What’s a function?
Functions have inputs and outputs

Here’s a function that maps integers to integers:

```
public static int square(int x) {
    return x*x;
}
```

Here’s a function that takes the length of a list:

```
public static <T> int length(IList<T> list) {
    if(list.empty()) return 0;
    return 1 + length(list.tail());
}
```
Functions are just objects

We saw this in week 2 when we talked about lambdas

```java
public <Q> IList<Q> map(Function<T,Q> f) {
    return new LazyList<>(
        f.apply(headVal),
        () -> futureTail.get().map(f));
}
```

We’re passing a lambda to “map” over elements of the list

<table>
<thead>
<tr>
<th>The original function</th>
<th>Function&lt;T,Q&gt; f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applying the function</td>
<td>f.apply(headVal)</td>
</tr>
<tr>
<td>Making a new function</td>
<td>() -&gt; futureTail.get().map(f)</td>
</tr>
</tbody>
</table>
Functions are defined in an environment

Functions with unbound variables look outward to get them

```java
public <Q> IList<Q> map(Function<T,Q> f) {
    return new LazyList<>() {
        f.apply(headVal),
        () -> futureTail.get().map(f));
    }
}
```

<table>
<thead>
<tr>
<th>futureTail</th>
<th>Member variable in LazyList</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>The parameter of map()</td>
</tr>
</tbody>
</table>

**Closure**: a function plus the environment where it was defined

Functions hold onto this, even after the environment is "gone".
Higher-order functions
Function composition

Mathematicians have been doing this forever.
The composition operator $\circ$ takes two functions and gives you another function

\[
\begin{align*}
  f(x) &= \cdots \\
  g(x) &= \cdots \\
  h &= f \circ g \\
  h(x) &= f(g(x))
\end{align*}
\]
Java 8 functions also support this!

```
interface Function<T,R> {
        returns a composed function that first applies this function to its input, and then applies the after function to the result.
    R apply(T t)
        applies this function to the given argument.
    default <R> Function<?, ?> before()
        returns a composed function that first applies the before function to its input, and then applies this function to the result.
    static <T> Function<T,T> identity()
        returns a function that always returns its input argument.
}
```
RPNCalc functions

Each function has the same input type and output type (Optional<IList<Double>>)

```java
static Optional<IList<Double>> add(Optional<IList<Double>> ostack) {
    return ostack.flatMap((stack) -> {
        if (stack.empty() || stack.tail().empty()) return Optional.empty();
        double result = stack.head() + stack.tail().head();
        return Optional.of(stack.tail().tail().add(result));
    });
}
```

This means that they’ll compose!
You probably noticed the pretty “registry”

```java
private static IMap<TokenType, UnaryOperator<Optional<IList<Double>>> registry = TreapMap.of(
    new KeyValue<>(TokenType.PLUS, RPNCalculator::add),
    new KeyValue<>(TokenType.TIMES, RPNCalculator::multiply),
    new KeyValue<>(TokenType.MINUS, RPNCalculator::subtract),
    new KeyValue<>(TokenType.DIVIDE, RPNCalculator::divide),
    new KeyValue<>(TokenType.DUP, RPNCalculator::dup),
    new KeyValue<>(TokenType.DROP, RPNCalculator::drop),
    new KeyValue<>(TokenType.SWAP, RPNCalculator::swap),
    new KeyValue<>(TokenType.EQUALS, RPNCalculator::noop),
    new KeyValue<>(TokenType.FAIL, RPNCalculator::fail),
    new KeyValue<>(TokenType.CLEAR, RPNCalculator::clear)
);
```

A UnaryOperator<T> is like a Function<T,T>
(It’s actually a subtype. For now, pretend it’s just shorthand.)
But first, what about pushing numbers?

We’re treating new numbers as something special

```java
static Optional<IList<Double>> eval(Optional<IList<Double>> ostack, Token<TokenType> token) {
    if (token.type == TokenType.NUMBER) {
        return ostack.map((stack) -> stack.add(Double.parseDouble(token.data)));
    } else {
        UnaryOperator<Optional<IList<Double>>> operator =
            registry.oget(token.type).orElse(RPNCalculator::fail);
        return operator.apply(ostack);
    }
}
```

We “need” to fix this.
But first, what about pushing numbers?

We’re treating new numbers as something special

```java
static Optional<IList<Double>> eval(Optional<IList<Double>> ostack, Token<TokenType> token) {
    if (token.type == TokenType.NUMBER) {
        return ostack.map((stack) -> stack.add(Double.parseDouble(token.data)));
    }
    else {
        UnaryOperator<Optional<IList<Double>>> operator =
            registry.oget(token.type).orElse(RPNCalculator::fail);
        return operator.apply(ostack);
    }
}
```

We “need” to fix this.
Here’s a function that returns a function

Give it a number, it gives you a function that pushes that number

```java
static UnaryOperator<Optional<IList<Double>>> numberPusher(double number) {
    return ostack -> ostack.map(stack -> stack.add(number));
}
```

Now, we can represent every stack operation as a function!
Here’s a function that returns a function

Give it a number, it gives you a function that pushes that number

```java
static UnaryOperator<Optional<IList<Double>>> numberPusher(double number) {
    return ostack -> ostack.map(stack -> stack.add(number));
}
```

Now, we can represent every stack operation as a function!
Composing RPN stack operations

// \( f(x) = (x + 10) \times 27 \)

```java
Function<Optional<ILList<Double>>,Optional<ILList<Double>>> push27 = 
    RPNCalculator.numberPusher(27);
Function<Optional<ILList<Double>>,Optional<ILList<Double>>> push10 = 
    RPNCalculator.numberPusher(10);
Function<Optional<ILList<Double>>,Optional<ILList<Double>>> add = 
    RPNCalculator::add;
Function<Optional<ILList<Double>>,Optional<ILList<Double>>> mult = 
    RPNCalculator::multiply;

Function<Optional<ILList<Double>>,Optional<ILList<Double>>> f =
    push10.andThen(add).andThen(push27).andThen(mult);
```
Composing RPN stack operations

// \( f(x) = (x + 10) \times 27 \)

```java
Function<Optional<IList<Double>>,Optional<IList<Double>>> push27 = 
    RPNCalculator.numberPusher(27);
Function<Optional<IList<Double>>,Optional<IList<Double>>> push10 = 
    RPNCalculator.numberPusher(10);
Function<Optional<IList<Double>>,Optional<IList<Double>>> add = 
    RPNCalculator::add;
Function<Optional<IList<Double>>,Optional<IList<Double>>> mult = 
    RPNCalculator::multiply;

Function<Optional<IList<Double>>,Optional<IList<Double>>> f = 
    push10.andThen(add).andThen(push27).andThen(mult);
```

Function composition!
Function<Optional<IList<Double>>, Optional<IList<Double>>> f2 =
  RPNCalculator.numberPusher(10)
  .andThen(RPNCalculator::add)
  .andThen(RPNCalculator.numberPusher(27))
  .andThen(RPNCalculator::multiply);
Written more concisely

// $f(x) = (x + 10) \times 27$

```java
Function<Optional<IList<Double>>, Optional<IList<Double>>> f2 =
    RPNCalculator.numberPusher(10)
    .andThen(RPNCalculator::add)
    .andThen(RPNCalculator.numberPusher(27))
    .andThen(RPNCalculator::multiply);

assertEquals(
    Optional.of(List.of((3.0 + 10.0) * 27.0)),
    f2.apply(RPNCalculator.of(3.0)));```
How do you debug these things?

Q: How do you see what all those functions are doing?
A0: Set breakpoints and single-step (very painful!)
A1: Rewrite them to internally write to the Log (old school, ugly!)
A2: Wrap them with other functions that write to the Log (new school!)

New this week in Log.java:

```java
public static <A,B> Function<A,B> iWrap(
    String tag,
    Function<A, B> func) {

    return input -> {
        B output = func.apply(input);
        Log.i(tag, () -> String.format("%s -> %s", input.toString(), output.toString()));
        return output;
    }
}
```
Version 3: With wrapped functions

// f(x) = (x + 10) * 27

Function<Optional<IList<Double>>,Optional<IList<Double>>> f3 =
    Log.iWrap(TAG, RPNCalculator.numberPusher(10))
    .andThen(Log.iWrap(TAG, RPNCalculator::add))
    .andThen(Log.iWrap(TAG, RPNCalculator::numberPusher(27)))
    .andThen(Log.iWrap(TAG, RPNCalculator::multiply));
Version 3: With wrapped functions

// \( f(x) = (x + 10) \times 27 \)

```java
Function<Optional<IList<Double>>, Optional<IList<Double>>> f3 =
    Log.iWrap(TAG, RPNCalculator.numberPusher(10))
    .andThen(Log.iWrap(TAG, RPNCalculator::add))
    .andThen(Log.iWrap(TAG, RPNCalculator::numberPusher(27)))
    .andThen(Log.iWrap(TAG, RPNCalculator::multiply));
```

Oct 23, 2015 3:17:05 PM RPNCalculatorTest
INFO: Optional[3.0] -> Optional[10.0 3.0]
Oct 23, 2015 3:17:05 PM RPNCalculatorTest
INFO: Optional[10.0 3.0] -> Optional[13.0]
Oct 23, 2015 3:17:05 PM RPNCalculatorTest
INFO: Optional[13.0] -> Optional[27.0 13.0]
Oct 23, 2015 3:17:05 PM RPNCalculatorTest
INFO: Optional[27.0 13.0] -> Optional[351.0]
Digression: Monads

We very briefly mentioned this earlier in the semester

monads are elephants
monads are just monoids in the category of endofunctors
monads are trees with grafting
monads are elephants part 3
monads are burritos
Let’s break this down

“Monads are just monoids in the category of endofunctors”

Okay... we need to:

- define each of those terms
- explain why it’s cool
- and explain what this has to do with higher-order functions
A “monoid” is a mathematical entity with some useful properties:

- There is at least one “element” within the monoid *(think: integers in \( \mathbb{Z} \))*
- There is an associative binary operator that’s “closed” *(think: integer addition)*
  \[
  \forall a, b \in \mathbb{Z} \quad : \quad a + b \in \mathbb{Z}
  \]
  \[
  \forall a, b, c \in \mathbb{Z} \quad : \quad a + (b + c) = (a + b) + c
  \]
- There’s an “identity” element *(think: the integer zero)*
Functors and endofunctors

A functor is a mapping from one category to another
Example: a function from strings to integers (e.g., `String.length()`)  

An endofunctor is a mapping from a category to itself
Example: a function from strings to strings (e.g., `String.toLowerCase()`)  

Our UnaryOperators are endofunctors!

(And this matters why?)
“Monads are just monoids ...”

Unary Operators have an associative operator: function composition

The `f.andThen(g)` operation composes two functions `f` and `g`

Function composition is a binary operation (combine two funcs) and is associative

\[
f \circ (g \circ h) = (f \circ g) \circ h
\]

The resulting functions are exactly the same

We have an identity function \((x \rightarrow x)\)

Function composition is closed

We'll always get another function when composing functions like this, never an error.

Therefore... we have monads!

(Again, yes, but why?)
Monadic function composition

No matter how we compose our optional-stack operators, we’ll always get another
Function<Optional<IList<Double>>,Optional<IList<Double>>> f2 =
   RPNCalculator.numberPusher(10)
   .andThen(RPNCalculator::add)
   .andThen(RPNCalculator.numberPusher(27))
   .andThen(RPNCalculator::multiply);

By using Optional, we can stay closed (monadic) even when errors happen
So we can safely compose these functions in any order, without our code breaking.

Monadic coding = general-purpose, safe coding
Something worth keeping in mind when designing your adventure game.
Java8 vs. “real” functional languages

Haskell loves this monadic stuff
They use it to “emulate” mutation, among other crazy things
Java8 vs. “real” functional languages

Haskell loves this monadic stuff
They use it to “emulate” mutation, among other crazy things

The idea of functions taking Optional<T> to Optional<T> is ubiquitous
Most functional languages have dedicated features to make this look pretty.
Java? Not so much.

```java
static Optional<IList<Double>> add(Optional<IList<Double>> ostack) {
    return ostack.flatMap((stack) -> {
        if (stack.empty() || stack.tail().empty()) return Optional.empty();
        double result = stack.head() + stack.tail().head();
        return Optional.of(stack.tail().tail().add(result));
    });
}
```

“okay”
“ugly”
Still, we can write beautiful Java8 code

This is (almost) as good as it gets

```java
/**
 * Given a token, return a function (from optional stacks to optional stacks)
 * corresponding to that token
 */
static Function<Optional<IList<Double>>, Optional<IList<Double>>> getFunction(Token<TokenType> token) {
    return (token.type == TokenType.NUMBER)
        ? numberPusher(Double.parseDouble(token.data))
        : registry.oget(token.type).orElse(RPNCalculator::fail);
}

/**
 * Given a list of tokens, return a function (from optional stacks to optional stacks) corresponding
 * to each token applied in sequence
 */
static Function<Optional<IList<Double>>, Optional<IList<Double>>> getFunction(IList<Token<TokenType>> tokenList) {
    if (tokenList.empty())
        return Function.identity();
    return getFunction(tokenList.head()).andThen(getFunction(tokenList.tail()));
}
```
Still, we can write beautiful Java8 code

This is (almost) as good as it gets

```java
/**
 * Given a token, return a function (from optional stacks to optional stacks)
 * corresponding to that token
 */
static Function<Optional<IList<Double>>,Optional<IList<Double>>> getFunction(Token<TokenType> token) {
    return (token.type == TokenType.NUMBER)
        ? numberPusher(Double.parseDouble(token.data))
        : registry.oget(token.type).orElse(RPNCalculator::fail);
}

/**
 * Given a list of tokens, return a function (from optional stacks to optional stacks) corresponding
 * to each token applied in sequence
 */
static Function<Optional<IList<Double>>,Optional<IList<Double>>>
getFunction(IList<Token<TokenType>> tokenList) {
    if (tokenList.empty())
        return Function.identity();
    return getFunction(tokenList.head()).andThen(getFunction(tokenList.tail()));
}
```

These type declarations are awful (other languages infer them)
Still, we can write beautiful Java8 code

This is (almost) as good as it gets

```java
/**
 * Given a token, return a function (from optional stacks to optional stacks)
 * corresponding to that token
 */
static Function<Optional<IList<Double>>,Optional<IList<Double>>> getFunction(Token<TokenType> token) {
    return (token.type == TokenType.NUMBER)
        ? numberPusher(Double.parseDouble(token.data))
        : registry.oget(token.type).orElse(RPNCalculator::fail);
}

/**
 * Given a list of tokens, return a function (from optional stacks to optional stacks) corresponding to each token applied in sequence
 */
static Function<Optional<IList<Double>>,Optional<IList<Double>>> getFunction(IList<Token<TokenType>> tokenList) {
    if(tokenList.empty())
        return Function.identity();
    return getFunction(tokenList.head()).andThen(getFunction(tokenList.tail()));
}
```

Every operation can be a function! Elegant and extensible.
Still, we can write beautiful Java8 code

This is (almost) as good as it gets

```java
/**
 * Given a token, return a function (from optional stacks to optional stacks)
 * corresponding to that token
 */
static Function<Optional<IList<Double>>,Optional<IList<Double>>> getFunction(Token<TokenType> token) {
    return (token.type == TokenType.NUMBER)
        ? numberPusher(Double.parseDouble(token.data))
        : registry.oget(token.type).orElse(RPNCalculator::fail);
}

/**
 * Given a list of tokens, return a function (from optional stacks to optional stacks) corresponding to each token applied in sequence
 */
static Function<Optional<IList<Double>>,Optional<IList<Double>>>
    getFunction(IList<Token<TokenType>> tokenList) {
    if (tokenList.empty())
        return Function.identity();
    return getFunction(tokenList.head()).andThen(getFunction(tokenList.tail()));
}
```

Recursive, monadic function composition!
One last trick before we go

Aren’t you sick of seeing these long type signatures?
Function<Optional<IList<Double>>,Optional<IList<Double>>>

There’s some hope here!
interface CalcOp extends UnaryOperator<Optional<IList<Double>>>>

You can extend an interface like this and it mostly works
But you can’t do that for Optional
  1) It’s not an interface, it’s a class
  2) It’s final, so you can’t extend it either
You could build a “class Stack” that internally keeps the Optional<IList<Double>>
  1) One more layer of “.get()” methods to make your code ugly
Here’s how it ultimately looks

```cpp
// f(x) = (x + 10) * 27
RPNCalculator.CalcOp f2 =
    RPNCalculator.numberPusher(10)
    .andThenOp(RPNCalculator::add)
    .andThenOp(RPNCalculator.numberPusher(27))
    .andThenOp(RPNCalculator::multiply);
```
Here’s how it ultimately looks

```java
// f(x) = (x + 10) * 27
RPNCalculator.CalcOp f2 =
    RPNCalculator.numberPusher(10)
    .andThenOp(RPNCalculator::add)
    .andThenOp(RPNCalculator.numberPusher(27))
    .andThenOp(RPNCalculator::multiply);
```

Way better than what we had before:

```java
Function<Optional<IList<Double>>, Optional<IList<Double>>>
Here’s how it ultimately looks

```cpp
// f(x) = (x + 10) * 27
RPNCalculator.CalcOp f2 =
    RPNCalculator.numberPusher(10)
    .andThenOp(RPNCalculator::add)
    .andThenOp(RPNCalculator::numberPusher(27))
    .andThenOp(RPNCalculator::multiply);
```

We couldn’t call it `.andThen` because that conflicted with `Function.andThen`
If you want to try this in your own code...

There are some subtle tricks where Java doesn’t let you do what you want. That’s why we ended up with “andThenOp” instead of “andThen”.

There are extensive comments in the code that explain why.
You’re not required to understand this stuff!

This trick makes you code more beautiful, but...
Focus on correctness first, beauty second. If you’re spending hours trying to figure out the particularities of the Java type system just to make your code look cleaner, then you’re not being particularly productive.
Inspiration for this lecture brought to you by a beautiful talk on “Functional Programming Patterns” by Scott Wlaschin
It’s all about Microsoft’s F# programming language, but the ideas mostly still work in Java8.

http://fsharpforfunandprofit.com/fppatterns/