PLRG

CS320 - Scala Seminar

Streams

Jaemin Hong

hjm0901@gmail.com

October 31, 2018

PLRG

Review

```scala
trait List

case object Nil extends List

case class Cons(head: Int, tail: List) extends List
```

Review

```
def inc1(l: List): List =
  list_map(l, _ + 1)


def odd(l: List): List =
  list_filter(l, _ % 2 != 0)
```

PLRG

Review

PLRG

```
def sum(l: List): Int =
  list_foldRight(l, 0, _ + _)
```

```
def sum(l: List): Int =
  list_foldLeft(l, 0, _ + _)
```

Review

```scala
trait Option

case object None extends Option

case class Some(n: Int) extends Option
```

Expression Sequencing

PLRG

- Can sequence multiple expressions using line breaks and curly braces.
- Use semicolons to write multiple expressions in a single line.
- Evaluate expressions sequentially and keep the result of the last expression.

```
{ [expr]
  [expr] }

{ [expr]; [expr]; [expr] }
```

PLRG

Expression Sequencing

```
scala> { println("evaluated!")
         1 }
evaluated!
res: Int = 1
scala> { println("evaluated!"); 1 }
evaluated!
res: Int = 1
```

PLRG

Lazy Values

- Evaluate an expression when a variable is referred.
- Evaluate an expression at most once.
- Lazy values are 'call-by-need'.

```
lazy val [id] = [expr]
```

PLRG

Lazy Values

```
scala> val x = { println("evaluated!"); 1 }

evaluated!

x: Int = 1



scala> x + x

res: Int = 2
```

PLRG

Lazy Values

```
scala> lazy val x = { println("evaluated!"); 1 }

x: Int = <lazy>



scala> x + x

evaluated!

res: Int = 2
```

PLRG

Lazy Values

```
scala> def x() = { println("evaluated!"); 1 }
x: ()Int

scala> x() + x()
evaluated!
evaluated!
res: Int = 2
```

PLRG

By-Name Parameters

- Evaluate an argument when the variable is referred in the body of a function.
- Evaluate an expression zero or more times.
- By-name parameters are 'call-by-name'.

```
def [id]([id]: => [type]) = [expr]
```

PLRG

By-Name Parameters

```
scala> def f(x: Int) = 0
f0: (x: Int)Int


scala> f({ println("evaluated!"); 1 })
evaluated!
res: Int = 0
```

By-Name Parameters

PLRG

```
scala> def f(x: => Int) = 0

f0: (x: Int)Int



scala> f({ println("evaluated!"); 1 })

res: Int = 0
```

By-Name Parameters

PLRG

```
scala> def f(x: Int) = x + x

f0: (x: Int)Int



scala> f({ println("evaluated!"); 1 })

evaluated!

res: Int = 2
```

By-Name Parameters

PLRG

```
scala> def f(x: => Int) = x + x
f0: (x: Int)Int

scala> f({ println("evaluated!"); 1 })
evaluated!
evaluated!
res: Int = 2
```

Inefficiency of Lists

PLRG

- Find the first prime in a given interval of integers.
    - Return a Some of the first prime if it exists.
    - Return the None otherwise.

    - Function isPrime, which takes one integer as a parameter and returns whether it is prime or not, is given.

Inefficiency of Lists

PLRG

- An imperative solution:

```
def findFirstPrime(start: Int, end: Int): Option[Int] = {
  var x = start
  while (x <= end) {
    if (isPrime(x)) return Some(x)
    x += 1
  }
  return None
}
```

Inefficiency of Lists

PLRG

- A functional solution with a list:

```
def findFirstPrime(start: Int, end: Int): Option[Int] =
  (start to end).filter(isPrime).headOption
```

Inefficiency of Lists

PLRG

- Find the nth prime number.
    - Take integer n as a parameter and return the nth prime number.
    - Function isPrime, which takes one integer as a parameter and returns whether it is prime or not, is given.

Inefficiency of Lists

- An imperative solution:

```
def nthPrime(n: Int): Int = {
  var x = 1; var counter = 0
  while (counter < n) {
    x += 1
    if (isPrime(x)) counter += 1
  }
  x
}
```

Inefficiency of Lists

PLRG

- A functional solution with a list:

```
def nthPrime(n: Int): Int =

  (2 to ???).filter(isPrime)(n)
```

Inefficiency of Lists

PLRG

- A functional solution with a list:

```
def nthPrime(n: Int): Int =
  (2 to Int.MaxValue).filter(isPrime)(n)
```

Streams

- To solve it, define a list with a lazy tail.
- Evaluate only head when it is created.
- Tail is evaluated when it is needed.

- We call it "Stream".

Streams

- Streams can be finite like lists.
- However, they can be infinite as well.
- Today, we will define our own integer infinite streams using Scala.

PLRG

Defining Streams

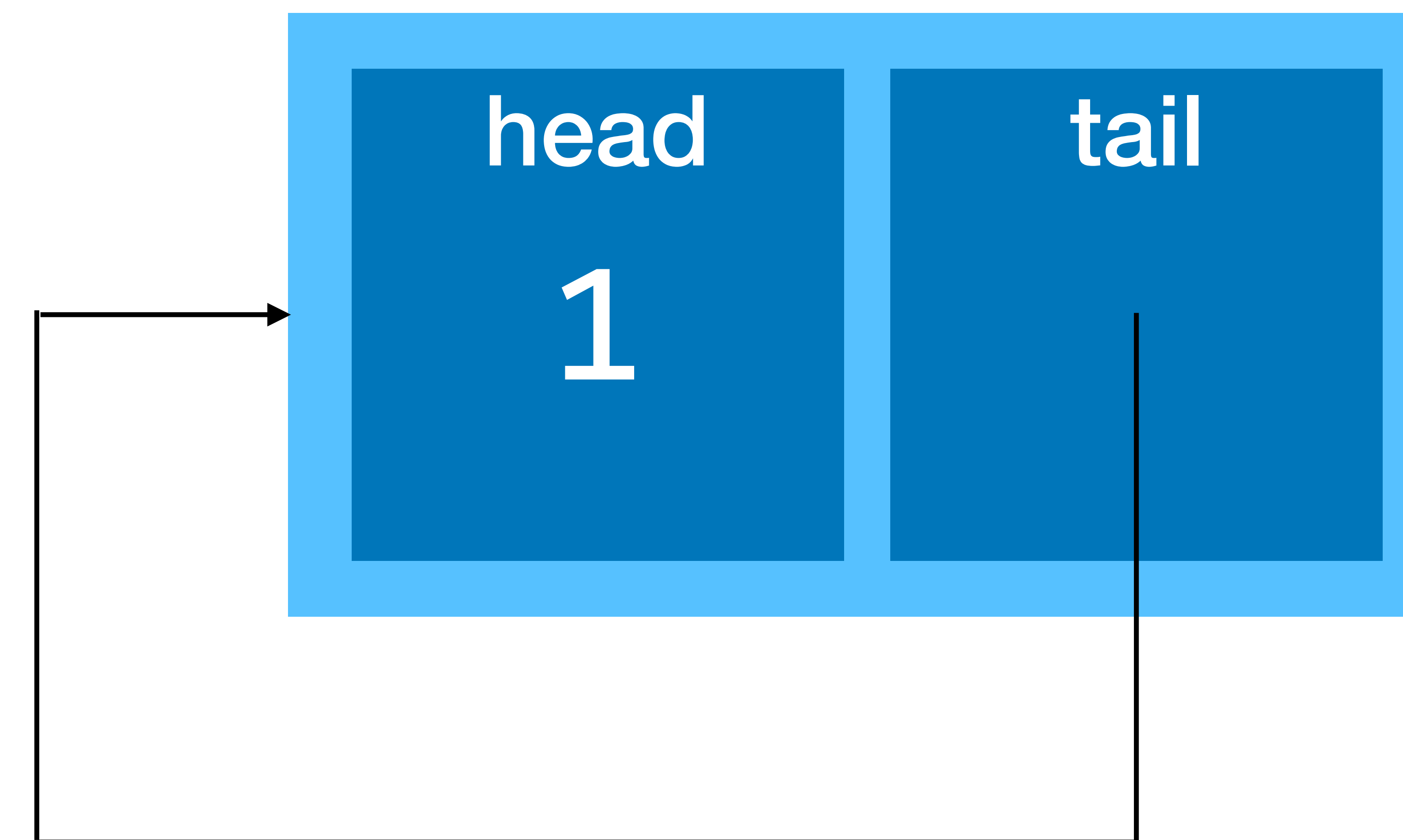
```
class Stream(val head: Int, tl: => Stream) {
  lazy val tail = tl
}

def Stream(hd: Int, tl: => Stream): Stream =
  new Stream(hd, tl)
```

Making a Stream

PLRG

- Define stream `ones`, whose every element is one.

```
lazy val ones: Stream =
```



Printing Streams

PLRG

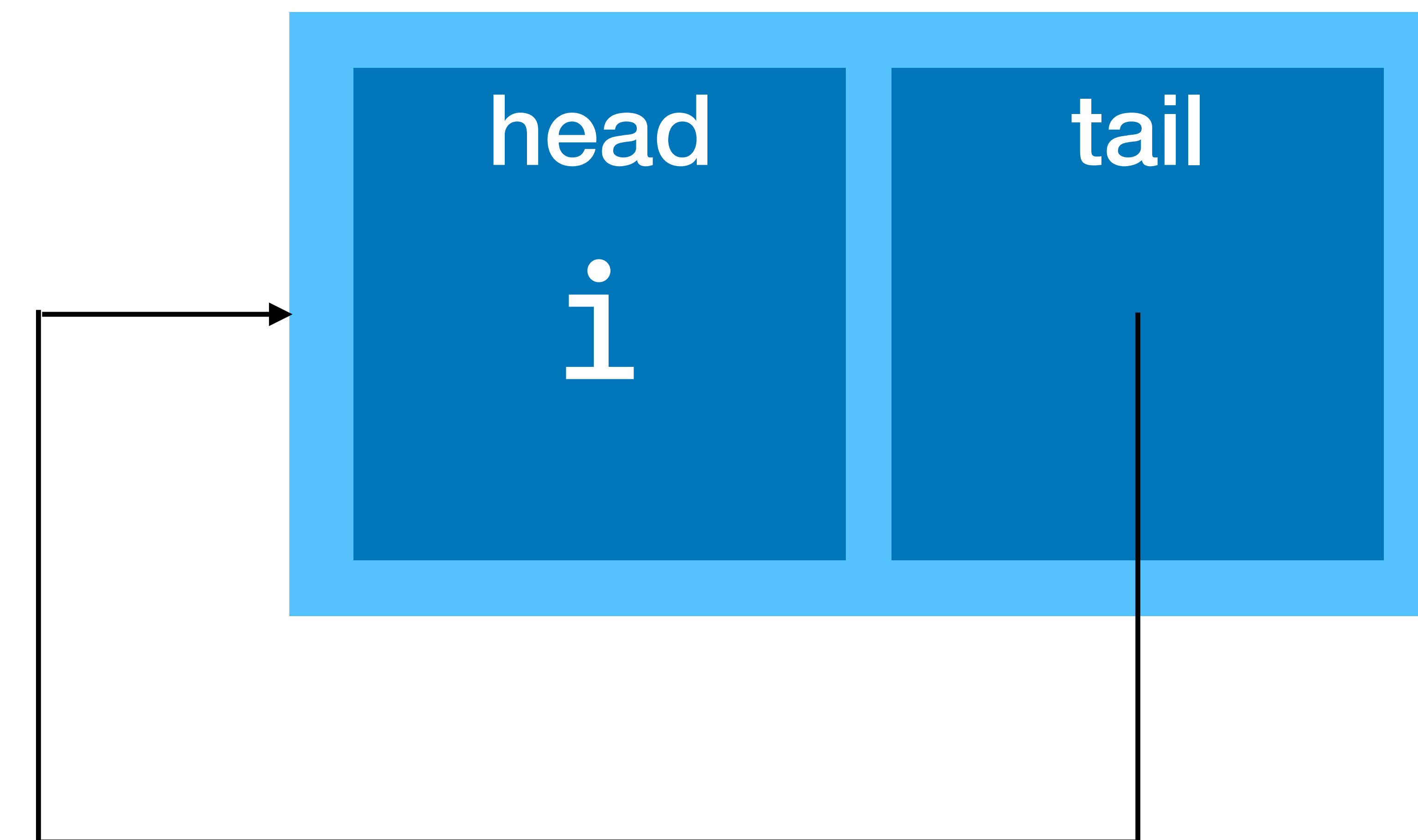
```
def print(st: Stream, n: Int): Unit =
  if (n > 0) {
    println(st.head)
    print(st.tail, n - 1)
  }
```

Constant Streams

PLRG

- Define function `const`, which
  - takes integer `i` as a parameter and
  - returns a stream, whose every element is `i`.

```
def const(i: Int): Stream =
```



Adding Streams

```
a = a₀, a₁, a₂, ...

b = b₀, b₁, b₂, ...

add(a, b) = (a₀ + b₀), (a₁ + b₁), (a₂ + b₂), ...


def add(st0: Stream, st1: Stream): Stream =
```

PLRG

Defining Natural Numbers

- Define stream `nats`, whose ith element is i.

```
nats = 0, 1, 2, 3, ...

     = 0, (0 + 1), (1 + 1), (2 + 1), ...

     = 0, ((0, 1, 2, ...) + (1, 1, 1, ...))


lazy val nats: Stream =
```

Getting Elements of Streams

```
def get_stream(st: Stream, i: Int): Int =
  if (i <= 0) st.head
  else get_stream(st.tail, i - 1)
```

Streams with Higher-Order Functions

PLRG

- Define function `map_stream`, a "stream version" of the map function.

```
f(a) = f(a₀), f(a₁), f(a₂), f(a₃), ...
```

```
def map_stream(st: Stream, f: Int => Int): Stream =
```

```
lazy val nats: Stream =
```

Generalizing Natural Numbers

PLRG

- A recurrence equation:

$$a_0 = n$$

$$a_i = f(a_{i-1}) \text{ for } i > 0$$

$$a = a_0, a_1, a_2, a_3, \ldots$$
$$= n, f(a_0), f(a_1), f(a_2), \ldots$$
$$= n, f((a_0, a_1, a_2, \ldots))$$

Generalizing Natural Numbers

$$a = a_0,\ a_1,\ a_2,\ a_3,\ ...$$

$$= n,\ f(a_0),\ f(a_1),\ f(a_2),\ ...$$

$$= n,\ f((a_0,\ a_1,\ a_2,\ ...))$$

```
def recurrence1(n: Int, f: Int => Int): Stream =


val nats: Stream =
```

Fibonacci Numbers

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-2} + F_{i-1}$ for $i > 1$

$F = 0, 1, F_0 + F_1, F_1 + F_2, ...$

$= 0, 1, ((F_0, F_1, ...) + (F_1, F_2, ...))$

Fibonacci Numbers

```
F = 0, 1, a₀ + a₁, a₁ + a₂, ...

  = 0, 1, ((a₀, a₁, ...) + (a₁, a₂, ...))
```

```
lazy val fibs: Stream =
```

The nth Fibonacci number: `get_stream(fibs, n)`

Generalizing Function add

PLRG

- Define function `map2_stream`:

$$f(a, b) = f(a_0, b_0), f(a_1, b_1), f(a_2, b_2), f(a_3, b_3), \ldots$$

```
def map2_stream(
  st0: Stream, st1: Stream, f: (Int, Int) => Int
): Stream =

lazy val fibs: Stream =
```

Approximating the Golden Ratio

PLRG

F_{n+1}/F_n converges to the golden ratio when n goes to the infinity.

```
val goldenApprox: Stream =
  map2_stream(
    map_stream(fibs.tail.tail, _ * 10000),
    fibs.tail,
    _ / _
  )
```

Generalizing Fibonacci Numbers

PLRG

- A recurrence equation:

$$a_0 = n$$

$$a_1 = m$$

$$a_i = f(a_{i-2}, a_{i-1}) \text{ for } i > 1$$

$$a = a_0, a_1, a_2, a_3, \ldots$$

$$= n, m, f(a_0, a_1), f(a_1, a_2), \ldots$$

$$= n, m, f((a_0, a_1, \ldots), (a_1, a_2, \ldots))$$

Generalizing Fibonacci Numbers

PLRG

```
a = a₀, a₁, a₂, a₃, ...
  = n, m, f(a₀, a₁), f(a₁, a₂), ...
  = n, m, f((a₀, a₁, ...), (a₁, a₂, ...))

def recurrence2(
  n: Int, m: Int, f: (Int, Int) => Int
): Stream =

val fibs: Stream =
```

Series

- A series of a sequence:

$$s_i = a_0 + a_1 + \ldots + a_i$$

$$s = s_0, s_1, s_2, s_3, \ldots$$

$$= a_0, a_0 + a_1, a_0 + a_1 + a_2, a_0 + a_1 + a_2 + a_3, \ldots$$

$$= a_0, s_0 + a_1, s_1 + a_2, s_2 + a_3, \ldots$$

$$= a_0, ((s_0, s_1, s_2, \ldots) + (a_1, a_2, a_3, \ldots))$$

Series

$$s = s_0,\ s_1,\ s_2,\ s_3,\ \ldots$$

$$= a_0,\ a_0 + a_1,\ a_0 + a_1 + a_2,\ a_0 + a_1 + a_2 + a_3,\ \ldots$$

$$= a_0,\ s_0 + a_1,\ s_1 + a_2,\ s_2 + a_3,\ \ldots$$

$$= a_0,\ ((s_0,\ s_1,\ s_2,\ \ldots) + (a_1,\ a_2,\ a_3,\ \ldots))$$

```
def series(st: Stream): Stream =
```

Approximating pi

PLRG

$1/1^2 + 1/2^2 + 1/3^2 + 1/4^2 + \ldots$ converges to $pi^2/6$

```
val piApprox: Stream =
  map_stream(
    series(
      map_stream(
        nats.tail,
        n => 600000000 / n / n
      )
    ),
    Math.sqrt(_).toInt
  )
```

Prime Numbers

PLRG

- Define function `filter_stream`, a "stream version" of the filter function.

```
def map_stream(st: Stream, f: Int => Int): Stream =
```

```
val primes: Stream =
```

The nth prime number: `get_stream(primes, n)`

Mersenne Prime Numbers

PLRG

- Mersenne primes are primes of the form $2^n - 1$.

```
val mersennes: Stream =
  filter_stream(
    map_stream(
      recurrence1(1, _ * 2),
      _ - 1
    ),
    isPrime
  )
```

Summary

- Scala offers a notion of laziness by lazy values and by-name parameters.
- A stream is a lazy list-like data structure, whose head is strict but tail is lazy.
- We can implement infinite data collections using streams.
- Streams become more useful when they are used with higher-order functions.