

# Comp 311

# Functional Programming

Eric Allen, Two Sigma Investments  
Robert “Corky” Cartwright, Rice University  
Sağnak Taşırılar, Two Sigma Investments

Streams

# Streams

- a form of “lazy” sequence
- inspired by signal-processing (e.g. digital circuits)
- Components accept *streams* of signals as input, transform their input, and produce streams of signals as outputs

# Streams

```
abstract class Stream[+T] {  
  def head(): T  
  def tail(): Stream[T]  
  def map[S](f: T => S): Stream[S]  
  def flatMap[S](f: T => Stream[S]): Stream[S]  
  def ++[S >: T](that: Stream[S]): Stream[S]  
  def withFilter(f: T => Boolean): Stream[T]  
  def nth(n: Int): T  
}
```

# Streams

```
case object NilStream extends Stream[Nothing] {
  def head() = throw new Error()
  def tail() = throw new Error()
  def map[S](f: Nothing => S): Stream[S] = NilStream
  def flatMap[S](f: Nothing => Stream[S]): Stream[S] =
    NilStream
  def ++[S >: Nothing](that: Stream[S]) = that
  def withFilter(f: Nothing => Boolean) = NilStream
  def nth(n: Int) = throw new Error()
}
```

# Streams

```
case class ConsStream[+T](head: T, lazyTail: () => Stream[T])
extends Stream[T] {
  def tail = lazyTail()
  def map[S](f: T => S): Stream[S] =
    ConsStream(f(head), () => (tail map f))
  def flatMap[S](f: T => Stream[S]): Stream[S] =
    f(head) ++ tail.flatMap(f)
  def ++[S >: T](that: Stream[S]): Stream[S] =
    ConsStream(head, () => tail ++ that)
  ...
}
```

# Streams

```
case class ConsStream[+T](head: T, lazyTail: () => Stream[T])
extends Stream[T] {
  ...
  def withFilter(f: T => Boolean) = {
    if (f(head)) ConsStream(head, () => tail.withFilter(f))
    else tail.withFilter(f)
  }
  def nth(n: Int) = {
    require (n >= 0)
    if (n == 0) head
    else tail.nth(n - 1)
  }
}
```

# Streams

```
def range(low: Int, high: Int): Stream[Int] =  
  if (low > high) NilStream  
  else ConsStream(low, () => range(low + 1, high))
```



# Streams

```
def intsFrom(n: Int): Stream[Int] =  
  ConsStream(n, () => intsFrom(n + 1))
```

# Streams

```
val nats = intsFrom(0)
```

# Streams

```
def fibGen(a: Int, b: Int): Stream[Int] =  
  ConsStream(a, () => fibGen(b, a + b))
```

# Streams

```
val fibs = fibGen(0, 1)
```

# Streams

```
def push(x: Int, ys: Stream[Int]) = {  
  ConsStream(x, () => ys)  
}
```

# Streams

```
def isDivisible(m: Int, n: Int) = (m % n == 0)
```

```
val noSevens = nats withFilter (isDivisible(_, 7))
```

# A Prime Sieve

```
def sieve(stream: Stream[Int]): Stream[Int] =  
  ConsStream(stream.head,  
    () => sieve(stream.tail withFilter  
      (x => !(isDivisible  
        (x, stream.head))))))
```

# Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10	Prime numbers
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	49	50	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	

[https://en.wikipedia.org/wiki/Sieve\\_of\\_Eratosthenes](https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes)

CC BY-SA 3.0

<https://commons.wikimedia.org/w/index.php?curid=2810935>



# A Stream of Primes

```
val primes = sieve(intsFrom(2))
```

# A Stream of Primes

```
> primes.head
```

```
res5: Int = 2
```

```
> primes.nth(1)
```

```
res6: Int = 3
```

```
> primes.nth(2)
```

```
res7: Int = 5
```

```
> primes.nth(3)
```

```
res8: Int = 7
```

# Streams

```
def add(xs: Stream[Int],
       ys: Stream[Int]) : Stream[Int] = {

  (xs, ys) match {
    case (NilStream, _) => ys
    case (_, NilStream) => xs
    case (ConsStream(x, f), ConsStream(y, g)) =>
      ConsStream(x + y, () => add(f(), g()))
  }
}
```

# Streams

```
def ones(): Stream[Int] = ConsStream(1, ones)
```

# Alternative Definition of the Stream of Natural Numbers

```
def nats(): Stream[Int] =  
  ConsStream(0, () => add(ones, nats))
```

# Alternative Definition of the Fibonacci Stream

```
def fibs(): Stream[Int] =  
  ConsStream(0,  
    () => ConsStream(1,  
      () => add(fibs.tail, fibs)))
```

# Powers of Two

```
def scaleStream(c: Int, stream: Stream[Int]): Stream[Int] =  
  stream map (_ * c)
```

```
def powersOfTwo(): Stream[Int] =  
  ConsStream(1, () => scaleStream(2, powersOfTwo))
```

# Alternative Definition of the Stream of Primes

```
def primes() =  
  ConsStream(2, () => intsFrom(3) withFilter isPrime)  
  
def isPrime(n: Int): Boolean = {  
  def sieve(next: Stream[Int]): Boolean = {  
    if (square(next.head) > n) true  
    else if (isDivisible(n, next.head)) false  
    else sieve(next.tail)  
  }  
  sieve(primes)  
}
```



# Numeric Integration with Streams

$$S_i = c + \sum_{j=1}^i x_j dt$$

# Numeric Integration with Streams

```
def integral(integrand: Stream[Double],
            init: Double,
            dt: Double) = {

  def inner(): Stream[Double] = {
    ConsStream(init,
              () => addStreams(scaleStream(dt,
                                          integrand),
                              inner))
  }
  inner
}
```

# Streams and Local State

```
def withdraw(balance: Int, amounts: Stream[Int]):  
  Stream[Int] = {  
    ConsStream(balance,  
                () => withdraw(balance - amounts.head,  
                               amounts.tail))  
  }
```

# Discussion

- Our modeling of a bank account is a purely functional program without state
- Nevertheless:
  - If a user provides the stream of withdrawals, and
  - The stream of balances is displayed as outputs,
- The system will behave from a user's perspective as a stateful system

# Discussion

- The key to understanding this paradox is that the “state” is in the world:
  - The user/bank system is stateful and provides the input stream
  - If we could “step outside” our own perspective in time, we could view our withdrawal stream as another stateless stream of transactions

# Changing the State of Variables

# Changing the State of Variables

- Thus far, we have focused solely on purely functional programs
- This approach has gotten us remarkably far
- Sometimes, it is difficult to structure a program without some notion of stateful variables:
  - I/O, GUIs
  - Modeling a stateful system in the world

# Assignment and Local State

- We view the world as consisting of objects with state that changes over time
- It is often natural to model physical systems with computational objects with state that changes over time



# Assignment and Local State

- If we choose to model the flow of time in the system by elapsed time in the computation, we need a way to change the state of objects as a program runs
- If we choose to model state using symbolic names in our program, we need an assignment operator to allow for changing the value associated with a name

# Modeling an Address Book

```
class AddressBook() {  
  val addresses: Map[String,String] = Map()  
  
  def put(name: String, address: String) = {  
    ...  
  }  
  
  def lookup(name: String) = addresses(name)  
}
```

# Modeling an Address Book

```
class AddressBook() {  
  var addresses: Map[String,String] = Map()  
  
  def put(name: String, address: String) = {  
    addresses = addresses + (name -> address)  
  }  
  
  def lookup(name: String) = addresses(name)  
}
```

You now saw `var`; you are still not allowed to use it :)

# Sameness and Change

- In the context of assignment, our notion of equality becomes far more complex

```
val petersAddressBook = new AddressBook()  
val paulsAddressBook = new AddressBook()
```

```
val petersAddressBook = new AddressBook()  
val paulsAddressBook = paulsAddressBook
```

# Sameness and Change

- Effectively assignment forces us to view names as referring not to values, but to *places* that store values

# Referential Transparency

- The notion that equals can be substituted for equals in an expression without changing the value of the expression is known as *referential transparency*
- Referential transparency is one of the distinguishing aspects of functional programming
- It is lost as soon as we introduce assignment

# Referential Transparency

- Without referential transparency, the notion of what it means for two objects to be “the same” is far more difficult to explain
- One approach:
  - Modify one object and see whether the other object has changed in the same way

# Referential Transparency

- One approach:
  - Modify one object and see whether the other object has changed in the same way
  - But that involves observing a single object twice
  - How do we know we are observing the same object both times?



# Pitfalls of Imperative Programming

- The order of updates to variables is a classic source of bugs

```
def factorial(n: Int) = {  
  var product = 1  
  var counter = 1  
  def iter(): Int = {  
    if (counter > n) {  
      product  
    }  
    else {  
      product = product * counter  
      counter = counter + 1  
      iter()  
    }  
  }  
  iter()  
}
```

```
def factorial(n: Int) = {  
  var product = 1  
  var counter = 1  
  def iter(): Int = {  
    if (counter > n) {  
      product  
    }  
    else {  
      product = product * counter  
      counter = counter + 1  
      iter()  
    }  
  }  
  iter()  
}
```

*What if the order of these updates  
were reversed?*

# Review: The Environment Model of Evaluation

- Environments map names to values
- Every expression is evaluated in the context of an environment

# The Environment Model of Reduction

- To evaluate a name, simply reduce to the value it is mapped to in the environment

# The Environment Model of Reduction

- To evaluate a function, reduce it to a *closure*, which consists of two parts:
  - The body of the function
  - The environment in which the body occurs

# The Environment Model of Reduction

- Objects are also modeled as closures
  - What is the environment?
  - What corresponds to the body of the function?

# The Environment Model of Reduction

- To evaluate an application of a closure
  - Extend the environment of the closure, mapping the function's parameters to argument values
  - Evaluate the body of the closure in this new environment



# Variable Rebinding in the Environment Model

- The environment model provides us with the necessary machinery to model stateful variables
- To evaluate a variable  $v$  assignment:
  - Rebind the value  $v$  maps to in the environment in which the assignment occurs

# Rebinding a Variable in an Environment

- The rebound value of  $v$  is then used in all subsequent reductions involving the same environment
  - Includes closures involving that environment
- This model of variable assignment pushes the notion of state out to environments
- The “places” referred to by variables are simply components of environments

# Example: Pseudo-Random Number Generation

- There are many approaches to generating a pseudo-random stream of **Int** values
- One common approach is to define a *linear congruential generator (LCG)*:

$$X_{n+1} = (aX_n + c) \bmod m$$

- The pseudo-random numbers are the elements of this recurrence

# Linear Congruential Generators

- LCGs can produce generators capable of passing formal tests for randomness
- The quality of the results is highly dependent on the initial values selected
- Poor statistical properties
- Not well suited for cryptographic purposes

# A Linear Congruent Generator (C++11 `minstd_rand`)

```
def makeRandomGenerator(): () => Int = {  
    val a = 48271  
    val b = 0  
    val m = Int.MaxValue  
    var seed = 2  
  
    def inner() = {  
        seed = (a*seed + b) % m  
        seed  
    }  
    inner  
}
```

# A Linear Congruent Generator (C++11 `minstd_rand`)

```
val g = makeRandomGenerator()<E> ⇨  
val g =  
< def inner() = {  
    seed = (a*seed + b) % m  
    seed  
} ,  
val a = 48271  
val b = 0  
val m = Int.MaxValue  
var seed = 2 >
```

$g() \langle E \rangle \mapsto$

```
< def inner() = {  
    seed = (a*seed + b) % m  
    seed  
}  
 ,  
val a = 48271  
val b = 0  
val m = Int.MaxValue  
var seed = 2 >() <E> \mapsto
```

seed = (a\*seed + b) % m

seed,

```
< val a = 48271
```

```
  val b = 0
```

```
  val m = Int.MaxValue
```

```
  var seed = 2 >
```

↳

seed = (48271\*2 + 0) % Int.MaxValue

seed,

```
< val a = 48271
```

```
  val b = 0
```

```
  val m = Int.MaxValue
```

```
  var seed = 2 >
```

↳



```
seed, <val a = 48271  
      val b = 0  
      val m = Int.MaxValue  
      var seed = 96542>
```

↳

96542

```
seed, <val a = 48271  
      val b = 0  
      val m = Int.MaxValue  
      var seed = 96542>
```

↳

96542



*And now the environment closing over generator  $g$  binds  $seed$  to 96542.*