] =map[Int] (r) (n =>n % 2)
```

# Automata can be composed [1/2]

`flatMap` can be used to compose generators:

```
def flatMap[S,A,B] (s: State[S,A]) (f : A =>State[S,B]): State[S,B]
```

In our context of generators:

```
def flatMap[A,B] (r :Rand[A]) (f =>Rand[B]) : Rand[B] =
  flatMap[A,B] (r: State[RNG,A]) (f :State[RNG,B]) :State[RNG,B]
```

`flatMap` can compose generators (compute a random size list of random even integers):

```
val int :Rand[Int] =... (assume you have it)
val ints: Int =>Rand[List[Int]] =... (assume creates a random list of given length)
val ns :Rand[List[Int]] =int.flatMap( x =>ints(x).map (xs =>xs.map (_*2))
```

the state RNG passed **implicitly**; size generated with different state than each number

# Automata can be composed [2/2]

The `map2` function can compute a zipping of two automata over the same state space for us:

```
map2 [S,A,B,C] (sa: State[S,A]) (sb: State[S,B]) (f: (A,B)=>C) :State[S,C]
```

Could be used to create a product automaton
– interleaving computations, then `C` is `Either[A,B]`
– synchronizing two computations, then `C` is `(A,B)`

More fun in exercises :)

# Next week

- Next week we will design a parallel computation library, in purely functional style
- This shows (a bit) how Akka is implemented
- In two weeks, we will use the generators of random numbers to implement a modern testing framework
- So: keep reading the chapters and solve the exercises!