



今天的内容



- Stream
- Iterators
- Lazy values

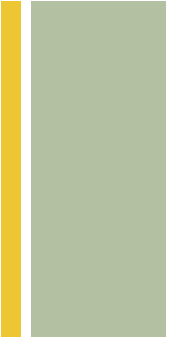
+ Computing with Streams

- Change with time – modeled by changing state of variables
- Variables and assignment → Price: simple and powerful substitution model for functional computation not applicable
- Model **state change** only by **immutable functions**?
 - Mathematics says Yes.
 - Time-changing quantity → function $f(t)$, t : time parameter
- `Var x:T → val x : List[T]`
 - Trade space for time
 - Advantage: “time-travel”; use list’s functions

```
def sumPrimes(start: Int, end: Int) =  
  sum(range(start, end) filter isPrime)
```



Less efficiency

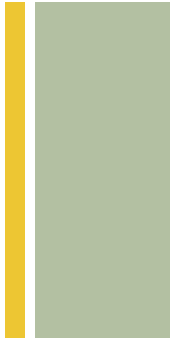


- Find the 2nd prime number

```
range(1000, 10000) filter isPrime at 1
```

- Trick for improvement
 - Avoid computing tail of a sequence unless that tail is actually necessary for computation

+ Stream



- Constant : empty

```
Stream.cons(1, Stream.cons(2, Stream.empty))
```

- Constructor : cons

- Stream.range(start: Int, end: Int)

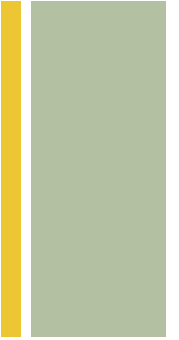
- Immediately returns a Stream object, its first element is “start”
- Other elements computed only when they are *demanded* by tail method

- Access just as lists

- isEmpty; head; tail

```
Stream.range(1000, 10000) filter isPrime at 1
```

+ Stream



- Instead of `x :: xs`

- `Stream.cons(x, xs)`

`x #:: xs == Stream.cons(x, xs)`

- Instead of `xs ::: ys`

- `xs append ys`

Infinite Streams

You saw that all elements of a stream except the first one are computed only when they are needed to produce a result.

This opens up the possibility to define infinite streams!

For instance, here is the stream of all integers starting from a given number:

```
def from(n: Int): Stream[Int] = n #:: from(n+1)
```

The stream of all natural numbers:

```
val nats = from(0)
```

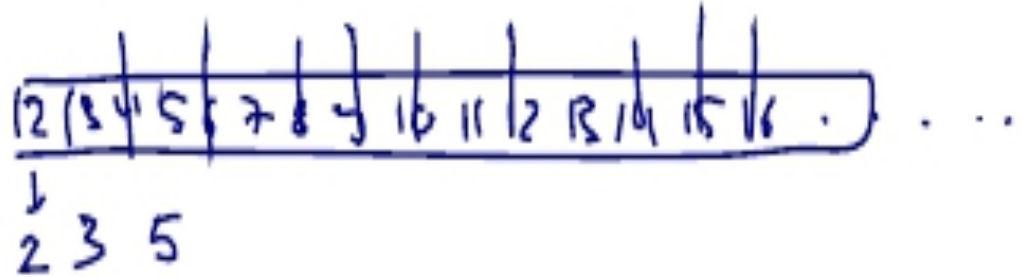
The stream of all multiples of 4:

```
nats map (_ * 4)
```

The Sieve of Eratosthenes 埃拉托斯特尼筛法

The Sieve of Eratosthenes is an ancient technique to calculate prime numbers.

The idea is as follows:



- ▶ Start with all integers from 2, the first prime number.
- ▶ Eliminate all multiples of 2.
- ▶ The first element of the resulting list is 3, a prime number.
- ▶ Eliminate all multiples of 3.
- ▶ Iterate forever. At each step, the first number in the list is a prime number and we eliminate all its multiples.

The Sieve of Eratosthenes in Code

Here's a function that implements this principle:

```
def sieve(s: Stream[Int]): Stream[Int] =  
  s.head #:: sieve(s.tail filter (_ % s.head != 0))
```

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

To see the list of the first N prime numbers, you can write

```
(primes take N).toList
```


Back to Square Roots

Our previous algorithm for square roots always used a `isGoodEnough` test to tell when to terminate the iteration.

With streams we can now express the concept of a converging sequence without having to worry about when to terminate it:

```
def sqrtStream(x: Double): Stream[Double] = {  
  def improve(guess: Double) = (guess + x / guess) / 2  
  lazy val guesses: Stream[Double] = 1 #:: (guesses map improve)  
  guesses  
}
```

Termination

We can add isGoodEnough later.

```
def isGoodEnough(guess: Double, x: Double) =  
  math.abs((guess * guess - x) / x) < 0.0001
```

```
sqrtStream(4) filter (isGoodEnough(_, 4))
```

Streams

Collections and Combinatorial Search

We've seen a number of immutable collections that provide powerful operations, in particular for combinatorial search.

For instance, to find the second prime number between 1000 and 10000:

```
((1000 to 10000) filter isPrime)(1)
```

This is *much* shorter than the recursive alternative:

```
def secondPrime(from: Int, to: Int) = nthPrime(from, to, 2)
def nthPrime(from: Int, to: Int, n: Int): Int =
  if (from >= to) throw new Error("no prime")
  else if (isPrime(from))

    if (n == 1) from else nthPrime(from + 1, to, n - 1)
  else nthPrime(from + 1, to, n)
```

Performance Problem

But from a standpoint of performance,

```
((1000 to 10000) filter isPrime)(1)
```

is pretty bad; it constructs *all* prime numbers between 1000 and 10000 in a list, but only ever looks at the first two elements of that list.

Reducing the upper bound would speed things up, but risks that we miss the second prime number all together.

Delayed Evaluation

However, we can make the short-code efficient by using a trick:

Avoid computing the tail of a sequence until it is needed for the evaluation result (which might be never)

This idea is implemented in a new class, the Stream.

Streams are similar to lists, but their tail is evaluated only *on demand*.

Defining Streams

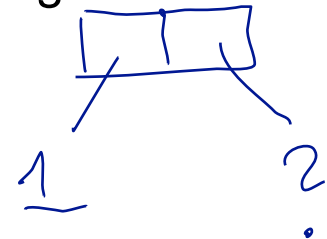
Streams are defined from a constant `Stream.empty` and a constructor `Stream.cons`.

For instance,

```
val xs = Stream.cons(1, Stream.cons(2, Stream.empty))
```

They can also be defined like the other collections by using the object `Stream` as a factory.

```
Stream(1, 2, 3)
```



The `toStream` method on a collection will turn the collection into a stream:

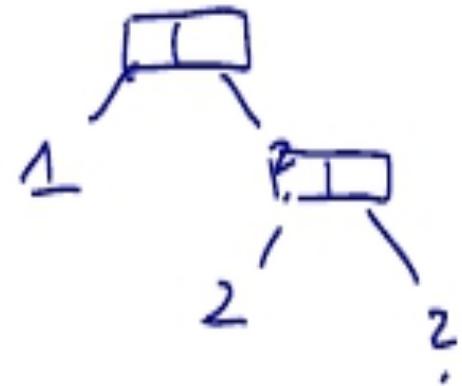
```
(1 to 1000).toStream    > res0: Stream[Int] = Stream(1, ?)
```

Stream Ranges

Let's try to write a function that returns `(lo until hi).toStream` directly:

streamRange

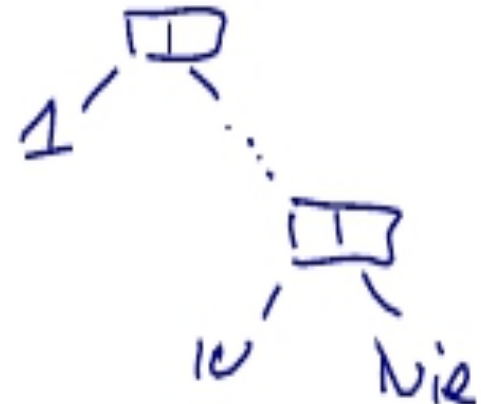
```
def streamRange(lo: Int, hi: Int): Stream[Int] =  
  if (lo >= hi) Stream.empty  
  else Stream.cons(lo, streamRange(lo + 1, hi))
```



Compare to the same function that produces a list:

```
def listRange(lo: Int, hi: Int): List[Int] =  
  if (lo >= hi) Nil  
  else lo :: listRange(lo + 1, hi)
```

listRange(1, 10)



Comparing the Two Range Functions

The functions have almost identical structure yet they evaluate quite differently.

- ▶ `listRange(start, end)` will produce a list with $\text{end} - \text{start}$ elements and return it.
- ▶ `streamRange(start, end)` returns a single object of type `Stream` with `start` as head element.
- ▶ The other elements are only computed when they are needed, where “needed” means that someone calls `tail` on the stream.

Methods on Streams

Stream supports almost all methods of List.

For instance, to find the second prime number between 1000 and 10000:

```
((1000 to 10000).toStream filter isPrime)(1)
```

Stream Cons Operator

The one major exception is `::`.

`x :: xs` always produces a list, never a stream.

There is however an alternative operator `#::` which produces a stream.

$$x \#:: xs == \text{Stream.cons}(x, xs)$$

`#::` can be used in expressions as well as patterns.

Implementation of Streams

The implementation of streams is quite close to the one of lists.

Here's the trait Stream:

```
trait Stream[+A] extends Seq[A] {  
  def isEmpty: Boolean  
  def head: A  
  def tail: Stream[A]  
  ...  
}
```

As for lists, all other methods can be defined in terms of these three.

Implementation of Streams(2)

Concrete implementations of streams are defined in the Stream companion object. Here's a first draft:

```
object Stream {  
  def cons[T](hd: T, tl: => Stream[T]) = new Stream[T] {  
    def isEmpty = false  
    def head = hd  
    def tail = tl  
  }  
  
  val empty = new Stream[Nothing] {  
    def isEmpty = true  
    def head = throw new NoSuchElementException("empty.head")  
    def tail = throw new NoSuchElementException("empty.tail")  
  }  
}
```

Stream.empty ≈ Nil
Stream.cons ≈ ::

Difference to List

The only important difference between the implementations of `List` and `Stream` concern `tl`, the second parameter of `Stream.cons`.

For streams, this is a by-name parameter.

That's why the second argument to `Stream.cons` is not evaluated at the point of call.

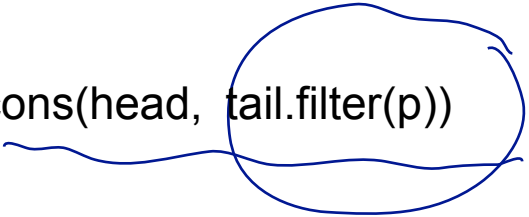
Instead, it will be evaluated each time someone calls `tail` on a `Stream` object.

Other Stream Methods

The other stream methods are implemented analogously to their list counterparts.

For instance, here's filter:

```
class Stream[+T] {  
  ...  
  def filter(p: T => Boolean): Stream[T] =  
    if (isEmpty) this  
    else if (p(head)) cons(head, tail.filter(p))  
    else tail.filter(p)  
}
```



Exercise

Consider this modification of `streamRange`.

```
def streamRange(lo: Int, hi: Int): Stream[Int] = {  
  print(lo+" ")  
  if (lo >= hi) Stream.empty  
  else Stream.cons(lo, streamRange(lo + 1, hi))  
}
```

When you write `streamRange(1, 10).take(3).toList` what gets printed?

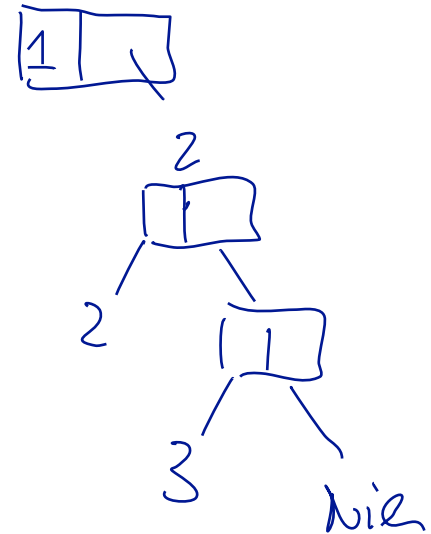
- ☐ Nothing
- ☐ 1
- ☐ 1 2 3
- ☐ 1 2 3 4
- ☐ 1 2 3 4 5 6 7 8 9

Exercise

Consider this modification of `streamRange`.

```
def streamRange(lo: Int, hi: Int): Stream[Int] = {  
  print(lo+" ")  
  if (lo >= hi) Stream.empty  
  else Stream.cons(lo, streamRange(lo + 1, hi))  
}
```

When you write `streamRange(1, 10).take(3).toList` what gets printed?



- ☐ Nothing
- ☐ 1
- ☐ 1 2 3
- ☒ 1 2 3 4
- ☐ 1 2 3 4 5 6 7 8 9

+ Iterators

- The imperative version of streams
- Describe potentially infinite lists

```
trait Iterator[+A] {  
  def hasNext: Boolean  
  def next: A
```

```
  val it: Iterator[Int] = Iterator.range(1, 100)  
  while (it.hasNext) {  
    val x = it.next  
    println(x * x)  
  }
```

+ Iterator Methods

- Many methods mimic corresponding functionality in lists
- Append
- Map, FlatMap, Foreach
- Filter
- Zip

```
def append[B >: A](that: Iterator[B]): Iterator[B] = new Iterator[B] {  
  def map[B](f: A => B): Iterator[B] = new Iterator[B] {  
    def flatMap[B](f: A => Iterator[B]): Iterator[B] = new Iterator[B] {  
      def foreach(f: A => Unit): Unit =  
        while (hasNext) { f(next) }  
    }  
  }  
  def zip[B](that: Iterator[B]) = new Iterator[(A, B)] {
```

+ Constructing iterators

■ Iterator.empty

```
object Iterator {  
  object empty extends Iterator[Nothing] {  
    def hasNext = false  
    def next = error("next on empty iterator")  
  }  
}
```

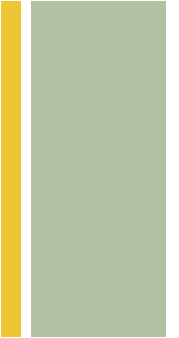
■ fromArray(xs:Array[A])

■ Iterator.range

```
object Iterator {  
  def range(start: Int, end: Int) = new Iterator[Int] {  
    private var current = start  
    def hasNext = current < end  
    def next = {  
      val r = current  
      if (current < end) current += 1  
      else error("end of iterator")  
      r  
    }  
  }  
}
```



Constructing an infinite iterator



- Possible to define iterators that go on forever
- From: returns successive integers from some start value

```
def from(start: Int) = new Iterator[Int] {  
  private var last = start - 1  
  def hasNext = true  
  def next = { last += 1; last }  
}
```

+ Using Iterators

- Print all elements

```
Iterator.fromArray(xs) foreach (x => println(x))
```

```
for (x <- Iterator.fromArray(xs))  
  println(x)
```

- Finding indices of all elements greater than some limit

```
import Iterator._  
fromArray(xs)  
  .zip(from(0))  
  .filter(case (x, i) => x > limit)  
  .map(case (x, i) => i)
```

```
import Iterator._  
for ((x, i) <- fromArray(xs) zip from(0); x > limit)  
yield i
```

fromArray has been
removed from API

object **Iterator**

The `Iterator` object provides various functions for creating specialized iterators.

Source [Iterator.scala](#)

Version 2.8

Since 2.8

► Linear Supertypes



Ordering **Alphabetic** By inheritance

Inherited **Iterator** AnyRef Any

Hide All **Show all** [Learn more about member selection](#)

Visibility **Public** All

Value Members

- `implicit def IteratorCanBuildFrom[A]: BufferedCanBuildFrom[A, Iterator]`
With the advent of `TraversableOnce` and `Iterator`, it can be useful to have a builder which operates on `Iterators` s
- `def apply[A](elems: A*): Iterator[A]`
Creates an iterator with given elements.
- `def continually[A](elem: ⇒ A): Iterator[A]`
Creates an infinite-length iterator returning the results of evaluating an expression.
- `val empty: Iterator[Nothing]`
The iterator which produces no values.
- `def fill[A](len: Int)(elem: ⇒ A): Iterator[A]`
Creates iterator that produces the results of some element computation a number of times.
- `def from(start: Int, step: Int): Iterator[Int]`
Creates an infinite-length iterator returning values equally spaced apart.
- `def from(start: Int): Iterator[Int]`
Creates an infinite-length iterator which returns successive values from some start value.
- `def iterate[T](start: T)(f: (T) ⇒ T): Iterator[T]`
Creates an infinite iterator that repeatedly applies a given function to the previous result.
- `def range(start: Int, end: Int, step: Int): Iterator[Int]`

+ Lazy Values

- Delay initialization of a value, until first time it is accessed.
 - Values that not needed for execution
 - Computational cost is significant

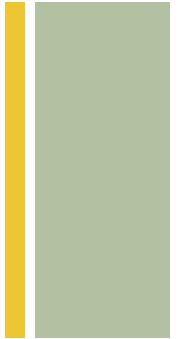
```
case class Employee(id: Int,  
                    name: String,  
                    managerId: Int) {  
  val manager: Employee = Db.get(managerId)  
  val team: List[Employee] = Db.team(id)  
}
```

Delay DB access until it
really needed

```
case class Employee(id: Int,  
                    name: String,  
                    managerId: Int) {  
  lazy val manager: Employee = Db.get(managerId)  
  lazy val team: List[Employee] = Db.team(id)  
}
```




Another use: resolve initialization order involving several modules

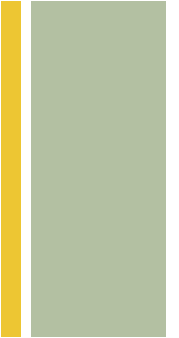


- Consider a compiler composed of several modules.

```
class Symbols(val compiler: Compiler) {  
  import compiler.types._  
  
  val Add = new Symbol("+", FunType(List(IntType, IntType), IntType))  
  val Sub = new Symbol("-", FunType(List(IntType, IntType), IntType))  
  
  class Symbol(name: String, tpe: Type) {  
    override def toString = name + ": " + tpe  
  }  
}
```



```
class Types(val compiler: Compiler) {  
  import compiler.symtab._  
  
  abstract class Type  
  case class FunType(args: List[Type], res: Type) extends Type  
  
  case class NamedType(sym: Symbol) extends Type  
  case object IntType extends Type  
}  
  
class Compiler {  
  val symtab = new Symbols(this)  
  val types  = new Types(this)  
}
```



- Can this work out?

```
val comp = new Compiler
```



Let the compiler figure out the right order

- Make such fields lazy

```
class Compiler {  
  lazy val symtab = new Symbols(this)  
  lazy val types  = new Types(this)  
}
```

Lazy Evaluation

The proposed implementation suffers from a serious potential performance problem: If `tail` is called several times, the corresponding stream will be recomputed each time.

This problem can be avoided by storing the result of the first evaluation of `tail` and re-using the stored result instead of recomputing `tail`.

This optimization is sound, since in a purely functional language an expression produces the same result each time it is evaluated.

We call this scheme *lazy evaluation* (as opposed to *by-name evaluation* in the case where everything is recomputed, and *strict evaluation* for normal parameters and `val` definitions.)

Lazy Evaluation in Scala

Haskell is a functional programming language that uses lazy evaluation by default.

Scala uses strict evaluation by default, but allows lazy evaluation of value definitions with the lazy val form:

lazy val x = expr

def x = expr

Exercise:

Consider the following program:

```
def expr = {  
  val x = { print("x"); 1 }  
  lazy val y = { print("y"); 2 }  
  def z = { print("z"); 3 }  
  z + y + x + z + y + x  
}  
expr
```

xzyz

If you run this program, what gets printed as a side effect of evaluating expr?

- | | |
|--------------------------------------|---------------------------------------|
| <input type="radio"/> zyxzyx | <input checked="" type="radio"/> xzyz |
| <input type="radio"/> xyzz | <input type="radio"/> zyzz |
| <input type="radio"/> something else | |

Implicit Parameters

Making Sort more General

Problem: How to parameterize msort so that it can also be used for lists with elements other than Int?

```
def msort[T](xs: List[T]): List[T] = ...
```

does not work, because the comparison `<` in merge is not defined for arbitrary types `T`.

Idea: Parameterize merge with the necessary comparison function.

Parameterization of Sort

The most flexible design is to make the function `sort` polymorphic and to pass the comparison operation as an additional parameter:

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean) = {  
  ...  
  merge(msort(fst)(lt), msort(snd)(lt))  
}
```

Merge then needs to be adapted as follows:

```
def merge(xs: List[T], ys: List[T]) = (xs, ys) match {  
  ...  
  case (x :: xs1, y :: ys1) =>  
    if (lt(x, y)) ...  
    else ...  
}
```

Calling Parameterized Sort

We can now call `msort` as follows:

```
val xs = List(-5, 6, 3, 2, 7)
```

```
val fruit = List("apple", "pear", "orange", "pineapple")
```

```
merge(xs)((x: Int, y: Int) => x < y)
```

```
merge(fruit)((x: String, y: String) => x.compareTo(y) < 0)
```

Or, since parameter types can be inferred from the call `merge(xs)`:

```
merge(xs)((x, y) => x < y)
```

Parametrization with Ordered

There is already a class in the standard library that represents orderings.

```
scala.math.Ordering[T]
```

provides ways to compare elements of type T. So instead of parameterizing with the `lt` operation directly, we could parameterize with `Ordering` instead:

```
def msort[T](xs: List[T])(ord: Ordering) =
```

```
  def merge(xs: List[T], ys: List[T]) =
```

```
    ... if (ord.lt(x, y)) ...
```

```
    ... merge(msort(fst)(ord), msort(snd)(ord)) ...
```

Ordered Instances:

Calling the new msort can be done like this:

```
import math.Ordering
```

```
msort(nums)(Ordering.Int)
```

```
msort(fruits)(Ordering.String)
```

This makes use of the values `Int` and `String` defined in the `scala.math.Ordering` object, which produce the right orderings on integers and strings.

Aside: Implicit Parameters

Problem: Passing around `lt` or `ord` values is cumbersome.
We can avoid this by making `ord` an implicit parameter.

```
def msort[T](xs: List[T])(implicit ord: Ordering) =
```

```
  def merge(xs: List[T], ys: List[T]) =
```

```
    ... if (ord.lt(x, y)) ...
```

```
    ... merge(msort(fst), msort(snd)) ...
```

Then calls to `msort` can avoid the ordering parameters:

```
msort(nums)
```

```
msort(fruits)
```

The compiler will figure out the right implicit to pass based on the demanded type.

Rules for Implicit Parameters

Say, a function takes an implicit parameter of type T.

The compiler will search an implicit definition that

- ▶ is marked implicit
- ▶ has a type compatible with T
- ▶ is visible at the point of the function call, or is defined in a companion object associated with T.

If there is a single (most specific) definition, it will be taken as actual argument for the implicit parameter.

Otherwise it's an error.

Exercise: Implicit Parameters

Consider the following line of the definition of `msort`:

```
... merge(msort(fst), msort(snd)) ...
```

Which implicit argument is inserted?

- ☐ `Ordering.Int`
- ☐ `Ordering.String`
- ☐ the `"ord"` parameter of `"msort"`