

# Comp 311

# Functional Programming

Eric Allen, PhD  
Vice President, Engineering  
Two Sigma Investments, LLC

# Passing Function Literals As Arguments

```
val xs = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Empty)))))  
    filter(_ < 3, xs) ↪* Cons(1, Cons(2, Empty))
```

# Guidelines On Using Function Literals

- Function literals are well-suited to situations in which:
  - The function is only used once
  - The function is not recursive
  - The function does not constitute a key concept in the problem domain

# Comprehensions

$$\{2x \mid x \in xs\}$$

# Mapping a Computation Over a List

```
def double(xs: List) = {  
  xs match {  
    case Empty => Empty  
    case Cons(y,ys) => Cons(y * y, double(ys))  
  }  
}
```

# Mapping a Computation Over a List

```
def negate(xs: List) = {  
  xs match {  
    case Empty => Empty  
    case Cons(y,ys) => (-y, negate(ys))  
  }  
}
```

# Negation as a Comprehension

$$\{-x \mid x \in xs\}$$

# Generalizing a Mapping Computation

```
def map(f: Int => Int, xs: List) = {  
  xs match {  
    case Empty => Empty  
    case Cons(y,ys) => Cons(f(y), map(f,ys))  
  }  
}
```



# Mapping a Computation Over a List

```
val xs = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Empty)))))
```

```
negate(xs) ↦*
```

```
Cons(-1, Cons(-2, Cons(-3, Cons(-4, Cons(-5, Cons(-6, Empty)))))
```

```
double(xs) ↦*
```

```
Cons(1, Cons(4, Cons(9, Cons(16, Cons(25, Cons(36, Empty)))))
```

# Mapping a Computation Over a List

```
val xs = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Empty)))))
```

```
map(-_, xs) ↦*
```

```
Cons(-1, Cons(-2, Cons(-3, Cons(-4, Cons(-5, Cons(-6, Empty)))))
```

```
map(x => x * x, xs) ↦*
```

```
Cons(1, Cons(4, Cons(9, Cons(16, Cons(25, Cons(36, Empty)))))
```

# Recall Our Sum Function Over Lists

```
def sum(xs: List): Int = {  
  xs match {  
    case Empty => 0  
    case Cons(y, ys) => y + sum(ys)  
  }  
}
```

In Mathematics, We Might  
Write this as a Summation

$$\sum_{x \in X} x$$

# And Our Product Function Over Lists

```
def product(xs: List): Int = {  
  xs match {  
    case Empty => 1  
    case Cons(y, ys) => y * sum(ys)  
  }  
}
```

In Mathematics, We Might  
Write this as a Product

$$\prod_{x \in x s} x$$

# We Abstract to a Reduction Function Over Lists

```
def reduce(base: Int, f: (Int, Int) => Int, xs: List): Int = {  
  xs match {  
    case Empty => base  
    case Cons(y,ys) => f(y, reduce(base, f, ys))  
  }  
}
```

# Example Reductions

```
val xs = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Empty)))))
```

```
reduce(0, (x,y) => x + y, xs) ↦* 21
```

```
reduce(1, (x,y) => x * y, xs) ↦* 720
```



# Min and Max

```
def max(xs: List) = {  
  reduce(Int.MinValue, (x,y) => if (x > y) x else y, xs)  
}
```

```
def min(xs: List) = {  
  reduce(Int.MaxValue, (x,y) => if (x < y) x else y, xs)  
}
```

# Simplifying Function Literals

- When *each* parameter is used only once in the body of a function literal, and in the order in which they are passed:
  - We can drop the parameter list
  - We simply write the body with an `_` at the place where each parameter is used

For example,

`((x: Int, y: Int) => (x + y))`

becomes

`_ + _`

# Example Reductions

```
val xs = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Empty)))))
```

```
reduce(0, _+_, xs) ↦* 21
```

```
reduce(1, _*__, xs) ↦* 720
```

Note the multiple parameters



# Combinations of Maps and Reductions

$$\sum_{x \in xs} x^2 + 1$$

# Combinations of Maps and Reductions

```
reduce(0, _+_, map(x => x*x + 1, xs))
```

# Summation

```
def summation(xs: List, f: Int => Int) =  
  reduce(0, _+_, map(f, xs))
```

# Summation

```
def square(x:Int) = x * x  
summation(xs, square(_)+1)
```

# More Syntactic Sugar

- Functions defined with **def** can be passed as arguments whenever an expression of a compatible function type is expected
- What constitutes a compatible function type?



# Partially Applied Functions

- If we want to pass a function as an argument, but supply some of the arguments to the function ourselves, we can wrap an application to the function in a function literal:

```
map(x => x + 1, xs)
```

# Partially Applied Functions

- If we want to pass a function as an argument, but supply some of the arguments to the function ourselves, we can wrap an application to the function in a function literal:

```
map(x => x + 1, xs)
```

which is equivalent to

```
map(_ + 1, xs)
```

# Partially Applied Functions

- **Eta Expansion:** Wrapping a function in function literal that takes all of the arguments of `f` and immediately calls `f` with those arguments

`(x: Int) => square(x)`

is equivalent to

`square`

# Mapping a Computation Over a List

We can use eta expansion to pass operators  
as arguments:

```
map(x => -x, xs)
```

# Mapping a Computation Over a List

We can use eta expansion to pass operators  
as arguments:

```
map(-_, xs)
```

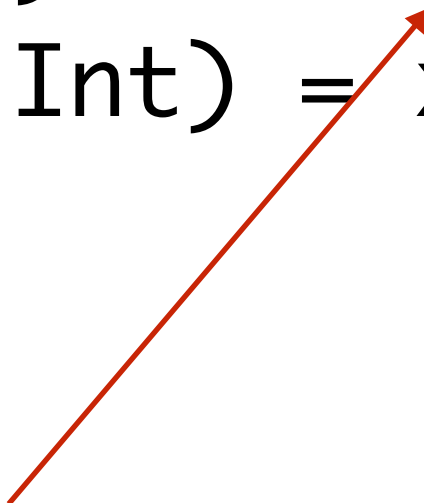
# Returning Functions as Values

# We Can Define Functions That Return Other Functions as Values

```
def add(x: Int): Int => Int = {  
  def addX(y: Int) = x + y  
  addX  
}
```

# We Can Define Functions That Return Other Functions as Values

```
def add(x: Int): Int => Int = {  
  def addX(y: Int) = x + y  
  addX  
}
```

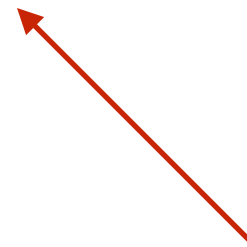


The explicit return type is needed because Scala type inference assumes an unapplied function is an error



# We Can Define Functions That Return Other Functions as Values

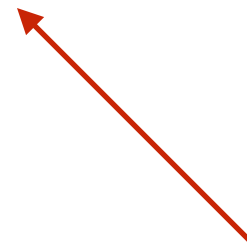
```
def add(x: Int) = {  
  def addX(y: Int) = x + y  
  addX _  
}
```



Alternatively, we can eta-expand addX to assure the type checker that we really do intend to return a function

# We Can Define Functions That Return Other Functions as Values

```
def add(x: Int) = {  
  def addX(y: Int) = x + y  
  addX _  
}
```



An underscore outside of parentheses in a function application denotes the entire tuple of arguments passed to the function

# We Can Define Functions That Return Other Functions as Values

```
def add(x: Int) = x + (_: Int)
```

We can instead define `add` by *partially* eta-expanding the `+` operator. But then we need to annotate the second operand with a type.

# Aside: Type Annotations

- In general, an expression annotated with a type is itself an expression:

`expr: Type`

- If the static type of `expr` is a subtype of `Type`, then the type of `expr:Type` is `Type`

# Partial Eta-Expansion

- We can partially eta-expand any function, but we need to annotate the argument types:

```
def reduce0 =  
  reduce(0, _: (Int, Int) => Int, _: List)
```

# Derivatives

```
def derivative(f: Double => Double, dx: Double) =  
  (x: Double) =>  
    (f(x + dx) - f(x)) /  
    dx
```

# Derivatives

```
def f(x: Double) = x * x  
def Df = derivative(f, 0.00001)
```

$f(4) \mapsto 16$

$Df(4) \mapsto 8.00000999952033$

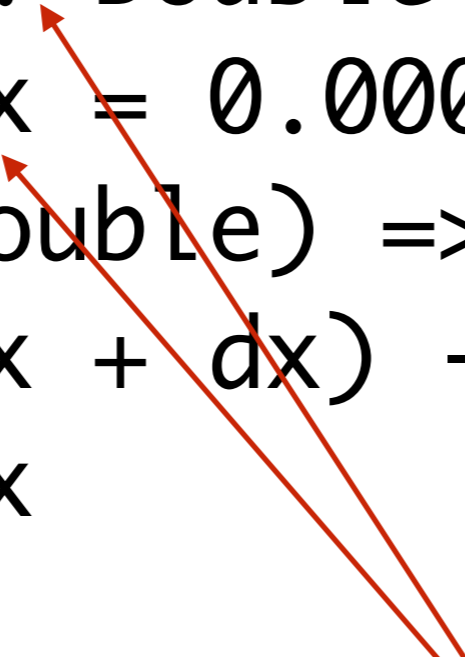
# Encapsulating dx

```
def D(f: Double => Double) = {  
  val dx = 0.00001  
  (x: Double) =>  
    (f(x + dx) - f(x)) /  
    dx  
}
```



# Encapsulating dx

```
def D(f: Double => Double) = {  
  val dx = 0.00001  
  (x: Double) =>  
    (f(x + dx) - f(x)) /  
    dx  
}
```



Our returned function “remembers”  
these values

# Applying a Derivative

```
def D(f: Double => Double) = {  
  val dx = 0.00001  
  (x: Double) =>  
    (f(x + dx) - f(x)) /  
    dx  
}
```

$D(f)(4) \mapsto$

$D((x: Double) => x * x)(4) \mapsto$

# Applying a Derivative

$D((x: \text{Double}) \Rightarrow x * x)(4) \mapsto$

`{val dx = 0.00001`

`(x: Double) =>`

`((x: Double) => x * x)(x + dx) -`

`(x: Double) => x * x)(x)) /`

`dx }`(4)  $\mapsto$

# Applying a Derivative

```
{(x: Double) =>
  ((x: Double) => x * x)(x + 0.00001) -
  ((x: Double) => x * x)(x)) /
  0.00001}(4) ↦
```

```
((x: Double) => x * x)(4 + 0.00001) -
  ((x: Double) => x * x)(4)) /
  0.00001 ↦
```

We must be careful to substitute only corresponding occurrences of x

# Applying a Derivative

$$\frac{((x: \text{Double}) \Rightarrow x * x)(4 + 0.00001) - ((x: \text{Double}) \Rightarrow x * x)(4)}{0.00001} \mapsto$$

$$\frac{((x: \text{Double}) \Rightarrow x * x)(4.00001) - ((x: \text{Double}) \Rightarrow x * x)(4)}{0.00001} \mapsto$$

$$\frac{(4.00001 * 4.00001) - (4 * 4)}{0.00001} \mapsto$$

# Applying a Derivative

$$\frac{((4.00001 * 4.00001) - (4 * 4))}{0.00001} \mapsto$$

$$\frac{(16.000080000099995 - 16)}{0.00001} \mapsto$$

$$8.00000999952033E-5 / 0.00001 \mapsto$$

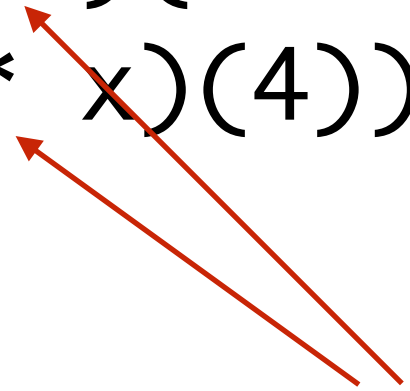
$$8.00000999952033$$

# Safe Substitution

# Applying a Derivative

```
{(x: Double) =>
  ((x: Double) => x * x)(x + 0.00001) -
  ((x: Double) => x * x)(x)) /
  0.00001}(4) ↦
```

```
((x: Double) => x * x)(4 + 0.00001) -
  ((x: Double) => x * x)(4)) /
  0.00001
```



In cases like this one, we can avoid accidental variable capture by selective renaming



# Safe Substitution

(a.k.a. Alpha Renaming)

- We can ensure we never accidentally substitute the wrong parameters by automatically renaming constants, functions, and parameters with *fresh* names
- A fresh name must not capture a name referred to in the scope of a parameter
- A fresh name must not be captured by a name in an enclosing scope

# Applying a Derivative

```
{(x: Double) =>
  ((y: Double) => y * y)(x + 0.00001) -
  ((z: Double) => z * z)(x)) /
  0.00001}(4) ↦
```

```
((y: Double) => y * y)(4 + 0.00001) -
  ((z: Double) => z * z)(4)) /
  0.00001
```

# Function Equivalence

- Now we have seen the three forms of function equivalence stipulated by the Lambda Calculus:
  - Alpha Renaming: Changing the names of a function's parameters does not affect the meaning of the function
  - Beta Reduction: To apply a function to an argument, reduce to the body of the function, substituting occurrences of the parameter with the corresponding argument
  - Eta Equivalence: Two functions are equivalent iff they are *extensionally equivalent*: They give the same results for all arguments

# Parametric Types

# Parametric Types

- We have defined two forms of lists: lists of ints and lists of shapes
- Many computations useful for one are useful for the other:
  - Map, reduce, filter, etc.
- It would be better to define lists and their operations once for all of these cases

# Parametric Types

- Higher-order functions take functions as arguments and return functions as results
- Likewise, *parametric types*, a.k.a., *a generic types*, takes types as arguments and return types as results

# Parametric Lists

- Every application of this parametric type to an argument yields a new type:

```
abstract class List[T] {  
    def ++(ys: List[T]): List[T]  
}
```

# Parametric Lists

- Every application of this parametric type to an argument yields a new type:

```
abstract class List[T <: Any] {  
  def ++(ys: List[T]): List[T]  
}
```



- We augment the declarations of type parameters to permit an upper bound on all instantiations of a parameter
  - By default, the bound is **Any**



# Syntax of Parametric Class Definitions

```
<modifiers> class C[T1 <: N, ..., TN <: N] extends N {  
  <ordinary class body>  
}
```

- We denote “naked” type parameters as **T1**, **T2**, etc.
- We denote all other types with **N**, **M**, etc.

# Syntax of Parametric Class Definitions

```
<modifiers> class C[T1 <: N, ..., TN <: N] extends N {  
  <ordinary class body>  
}
```

- Declared type parameters T1, ..., TN are in scope throughout the entire class definition, including:
  - The bounds of type parameters
  - The **extends** clause
- Object definitions must not be parametric

# Parametric Lists

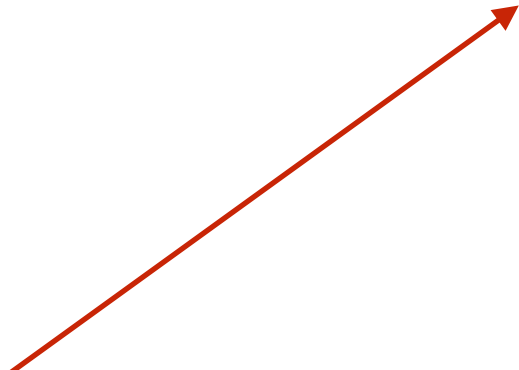
- Every application of this parametric type yields a new type:

```
List[Int]  
List[String]  
List[List[Double]]  
etc.
```

# Parametric Lists

- Every application (a.k.a., *instantiation*) of this parametric type yields a new type:

```
abstract class List[T] {  
  def ++(ys: List[T]): List[T]  
}
```



Note that our parametric type can be instantiated with type parameters, including its own!

# Parametric Lists

```
case class Empty[S]() extends List[S] {  
  def ++(ys: List[S]) = ys  
}
```

```
case class Cons[T](head: T, tail: List[T]) extends List[T] {  
  def ++(ys: List[T]) = Cons[T](head, tail ++ ys)  
}
```

# Parametric Lists

```
case class Empty[S]() extends List[S] {  
  def ++(ys: List[S]) = ys  
}
```

```
case class Cons[T](head: T, tail: List[T]) extends List[T] {  
  def ++(ys: List[T]) = Cons[T](head, tail ++ ys)  
}
```

Our definition requires a separate type `Empty[S]` for every instantiation of `S`. Thus we must define `Empty` as a class rather than an object.

# Type Environments

- To explain how to type check expressions in the context of parametric types, we must introduce the notion of *environments*
- We define a type parameter environment to hold a collection of zero or more type parameter declarations with their bounds
- Type environments can be extended with more declarations

# Type Checking a Class Definition

- To type check a parametric class definition:
  - Check the declarations of the class in a new type parameter environment that extends the enclosing environment with all its type parameters



# Type Checking a Function Definition

- To type check a function definition in environment  $E$ :
  - Check that the types of all parameters are *well-formed*
  - Find the type of the body of the function, substituting occurrences of parameters with their types
  - Ensure that the type of the body is a subtype of the declared return type (in environment  $E$ )

# Well-Formedness of Types

- A type is well-formed in environment  $E$  iff:
  - If it is a well-defined non-parametric type
  - It is a type parameter  $T$  in environment  $E$
  - It is an instantiation of a defined parametric type and:
    - All of its type arguments are well-formed types in  $E$
    - All of its type arguments respect the bounds on their corresponding type parameters

# Subtyping With Environments

- It is non-sensical to compare types in separate type environments:

```
case class Empty[S]() extends List[S] {  
  def ++(ys: List[S]) = ys  
}
```

```
case class Cons[T](head: T, tail: List[T]) extends List[T] {  
  def ++(ys: List[T]) = Cons[T](head, tail ++ ys)  
}
```

- Is S a subtype of T?

# Subtyping With Environments

- We must modify our subtyping rules to refer to an environment  $E$ :
  - $S <: S$  in  $E$
  - If  $S <: T$  in  $E$  and  $T <: U$  in  $E$  then  $S <: U$  in  $E$

# Subtyping With Environments

- If:
  - `class C[T1, ..., TN] extends D[U1, ...UM]`
  - and  $X_1, \dots, X_N$  are well-formed in  $E$
  - then  $C[X_1, \dots, X_N] <: D[U_1, \dots, U_M][T_1 \mapsto X_1, \dots, T_N \mapsto X_N]$  in  $E$

# Subtyping With Environments

- If:
  - `class C[T1, ..., TN] extends D[U1, ...UM]`
  - and  $X_1, \dots, X_N$  are well-formed in  $E$
  - then  $C[X_1, \dots, X_N] <: D[U_1, \dots, U_M][T_1 \mapsto X_1, \dots, T_N \mapsto X_N]$  in  $E$

We use this notation to indicate safe substitution of  $T_1$  for  $X_1$ ,  
...  $T_N$  for  $X_N$  in  $D[U_1, \dots, U_M]$

# Covariance

- Can one instantiation of a parametric type be a subtype of another?
- Currently our rules allow this only in the reflexive case:

`List[Int] <: List[Int] in E`

# Covariance

- It would be useful to allow some instantiations to be subtypes of another
- For example, we would like it to be the case that:

```
List[Int] <: List[Any]
```



# Covariance

- In general, we say that a parametric type  $C$  is covariant with respect to its type parameter  $S$  if:

$$S <: T \text{ in } E$$

implies

$$C[S] <: C[T] \text{ in } E$$

- We must be careful that such relationships do not break the soundness of our type system

# Covariance

- For a parametric type such as:

```
abstract class List[T <: Any] {  
    def ++(ys: List[T]): List[T]  
}
```

- And types  $S$  and  $T$ , such that  $S <: T$  in some environment  $E$ :
- What must we check about the body of class `List` to allow for `List[S] <: List[T]` in  $E$ ?

# Covariance

- Consider instantiations for types `String` and `Any`:

```
abstract class List[Any] {  
  def ++(ys: List[Any]): List[Any]  
}
```

```
abstract class List[String] {  
  def ++(ys: List[String]): List[String]  
}
```

# Covariance

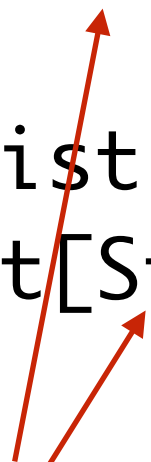
- If these were ordinary classes connected by an **extends** class:
- We would need to ensure that the overriding definition of **++** in class **List[String]** was compatible with the overridden definition in **List[Any]**

# Covariance

```
abstract class List[Any] {  
  def ++(ys: List[Any]): List[Any]  
}  
abstract class List[String] extends List[Any] {  
  def ++(ys: List[String]): List[String]  
}
```

# Covariance

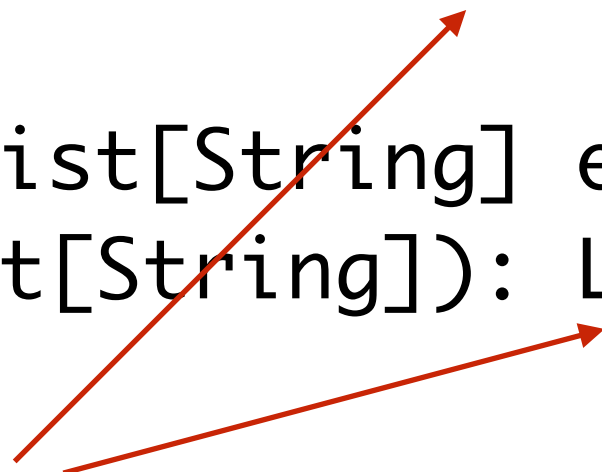
```
abstract class List[Any] {  
  def ++(ys: List[Any]): List[Any]  
}  
abstract class List[String] extends List[Any] {  
  def ++(ys: List[String]): List[String]  
}
```



But if `List[String] <: List[Any]` in E  
then this is not a valid override

# Covariance

```
abstract class List[Any] {  
  def ++(ys: List[Any]): List[Any]  
}  
abstract class List[String] extends List[Any] {  
  def ++(ys: List[String]): List[String]  
}
```



On the other hand, the return types  
are not problematic

# Covariance

- From our example, we can glean the following rule:
  - We allow a parametric class  $C$  to be covariant with respect to a type parameter  $T$  so long as  $T$  does not appear in the types of the method parameters of  $C$



# Covariance

```
abstract class List[+T] {}
```

- We stipulate that a parametric type is covariant in a parameter **T** by prefixing a **+** at the definition of **T**
- (We will return to our definition of **append** later)

# Covariance

```
case object Empty extends List[Nothing] {  
}
```

```
case class Cons[+T](head: T, tail: List[T])  
extends List[T] {  
}
```