Structural Induction on Trees
Structural Induction on Trees

Structural induction is not limited to lists; it applies to any tree structure.

The general induction principle is the following:

To prove a property $P(t)$ for all trees $t$ of a certain type,

- show that $P(1)$ holds for all leaves $1$ of a tree,
- for each type of internal node $t$ with subtrees $s_1, ..., s_n$, show that

$$P(s_1) \land ... \land P(s_n) \text{ implies } P(t).$$
Example: IntSets

Recall our definition of IntSet with the operations contains and incl:

```scala
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}

object Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
}
```
Example: IntSets (2)

```scala
case class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {

  def contains(x: Int): Boolean = 
    if (x < elem) left contains x 
    else if (x > elem) right contains x 
    else true

  def incl(x: Int): IntSet = 
    if (x < elem) NonEmpty(elem, left incl x, right) 
    else if (x > elem) NonEmpty(elem, left, right incl x) 
    else this
}
```
The Laws of IntSet

What does it mean to prove the correctness of this implementation?

One way to define and show the correctness of an implementation consists of proving the laws that it respects.

In the case of IntSet, we have the following three laws:

For any set $s$, and elements $x$ and $y$:

- Empty contains $x$ = false
- $(s \text{ incl } x)$ contains $x$ = true
- $(s \text{ incl } x)$ contains $y$ = s contains $y$ if $x \neq y$

(In fact, we can show that these laws completely characterize the desired data type).
Proving the Laws of IntSet (1)

How can we prove these laws?

*Proposition 1*: Empty contains $x = \text{false}$.

*Proof*: According to the definition of contains in Empty.
Proving the Laws of IntSet (2)

**Proposition 2:** \((s \text{ incl } x) \text{ contains } x = \text{ true}\)

Proof by structural induction on \(s\).

**Base case:** Empty

\((\text{Empty incl } x) \text{ contains } x\)
Proposition 2: \((s \text{ incl } x) \text{ contains } x = \text{ true}\)

Proof by structural induction on \(s\).

**Base case:** Empty

\((\text{Empty incl } x) \text{ contains } x\)

\[ = \text{NonEmpty}(x, \text{Empty, Empty}) \text{ contains } x \quad \text{// by definition of Empty.incl} \]
Proving the Laws of IntSet (2)

**Proposition 2:** \((s \text{ incl } x) \text{ contains } x = \text{ true}\)

Proof by structural induction on \(s\).

**Base case:** Empty

\((\text{Empty incl } x) \text{ contains } x\)

\[= \text{ NonEmpty}(x, \text{Empty}, \text{Empty}) \text{ contains } x \quad \text{// by definition of Empty.incl}\]

\[= \text{ true} \quad \text{// by definition of NonEmpty.contains}\]
Proving the Laws of IntSet (3)

**Induction step:** $\text{NonEmpty}(x, l, r)$

$(\text{NonEmpty}(x, l, r) \text{ incl } x)$ contains $x$

$\text{NonEmpty}(z, l, r)$

$Z = x$

$Z \neq x$
Proving the Laws of IntSet (3)

**Induction step:** NonEmpty(x, l, r)

\[
\text{(NonEmpty}(x, l, r) \text{ incl } x) \text{ contains } x
\]

\[
= \text{NonEmpty}(x, l, r) \text{ contains } x \quad \text{// by definition of NonEmpty.incl}
\]
Induction step: NonEmpty(x, l, r)

(NonEmpty(x, l, r) incl x) contains x

= NonEmpty(x, l, r) contains x  // by definition of NonEmpty.incl

= true  // by definition of NonEmpty.contains
Induction step: NonEmpty(y, l, r) where y < x

(NonEmpty(y, l, r) incl x) contains x
Proving the Laws of IntSet (4)

**Induction step:** NonEmpty(y, l, r) where \( y < x \)

\[(\text{NonEmpty}(y, l, r) \text{ incl } x) \text{ contains } x = \text{NonEmpty}(y, l, r \text{ incl } x) \text{ contains } x \quad \text{// by definition of NonEmpty.incl}\]
Proving the Laws of IntSet (4)

**Induction step:** NonEmpty(y, l, r) where y < x

\[
(\text{NonEmpty}(y, l, r) \text{ incl } x) \text{ contains } x
\]

\[
= \text{ NonEmpty}(y, l, r \text{ incl } x) \text{ contains } x \quad \text{// by definition of NonEmpty.incl}
\]

\[
= (r \text{ incl } x) \text{ contains } x \quad \text{// by definition of NonEmpty.contains}
\]
Proving the Laws of IntSet (4)

**Induction step:** \( NonEmpty(y, l, r) \) where \( y < x \)

\[
(NonEmpty(y, l, r) \text{ incl } x) \text{ contains } x
\]

\[
= \text{ NonEmpty}(y, l, r \text{ incl } x) \text{ contains } x \quad \text{ // by definition of NonEmpty.incl}
\]

\[
= (r \text{ incl } x) \text{ contains } x \quad \text{ // by definition of NonEmpty.contains}
\]

\[
= \text{ true} \quad \text{ // by the induction hypothesis}
\]
Proof of the Laws of IntSet (4)

**Induction step:** $\text{NonEmpty}(y, l, r) \textbf{ where } y < x$

$(\text{NonEmpty}(y, l, r) \text{ incl } x)$ contains $x$

$= \text{NonEmpty}(y, l, r \text{ incl } x)$ contains $x$ // by definition of NonEmpty.incl

$= (r \text{ incl } x)$ contains $x$ // by definition of NonEmpty.contains

$= \text{true}$ // by the induction hypothesis

**Induction step:** $\text{NonEmpty}(y, l, r) \textbf{ where } y > x$ is analogous
Proposition 3: If \( x \neq y \) then

\[(xs \text{ incl } y) \text{ contains } x = xs \text{ contains } x.\]

Proof by structural induction on \( s \). Assume that \( y < x \) (the dual case \( x < y \) is analogous).

**Base case: Empty**

\[(\text{Empty incl } y) \text{ contains } x \quad \text{// to show: } = \quad \text{Empty contains } x\]
Proving the Laws of IntSet (5)

Proposition 3: If \(x \neq y\) then

\[(xs \text{ incl } y) \text{ contains } x = xs \text{ contains } x.\]

Proof by structural induction on \(s\). Assume that \(y < x\) (the dual case \(x < y\) is analogous).

**Base case:** \(\text{Empty}\)

\[(\text{Empty incl } y) \text{ contains } x \quad \text{// to show: } = \quad \text{Empty contains } x\]

\[= \quad \text{NonEmpty}(y, \text{Empty, Empty}) \text{ contains } x \quad \text{// by definition of Empty.incl}\]
Proving the Laws of IntSet (5)

*Proposition 3*: If \( x \neq y \) then

\[(xs \text{ incl } y) \text{ contains } x \quad = \quad xs \text{ contains } x.\]

Proof by structural induction on \( s \). Assume that \( y < x \) (the dual case \( x < y \) is analogous).

**Base case: Empty**

\[(Empty \text{ incl } y) \text{ contains } x \quad // \text{ to show: } = \quad Empty \text{ contains } x\]

\[= \quad \text{NonEmpty}(y, Empty, Empty) \text{ contains } x \quad // \text{ by definition of Empty.incl}\]

\[= \quad Empty \text{ contains } x \quad // \text{ by definition of NonEmpty.contains}\]
Proving the Laws of IntSet (6)

For the inductive step, we need to consider a tree \( \text{NonEmpty}(z, 1, r) \). We distinguish five cases:

1. \( z = x \)
2. \( z = y \)
3. \( z < y < x \)
4. \( y < z < x \)
5. \( y < x < z \)
First Two Cases: $z = x, z = y$

**Induction step:** NonEmpty($x, l, r$)

(NonEmpty($x, l, r$) incl $y$) contains $x$  // to show: = NonEmpty($x, l, r$) contains $x$
First Two Cases: \( z = x, \ z = y \)

**Induction step:** \( \text{NonEmpty}(x, l, r) \)

\[
(\text{NonEmpty}(x, l, r) \text{ incl } y) \text{ contains } x \quad \text{// to show: } = \quad \text{NonEmpty}(x, l, r) \text{ contains } x
\]

\[
= \quad \text{NonEmpty}(x, l \text{ incl } y, r) \text{ contains } x \quad \text{// by definition of NonEmpty.incl}
\]
First Two Cases: $z = x$, $z = y$

**Induction step:** $\text{NonEmpty}(x, l, r)$

$(\text{NonEmpty}(x, l, r) \text{ incl } y) \text{ contains } x$  // to show: $= \text{ NonEmpty}(x, l, r) \text{ contains } x$

$= \text{ NonEmpty}(x, l \text{ incl } y, r) \text{ contains } x$  // by definition of $\text{NonEmpty}.\text{incl}$

$= \text{ true}$  // by definition of $\text{NonEmpty}.\text{contains}$
First Two Cases: $z = x, z = y$

**Induction step:** \( \text{NonEmpty}(x, l, r) \)

\[
(\text{NonEmpty}(x, l, r) \text{ incl } y) \text{ contains } x \quad \text{// to show: } \quad \text{NonEmpty}(x, l, r) \text{ contains } x
\]

\[
= \text{NonEmpty}(x, l \text{ incl } y, r) \text{ contains } x \quad \text{// by definition of NonEmpty.incl}
\]

\[
= \text{true} \quad \text{// by definition of NonEmpty.contains}
\]

\[
= \text{NonEmpty}(x, l, r) \text{ contains } x \quad \text{// by definition of NonEmpty.contains}
\]
First Two Cases: \( z = x, \ z = y \)

**Induction step:** \( \text{NonEmpty}(x, l, r) \)

\[(\text{NonEmpty}(x, l, r) \text{ incl } y) \text{ contains } x \quad // \text{to show: } = \quad \text{NonEmpty}(x, l, r) \text{ contains } x\]

\[= \quad \text{NonEmpty}(x, l \text{ incl } y, r) \text{ contains } x \quad // \text{by definition of NonEmpty.incl}\]

\[= \quad \text{true} \quad // \text{by definition of NonEmpty.contains}\]

\[= \quad \text{NonEmpty}(x, l, r) \text{ contains } x \quad // \text{by definition of NonEmpty.contains}\]

**Induction step:** \( \text{NonEmpty}(y, l, r) \)

\[(\text{NonEmpty}(y, l, r) \text{ incl } y) \text{ contains } x \quad // \text{to show: } = \quad \text{NonEmpty}(y, l, r) \text{ contains } x\]
First Two Cases: $z = x$, $z = y$

**Induction step:** $\text{NonEmpty}(x, l, r)$

$(\text{NonEmpty}(x, l, r) \text{ incl } y) \text{ contains } x$  // to show: $= \text{NonEmpty}(x, l, r) \text{ contains } x$

$= \text{NonEmpty}(x, l \text{ incl } y, r) \text{ contains } x$  // by definition of $\text{NonEmpty.incl}$

$= \text{true}$  // by definition of $\text{NonEmpty.contains}$

$= \text{NonEmpty}(x, l, r) \text{ contains } x$  // by definition of $\text{NonEmpty.contains}$

**Induction step:** $\text{NonEmpty}(y, l, r)$

$(\text{NonEmpty}(y, l, r) \text{ incl } y) \text{ contains } x$  // to show: $= \text{NonEmpty}(y, l, r) \text{ contains } x$

$= \text{NonEmpty}(y, l, r) \text{ contains } x$  // by definition of $\text{NonEmpty.incl}$
Case $z < y$

**Induction step:** $\text{NonEmpty}(z, l, r)$ where $z < y < x$

$(\text{NonEmpty}(z, l, r) \text{ incl } y)$ contains $x$  // to show: $= \text{NonEmpty}(z, l, r)$ contains $x$
Case $z < y$

**Induction step:** $\text{NonEmpty}(z, l, r)$ where $z < y < x$

$(\text{NonEmpty}(z, l, r) \text{ incl } y)$ contains $x$  // to show: = $\text{NonEmpty}(z, l, r)$ contains $x$

= $\text{NonEmpty}(z, l, r \text{ incl } y)$ contains $x$  // by definition of $\text{NonEmpty.incl}$
Case $z < y$

**Induction step:** \( \text{NonEmpty}(z, l, r) \) where $z < y < x$

\[(\text{NonEmpty}(z, l, r) \text{ incl } y) \text{ contains } x \quad \text{// to show: } = \quad \text{NonEmpty}(z, l, r) \text{ contains } x\]

\[= \quad \text{NonEmpty}(z, l, r \text{ incl } y) \text{ contains } x \quad \text{// by definition of NonEmpty.incl}\]

\[= \quad (r \text{ incl } y) \text{ contains } x \quad \text{// by definition of NonEmpty.contains}\]
Case $z < y$

**Induction step:** NonEmpty($z, l, r$) where $z < y < x$

(NonEmpty($z, l, r$) incl $y$) contains $x$  // to show: = NonEmpty($z, l, r$) contains $x$

= NonEmpty($z, l, r$ incl $y$) contains $x$     // by definition of NonEmpty.incl

= (r incl $y$) contains $x$                      // by definition of NonEmpty.contains

= $r$ contains $x$                              // by the induction hypothesis
Case $z < y$

**Induction step:** $\text{NonEmpty}(z, l, r)$ \textbf{where} $z < y < x$

$(\text{NonEmpty}(z, l, r) \text{ incl } y) \text{ contains } x$ \hspace{1em} // to show: $= \text{NonEmpty}(z, l, r) \text{ contains } x$

$= \text{NonEmpty}(z, l, r \text{ incl } y) \text{ contains } x$ \hspace{1em} // by definition of $\text{NonEmpty.incl}$

$= (r \text{ incl } y) \text{ contains } x$ \hspace{1em} // by definition of $\text{NonEmpty.contains}$

$= r \text{ contains } x$ \hspace{1em} // by the induction hypothesis

$= \text{NonEmpty}(z, l, r) \text{ contains } x$ \hspace{1em} // by definition of $\text{NonEmpty.contains}$
Case $y < z < x$

**Induction step:** NonEmpty$(z, l, r)$ where $y < z < x$

(NonEmpty$(z, l, r)$ incl $y$) contains $x$  // to show: $= \text{ NonEmpty}(z, l, r)$ contains $x$
Case $y < z < x$

**Induction step:** $\text{NonEmpty}(z, l, r)$ where $y < z < x$

$(\text{NonEmpty}(z, l, r) \text{ incl } y)$ contains $x$  // to show: $= \text{NonEmpty}(z, l, r)$ contains $x$

$= \text{NonEmpty}(z, l \text{ incl } y, r)$ contains $x$  // by definition of $\text{NonEmpty.incl}$
Case $y < z < x$

**Induction step:** NonEmpty$(z, l, r)$ where $y < z < x$

(NonEmpty$(z, l, r)$ incl $y$) contains $x$  // to show: = NonEmpty$(z, l, r)$ contains $x$

= NonEmpty$(z, l$ incl $y, r)$ contains $x$  // by definition of NonEmpty.incl

= $r$ contains $x$  // by definition of NonEmpty.contains
Case $y < z < x$

**Induction step:** $\text{NonEmpty}(z, l, r)$ where $y < z < x$

$(\text{NonEmpty}(z, l, r) \text{ incl } y)$ contains $x$ // to show: $= \text{NonEmpty}(z, l, r)$ contains $x$

$= \text{NonEmpty}(z, l \text{ incl } y, r)$ contains $x$ // by definition of $\text{NonEmpty.incl}$

$= r$ contains $x$ // by definition of $\text{NonEmpty.contains}$

$= \text{NonEmpty}(z, l, r)$ contains $x$ // by definition of $\text{NonEmpty.contains}$
Case $x < z$

**Induction step:** $\text{NonEmpty}(z, l, r)$ where $y < x < z$

$(\text{NonEmpty}(z, l, r) \text{ incl } y)$ contains $x$  // to show: $= \text{ NonEmpty}(z, l, r)$ contains $x$
Case $x < z$

**Induction step:** $\text{NonEmpty}(z, l, r)$ where $y < x < z$

$(\text{NonEmpty}(z, l, r) \text{ incl } y)$ contains $x$  // to show: $= \text{NonEmpty}(z, l, r)$ contains $x$

$= \text{NonEmpty}(z, l \text{ incl } y, r)$ contains $x$  // by definition of NonEmpty.incl
Case $x < z$

\begin{itemize}
\item \textbf{Induction step:} \texttt{NonEmpty(z, l, r) where $y < x < z$}
\item $(\texttt{NonEmpty(z, l, r) incl y}) \text{ contains } x$  \hfill // to show: $= \texttt{NonEmpty(z, l, r) contains } x$
\item $= \texttt{NonEmpty(z, l incl y, r) contains } x$  \hfill // by definition of \texttt{NonEmpty.incl}
\item $= \texttt{(l incl y) contains } x$  \hfill // by definition of \texttt{NonEmpty.contains}
\end{itemize}
Case $x < z$

<table>
<thead>
<tr>
<th>Induction step: NonEmpty(z, l, r) where $y &lt; x &lt; z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NonEmpty(z, l, r) incl y) contains x // to show: = NonEmpty(z, l, r) contains x</td>
</tr>
<tr>
<td>= NonEmpty(z, l incl y, r) contains x // by definition of NonEmpty.incl</td>
</tr>
<tr>
<td>= (l incl y) contains x // by definition of NonEmpty.contains</td>
</tr>
<tr>
<td>= l contains x // by the induction hypothesis</td>
</tr>
</tbody>
</table>
Case $x < z$

**Induction step:** $\text{NonEmpty}(z, l, r)$ where $y < x < z$

$(\text{NonEmpty}(z, l, r) \text{ incl } y) \text{ contains } x$  // to show: $= \text{NonEmpty}(z, l, r) \text{ contains } x$

$= \text{NonEmpty}(z, l \text{ incl } y, r) \text{ contains } x$  // by definition of NonEmpty.incl

$= (l \text{ incl } y) \text{ contains } x$  // by definition of NonEmpty.contains

$= l \text{ contains } x$  // by the induction hypothesis

$= \text{NonEmpty}(z, l, r) \text{ contains } x$  // by definition of NonEmpty.contains

These are all the cases, so the proposition is established.
Exercise (Hard)

Suppose we add a function `union` to `IntSet`:

```scala
abstract class IntSet { 
  def union(other: IntSet): IntSet
}
object Empty extends IntSet { 
  def union(other: IntSet) = other
}
class NonEmpty(x: Int, l: IntSet, r: IntSet) extends IntSet { 
  def union(other: IntSet): IntSet = (l union (r union (other))) incl x
}
```
Exercise (Hard)

The correctness of union can be translated into the following law:

*Proposition 4:*

\[(xs \cup ys) \text{ contains } x = xs \text{ contains } x \lor ys \text{ contains } x\]

Show proposition 4 by using structural induction on \(xs\).
Streams
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:

```python
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,

\[
((1000 \text{ to } 10000) \ \text{filter isPrime})(1)
\]

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,

```scala
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 \((\text{lo until hi}).\text{toStream}\) directly:

```python
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:

```python
def listRange(lo: Int, hi: Int): List[Int] = 
    if (lo >= hi) Nil 
    else lo :: listRange(lo + 1, hi)
```
Comparing the Two Range Functions

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

- listRange(start, end) will produce a list with end - 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:

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

Here's the trait `Stream`:

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

```scala
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")
  }
}
```
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:

```scala
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)
}
```
Consider this modification of \texttt{streamRange}.

\begin{verbatim}
    print(lo + " ")
    if (lo >= hi) Stream.empty
    else Stream.cons(lo, streamRange(lo + 1, hi))
}
\end{verbatim}

When you write \texttt{streamRange(1, 10).take(3).toList}
what gets printed?

0 Nothing
0 1
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5 6 7 8 9
Exercise

Consider this modification of `streamRange`.

```python
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?

```
0   Nothing
0   1
1   1 2 3
0   1 2 3 4
0   1 2 3 4 5 6 7 8 9
```
Lazy Evaluation
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:

```scala
lazy val x = expr
```
Exercise:

Consider the following program:

```python
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
```

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

<table>
<thead>
<tr>
<th></th>
<th>zyxzyx</th>
<th></th>
<th>xzyz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>something else</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lazy Vals and Streams

Using a lazy value for tail, Stream.cons can be implemented more efficiently:

def cons[T](hd: T, tl: => Stream[T]) = new Stream[T] {
    def head = hd
    lazy val tail = tl
    ...
}
Seeing it in Action

To convince ourselves that the implementation of streams really does avoid unnecessary computation, let’s observe the execution trace of the expression:

\[
\text{(streamRange(1000, 10000) filter isPrime) apply 1}
\]
To convince ourselves that the implementation of streams really does avoid unnecessary computation, let’s observe the execution trace of the expression:

\[
\text{(streamRange}(1000, 10000) \text{ filter isPrime}) \text{ apply 1}
\]

\[
\rightarrow \text{(if } 1000 \geq 10000 \text{ empty) empty} \quad \text{// by expanding streamRange}
\]

\[
\text{else cons}(1000, \text{ streamRange}(1000 + 1, 10000))
\]

\[
.\text{filter(isPrime).apply(1)}
\]
Seeing it in Action

To convince ourselves that the implementation of streams really does avoid unnecessary computation, let’s observe the execution trace of the expression:

```
(streamRange(1000, 10000) filter isPrime) apply 1
```

-->  \( \text{if} \ (1000 \geq 10000) \ \text{empty} \)  \hspace{1cm} // \text{by expanding streamRange}
    \hspace{1cm} \text{else} \ \text{cons}(1000, \text{streamRange}(1000 + 1, 10000))
    \hspace{1cm} .\text{filter}(\text{isPrime}).\text{apply}(1)

--> \text{cons}(1000, \text{streamRange}(1000 + 1, 10000)) \hspace{1cm} // \text{by evaluating if}
    \hspace{1cm} .\text{filter}(\text{isPrime}).\text{apply}(1)
Let's abbreviate \( \text{cons}(1000, \text{streamRange}(1000 + 1, 10000)) \) to \( C_1 \).

\[ C_1.\text{filter}(\text{isPrime}).\text{apply}(1) \]
Evaluation Trace (2)

Let's abbreviate \(\text{cons}(1000, \text{streamRange}(1000 + 1, 10000))\) to \(C1\).

\[
C1.\text{filter(\text{isPrime})}.\text{apply}(1)
\]

\[
\rightarrow \begin{cases}
\text{if } (C1.\text{isEmpty}) & C1 \\
\text{else if } (\text{isPrime}(C1.\text{head})) & \text{cons}(C1.\text{head}, C1.\text{tail}.\text{filter(\text{isPrime}))} \\
\text{else} & C1.\text{tail}.\text{filter(\text{isPrime}))}
\end{cases}
\] // by expanding filter

apply(1)
Let's abbreviate \( \text{cons}(1000, \text{streamRange}(1000 + 1, 10000)) \) to \( C1 \).

\[
C1.\text{filter}(\text{isPrime}).\text{apply}(1)
\]

\[
\rightarrow (\text{if} \ (C1.\text{isEmpty}) \ C1 \quad \quad \quad \quad \quad \text{by expanding filter} \\
\text{else if} \ (\text{isPrime}(C1.\text{head})) \ \text{cons}(C1.\text{head}, \ C1.\text{tail}.\text{filter}(\text{isPrime})) \\
\text{else} \ C1.\text{tail}.\text{filter}(\text{isPrime}))
\]

\[
.\text{apply}(1)
\]

\[
\rightarrow (\text{if} \ (\text{isPrime}(C1.\text{head})) \ \text{cons}(C1.\text{head}, \ C1.\text{tail}.\text{filter}(\text{isPrime}))
\]

\[
\text{else} \ C1.\text{tail}.\text{filter}(\text{isPrime})) \quad \quad \quad \text{by eval. if}
\]

\[
.\text{apply}(1)
\]
Let's abbreviate \( \text{cons}(1000, \text{streamRange}(1000 + 1, 10000)) \) to \( C1 \).

\[
C1.\text{filter}(\text{isPrime}).\text{apply}(1)
\]

\[
\rightarrow (\text{if } (C1.\text{isEmpty}) C1 \quad \text{// by expanding filter} \\
\text{else if } (\text{isPrime}(C1.\text{head})) \text{ cons}(C1.\text{head}, C1.\text{tail.} \text{filter}(\text{isPrime})) \\
\text{else } C1.\text{tail.} \text{filter}(\text{isPrime}))
\]

\[
.\text{apply}(1)
\]

\[
\rightarrow (\text{if } (\text{isPrime}(C1.\text{head})) \text{ cons}(C1.\text{head}, C1.\text{tail.} \text{filter}(\text{isPrime})) \\
\text{else } C1.\text{tail.} \text{filter}(\text{isPrime})) \quad \text{// by eval. if} \\
.\text{apply}(1)
\]

\[
\rightarrow (\text{if } (\text{isPrime}(1000)) \text{ cons}(C1.\text{head}, C1.\text{tail.} \text{filter}(\text{isPrime})) \\
\text{else } C1.\text{tail.} \text{filter}(\text{isPrime})) \quad \text{// by eval. head} \\
.\text{apply}(1)
\]
Evaluation Trace (3)

```java
-->> (if (false) cons(C1.head, C1.tail.filter(isPrime))  // by eval. isPrime
    else C1.tail.filter(isPrime))
    .apply(1)
```
Evaluation Trace (3)

```java
-->> (if (false) cons(C1.head, C1.tail.filter(isPrime)) // by eval. isPrime
   else C1.tail.filter(isPrime))
   .apply(1)

--> C1.tail.filter(isPrime).apply(1) // by eval. if
```
Evaluation Trace (3)

-->> (if (false) cons(C1.head, C1.tail.filter(isPrime))) // by eval. isPrime
else C1.tail.filter(isPrime))
.apply(1)

-->> C1.tail.filter(isPrime).apply(1) // by eval. if

-->> streamRange(1001, 10000) // by eval. tail
.filter(isPrime).apply(1)

The evaluation sequence continues like this until:
Evaluation Trace (3)

```plaintext
-->> (if (false) cons(C1.head, C1.tail.filter(isPrime)) // by eval. isPrime
    else C1.tail.filter(isPrime))
  .apply(1)

--> C1.tail.filter(isPrime).apply(1)                    // by eval. if

-->> streamRange(1001, 10000)                           // by eval. tail
    .filter(isPrime).apply(1)

The evaluation sequence continues like this until:

-->> streamRange(1009, 10000)
    .filter(isPrime).apply(1)

--> cons(1009, streamRange(1009 + 1, 10000))            // by eval. streamRange
    .filter(isPrime).apply(1)
```
Let's abbreviate \( \text{cons}(1009, \text{streamRange}(1009 + 1, 10000)) \) to \( C2 \).

\[ C2.\text{filter}(\text{isPrime}).\text{apply}(1) \]
Let's abbreviate \( \text{cons}(1009, \text{streamRange}(1009 + 1, 10000)) \) to \( C_2 \).

\[
C_2.\text{filter}(\text{isPrime}).\text{apply}(1)
\]

\[\rightarrow \text{cons}(1009, C_2.\text{tail}.\text{filter}(\text{isPrime})).\text{apply}(1)\]
Let's abbreviate \( \text{cons}(1009, \text{streamRange}(1009 + 1, 10000)) \) to \( C_2 \).

\[
C_2.\text{filter}(\text{isPrime}).\text{apply}(1)
\]

\[
\rightarrow \text{cons}(1009, C_2.\text{tail}.\text{filter}(\text{isPrime})).\text{apply}(1)
\]

\[
\rightarrow \text{if } (1 == 0) \text{ cons}(1009, C_2.\text{tail}.\text{filter}(\text{isPrime})).\text{head} \quad \text{// by eval. apply}
\]

\[
\text{else cons}(1009, C_2.\text{tail}.\text{filter}(\text{isPrime})).\text{tail}.\text{apply}(0)
\]

Assuming \text{apply} is defined like this in Stream\([T]\):

\[
\text{def apply}(n: \text{Int}): T =
\]

\[
\text{if } (n == 0) \text{ head}
\]

\[
\text{else tail.apply}(n-1)
\]
Let's abbreviate cons(1009, streamRange(1009 + 1, 10000)) to C2.

\[ C2\.filter(is\text{Prime}).apply(1) \]

\[ \text{---\text{>> cons}(1009, C2\.tail\.filter(is\text{Prime})).apply(1)} \]  \hspace{1em} // by eval. filter

\[ \text{---\text{>> if } (1 == 0) \text{ cons}(1009, C2\.tail\.filter(is\text{Prime})).head} \]  \hspace{1em} // by eval. apply
\[ \text{else cons}(1009, C2\.tail\.filter(is\text{Prime})).tail\.apply(0) \]

\[ \text{---\text{>> cons}(1009, C2\.tail\.filter(is\text{Prime})).tail\.apply(0)} \]  \hspace{1em} // by eval. if
Let's abbreviate \( \text{cons}(1009, \text{streamRange}(1009 + 1, 10000)) \) to \( C2 \).

\[
\begin{align*}
C2.\text{filter}(\text{isPrime}).\text{apply}(1) \\
\text{-->> } \text{cons}(1009, C2.\text{tail.\text{filter}(\text{isPrime)})).\text{apply}(1) & \quad \text{ // by eval. filter} \\
\text{--> if (1 == 0) cons(1009, C2.\text{tail.\text{filter}(\text{isPrime)})).\text{head} & \quad \text{ // by eval. apply} \\
& \quad \text{else cons(1009, C2.\text{tail.\text{filter}(\text{isPrime)})).\text{tail.\text{apply}(0)}} \\
\text{--> cons(1009, C2.\text{tail.\text{filter}(\text{isPrime)})).\text{tail.\text{apply}(0)} & \quad \text{ // by eval. if} \\
\text{--> C2.\text{tail.\text{filter}(\text{isPrime)}).\text{apply}(0)} & \quad \text{ // by eval. tail}
\end{align*}
\]
Evaluation Trace (4)

Let's abbreviate `cons(1009, streamRange(1009 + 1, 10000))` to C2.

```
C2.filter(isPrime).apply(1)
```

---> `cons(1009, C2.tail.filter(isPrime)).apply(1)`  // by eval. filter

--> `if (1 == 0) cons(1009, C2.tail.filter(isPrime)).head`  // by eval. apply
      `else cons(1009, C2.tail.filter(isPrime)).tail.apply(0)`

--> `cons(1009, C2.tail.filter(isPrime)).tail.apply(0)`  // by eval. if

--> `C2.tail.filter(isPrime).apply(0)`  // by eval. tail

--> `streamRange(1010, 10000).filter(isPrime).apply(0)`  // by eval. tail
The process continues until

---

\[
\text{streamRange(1013, 10000).filter(isPrime).apply(0)}
\]
The process continues until

...  

--> streamRange(1013, 10000).filter(isPrime).apply(0)

--> cons(1013, streamRange(1013 + 1, 10000))  // by eval. streamRange
.filter(isPrime).apply(0)

Let $C_3$ be a shorthand for $\text{cons}(1013, \text{streamRange}(1013 + 1, 10000))$.

$== C_3.\text{filter}(\text{isPrime}).\text{apply}(0)$
Evaluation Trace (5)

The process continues until

... 

--> streamRange(1013, 10000).filter(isPrime).apply(0)

--> cons(1013, streamRange(1013 + 1, 10000))  // by eval. streamRange
   .filter(isPrime).apply(0)

Let C3 be a shorthand for cons(1013, streamRange(1013 + 1, 10000)).

== C3.filter(isPrime).apply(0)

---> cons(1013, C3.tail.filter(isPrime)).apply(0)  // by eval. filter
Evaluation Trace (5)

The process continues until

...\

--> streamRange(1013, 10000).filter(isPrime).apply(0)

--> cons(1013, streamRange(1013 + 1, 10000)) // by eval. streamRange
  .filter(isPrime).apply(0)

Let C3 be a shorthand for cons(1013, streamRange(1013 + 1, 10000)).

== C3.filter(isPrime).apply(0)

-->> cons(1013, C3.tail.filter(isPrime)).apply(0) // by eval. filter

---> 1013 // by eval. apply

Only the part of the stream necessary to compute the result has been constructed.
Computing with Infinite Sequences
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:
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:

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

The stream of all natural numbers:

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

The stream of all multiples of 4:
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:

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

The stream of all natural numbers:

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

The stream of all multiples of 4:

```scala
nats map (_ * 4)
```
The Sieve of Eratosthenes

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:

```scala
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

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

```scala
def sqrtStream(x: Double): Stream[Double] = {
  def improve(guess: Double) = (guess + x / guess) / 2
  lazy val guesses: Stream[Double] = 1 #:: (guesses map improve)
  guesses
}
```
We can add isGoodEnough later.

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

sqrtStream(4) filter (isGoodEnough(_, 4))
Exercise:

Consider two ways to express the infinite stream of multiples of a given number N:

```scala
val xs = from(1) map (_ * N)
val ys = from(1) filter (_ % N == 0)
```

Which of the two streams generates its results faster?

<table>
<thead>
<tr>
<th></th>
<th>from(1) map (_ * N)</th>
<th>from(1) filter (_ % N == 0)</th>
</tr>
</thead>
</table>
Exercise:

Consider two ways to express the infinite stream of multiples of a given number \( N \):

\[
\begin{align*}
\text{val } xs &= \text{from}(1) \ \text{map} \ (\_ \ * \ N) \\
\text{val } ys &= \text{from}(1) \ \text{filter} \ (\_ \ % \ N \ == \ 0)
\end{align*}
\]

Which of the two streams generates its results faster?

\[
\begin{align*}
0 & \quad \text{from}(1) \ \text{map} \ (\_ \ * \ N) \\
0 & \quad \text{from}(1) \ \text{filter} \ (\_ \ % \ N \ == \ 0)
\end{align*}
\]
The Water Pouring Problem
States and Moves

Glass: Int
State: Vector[Int] (one entry per glass)
Moves:
Empty(glass)
Fill(glass)
Pour(from, to)
Variants

In a program of the complexity of the pouring program, there are many choices to be made.

Choice of representations.

- Specific classes for moves and paths, or some encoding?
- Object-oriented methods, or naked data structures with functions?

The present elaboration is just one solution, and not necessarily the shortest one.
Guiding Principles for Good Design

- Name everything you can.
- Put operations into natural scopes.
- Keep degrees of freedom for future refinements.
Course Conclusion
Traits of Functional Programming

Functional programming provides a coherent set of notations and methods based on

- higher-order functions,
- case classes and pattern matching,
- immutable collections,
- absence of mutable state,
- flexible evaluation strategies: strict/lazy/by name.

A useful toolkit for every programmer.

A different way of thinking about programs.
More Material on Scala

*Reference material:*

- Scala Ref Card *(adapted from a forum post by Laurent Poulain)*
- Twitter’s Scala School
- Programming in Scala
- Scala Tour

*To stay current:*

- Scala Meetups
- Typesafe Blog and Newsletter
- This Week in Scala Blogs
What Remains to Be Covered

Worthwhile topics we did not cover in this course:

Functional programming and state

- what does it mean to have mutable state?
- what changes if we add it?

Parallelism

- how to exploit immutability for parallel execution.

Domain-Specific Languages

- high-level libraries as embedded DSLs.
- interpretation techniques for external DSLs.