L5: Currying, Closure and Type

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

October 9th, 2020

Architecture Research Group

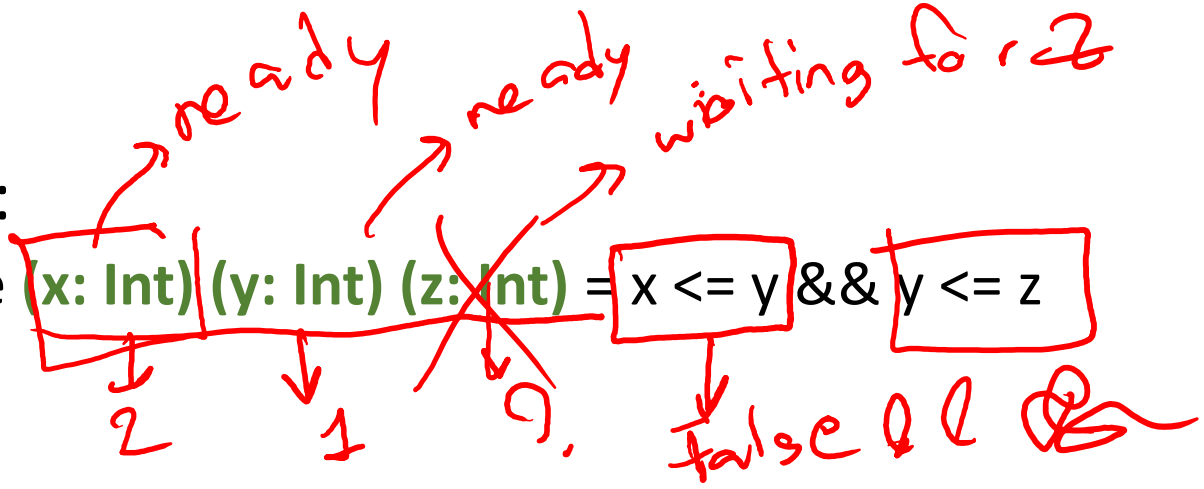
Computer Engineering, TGGS

Currying

Currying

- Instead of accepting parameters normally, accept through a sequence of functions
- def sortedUncurry(x: Int, y: Int, z: Int) = x <= y && y <= z
- val sortedTriple = { (x: Int) => (y: Int) => (z: Int) => x <= y && y <= z }

- Currying version:
  - def sortedTriple (x: Int) (y: Int) (z: Int) = x <= y && y <= z



Currying

- Benefits:
  - You can stage the function
    - Parts of the execution can run as soon as the values are ready
  - Maps well with dataflow model
  - This can allow the compiler and the hardware to be faster
    - Eliminate data dependency as soon as possible

- Actual efficiency: It depends
  - Compiler is very smart nowadays
  - Run a profiler to test the two formats

States and Mutable Variables

- We assumed variables are immutable
  - This is annoying in some cases
  - What if we need to store a state

- **State:** the intermediate steps that need to be stored
  - Real hardware also needs the concept of state

- So, many functional languages have mutable variables

- Benefit of mutable variables
  - Allow programmer to keep the state

Declaring Mutable Variables

- var x = value

- Example: implementing a while loop
  - Using currying and mutable variables

- def my_while (condition: => Boolean) (block: => Unit): Unit = {
  if (condition) {
  block
  my_while (condition) (block)
  } else ()
  }

Function Closure

Forall

- Recursive data types have another build-in utility
- x.forall(p) is the same as
- def forAll[T](x: List[T], p: T => Boolean): Boolean =

```
    x match {
        case Nil => true
        case h::t => p(h) && forAll(t, p)
    }
```

Apply p to elem of x

Forall: Example

- Consider the following expression type
- sealed trait Expr
  case class Constant(n: Double) extends Expr
  case class Negate(e: Expr) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr
- What if we want to map expression to certain func. f
- def map(f: Double => Double, e: Expr): Expr = e match {
  case Constant(x) => Constant(f(x))
  case Negate(e) => Negate(map(f, e))
  case Sum(e1,e2) => Sum(map(f,e1), map(f,e2))
  case Prod(e1,e2) => Prod(map(f,e1), map(f,e2)) }

Forall: Example

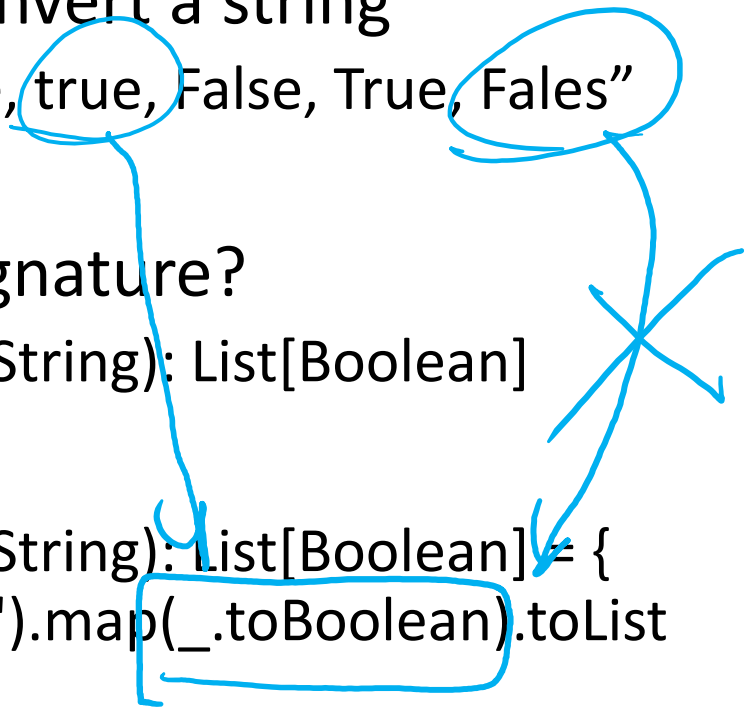
not on a list on our own data type

- Check if all expressions are positive
- def forAllConst(p: Double => Boolean, e: Expr): Boolean = e match
  {
  case Constant(x) => p(x)
  case Negate(e_) => forAllConst(p, e_)
  case Sum(e1,e2) => forAllConst(p, e1) && forAllConst(p, e2)
  case Prod(e1,e2) => forAllConst(p, e1) && forAllConst(p, e2)
  }

check if e is positive

Communicating Across Functions

- Let's say we want to convert a string
  - "True, False, True, False, true, False, True, Fales"
  - Into a list of Boolean
- What is the function signature?
  - def convertBoolList(st: String): List[Boolean]
- Step 1:
  - def convertBoolList(st: String): List[Boolean] = {
    val entries = st.split(",").map(_.toBoolean).toList
    entries
    }



- What might be the problem?

Communicating Across Functions

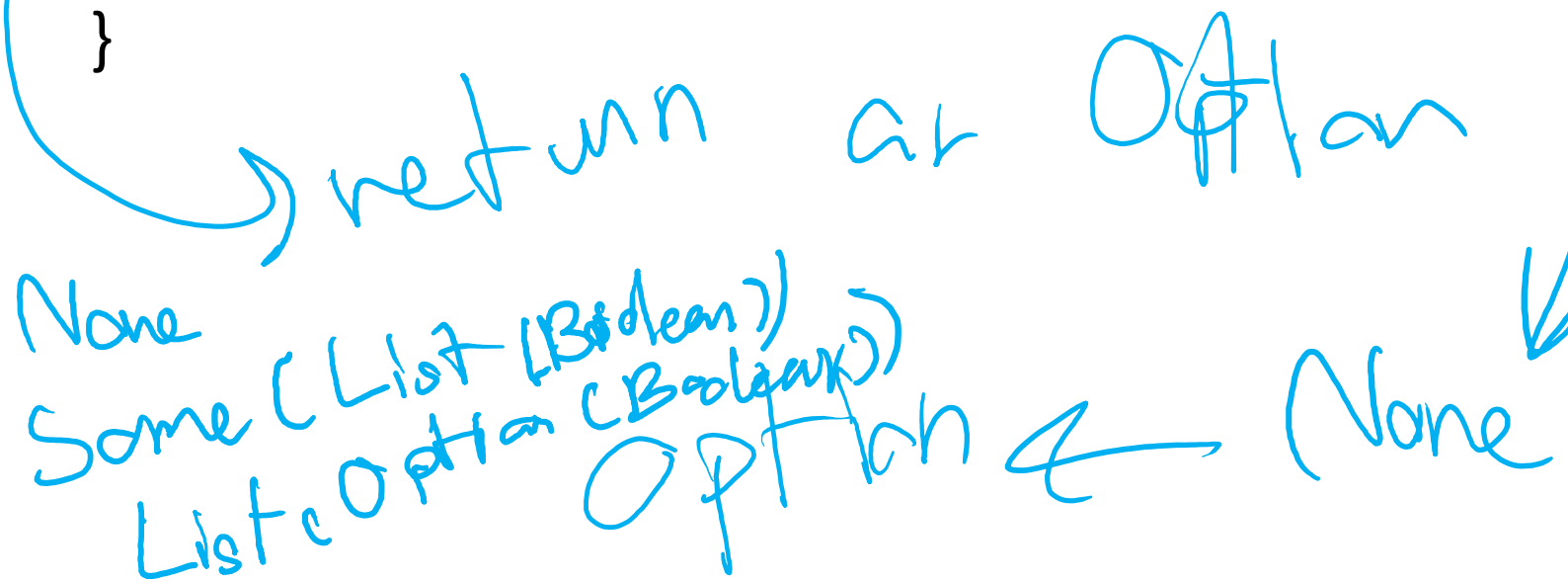
- Let's say we want to convert a string
  - "True, False, True, False, **true**, False, True, **Fales**"
  - Into a list of Boolean
- What is the function signature?
  - def convertBoolList(st: String): List[Boolean]
- Step 1:
  - def convertBoolList(st: String): List[Boolean] = {
    val entries = st.split(",").map(_.toBoolean).toList
    entries
    }
- What might be the problem?

Communicating Across Functions

- You need to communicate this problem
- We can make the function more expressive
  - Option type
- We can throw an exception

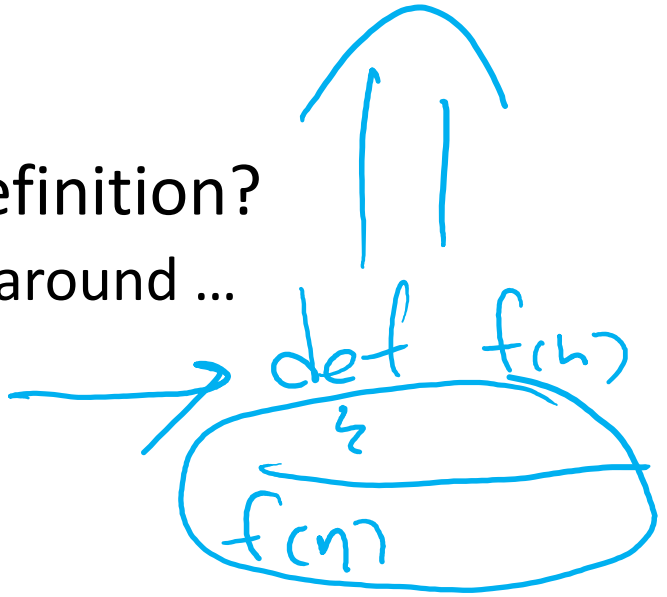
Communicating Across Functions

```
def convertBoolList(st: String): Option[List[Boolean]] = {
  try {
    val entries = st.split(",").map(_.toBoolean).toList
    Some(entries)
  } catch {
    case e: IllegalArgumentException => None
  }
}
```



Function Closure

- What is the scope of a function's definition?
  - Now that functions are being passed around …

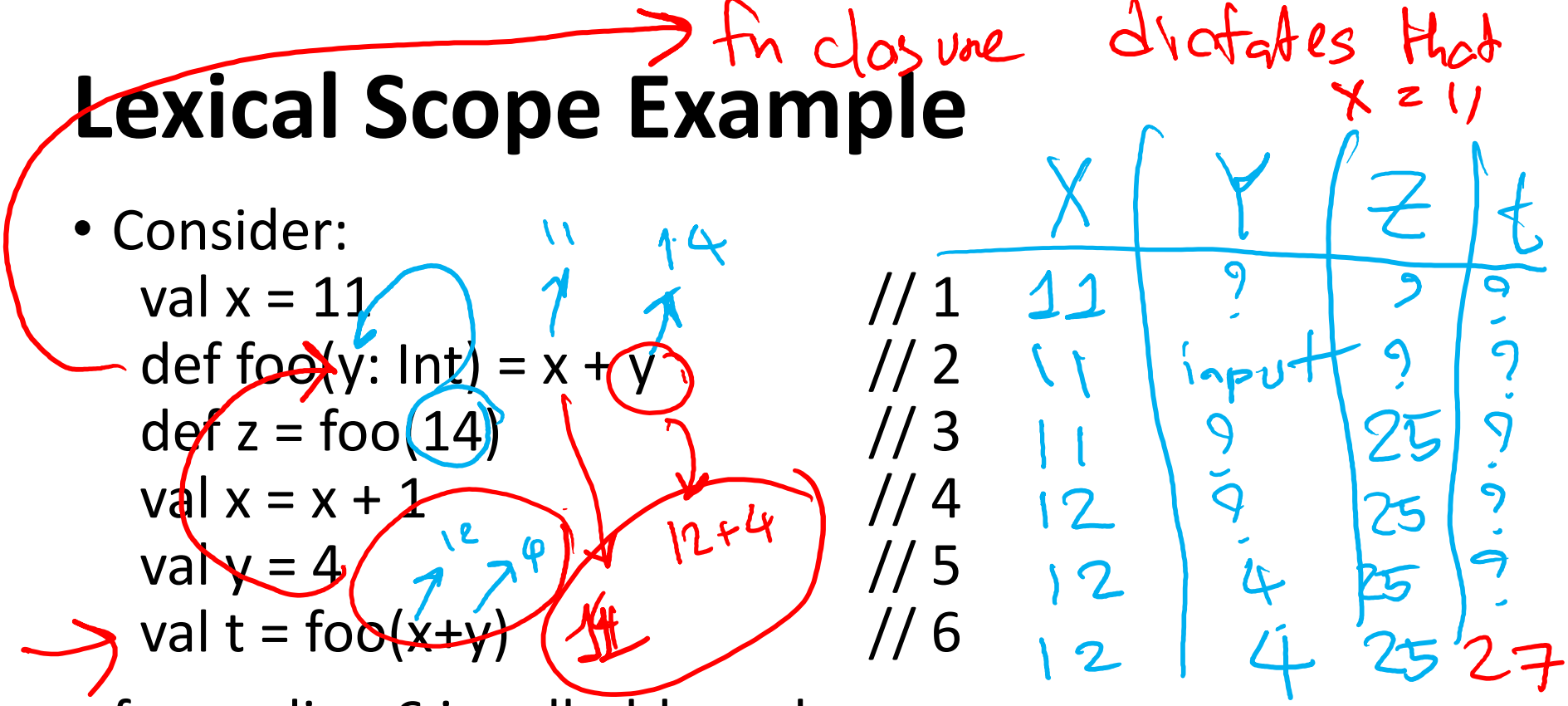


- Answer:
  - The body of a function is evaluated in the environment where the **function is defined**, not when it is called
- This is called ***the lexical scope***
- Let's do an example

Lexical Scope Example

- Consider:

```
val x = 11                          // 1
def foo(y: Int) = x + y             // 2
def z = foo(14)                     // 3
val x = x + 1                       // 4
val y = 4                           // 5
val t = foo(x+y)                    // 6
```



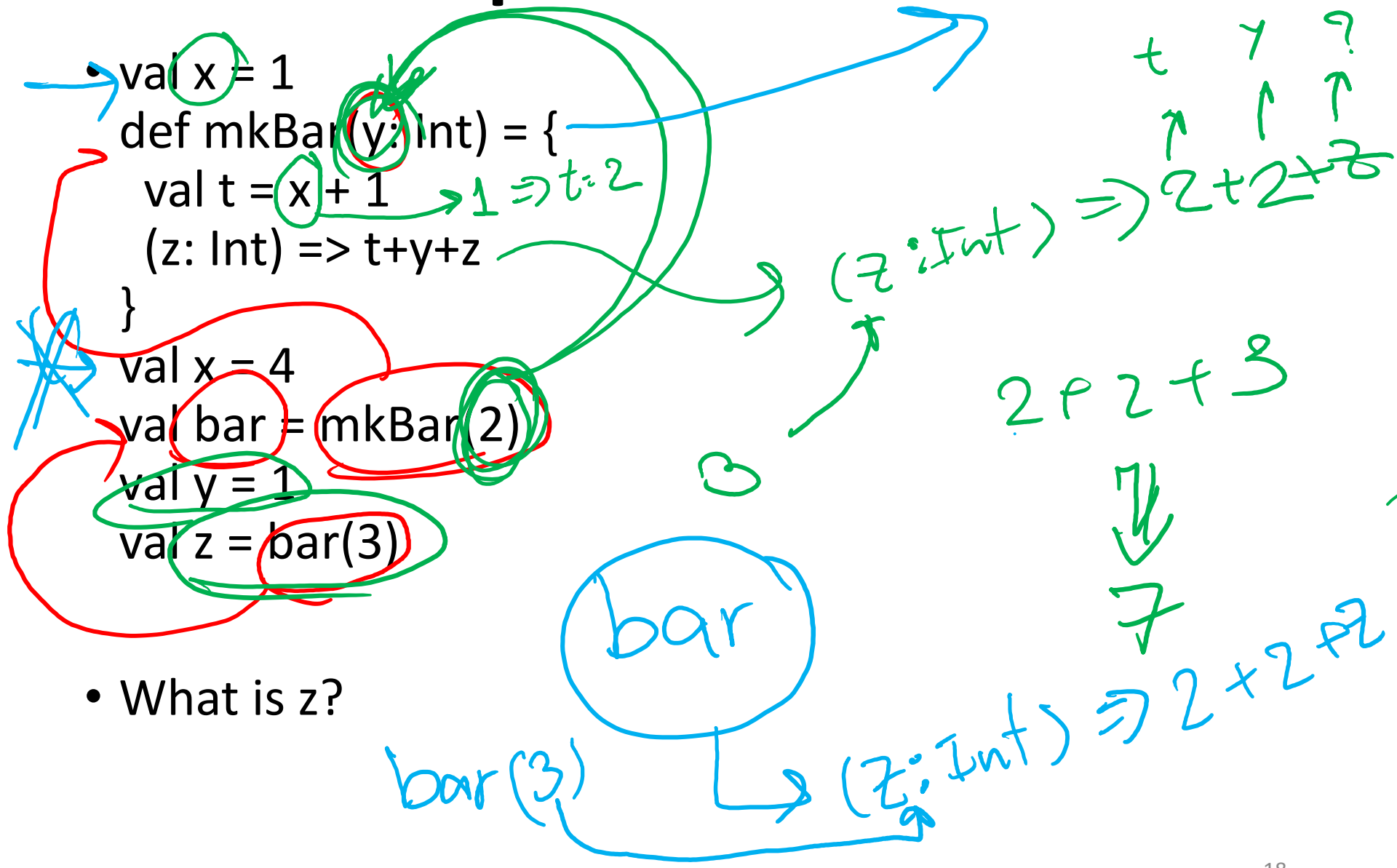
- foo on line 6 is called-by-value
  - Also, on that line, x is 12 and y is 4
- Inside foo itself, x is 11
  - y is the input parameter to foo (which is 16 from line 6)

Function Closure in Lexical Scope

- Using the last example
  - Somehow, foo takes the value x in the old environment
- Fundamentally, the execution will keep these old environment as needed
- A function definition has two parts
  - The code (the function you write)
  - The environment (at the point where you define the function)
- This part is called the function closure
  - You are teleported back to the old environment
- You cannot explicitly manipulate this environment

More Example

```
val x = 1
def mkBar(y: Int) = {
  val t = x + 1
  (z: Int) => t+y+z
}
val x = 4
val bar = mkBar(2)
val y = 1
val z = bar(3)
```



- What is z?

Dynamic Scope

→ Inverse of Lexical Scope

- Environment is used when the function is called
  - Instead of when it is defined

- PL researches shows more benefit for using lexical scoping

Using Function Closure

- Functions can be evluated at multiple places
  - A function body: not evaluated until the function is called
  - A function body: evaluated every time the function is called
  - A variable binding: evaluates its expression when the binding is evaluated, not every time the variable is used
- To avoid repeating computation, you can store the evaluation inside function closures

Example

- def longerThan(xs: List[String], s: String) =
  xs.filter(x => x.length > s.length)

- def longerThan(xs: List[String], s: String) = {
  val thresLen = s.length
  xs.filter(x => x.length > thresLen)
  }
- s.length is called once and bounded
  - And you use the anonymous function to compare with x.length

Example #2

- def fib(n:Int) : Long =
  if (n<=2) 1 else fib(n-1) + fib(n-2)
- def mkFibFoo(t:Int) = {
  (x:Long) => fib(t) + x
  }
- def mkBetterFibFoo(t: Int) = {
  val fibt = fib(t)
  (x: Long) => fibt + x
  }
- val f = mkFibFoo(45)
  // try f(1), f(2)
- val g = mkBetterFibFoo(45)
  // try g(1), g(2)

Diving into the Type System

The Type System

- Type: The prediction of the outcome of an expression and its property
  - This is known during the compile time
- If the expression is a string type, it will eventually evaluate to a string
  - This allows compiler to make more assumption
  - This allows programmers to make more assumption

Define a New Type in Scala

- We went through multiple of these examples

- Define a class, trait and objects
  - class ClassName
  - trait TraitName
  - object ObjectName
  - What are the differences between these three?

- You can also use “type” keyword
  - type Mytype = …

Using the Type After Declarations

- You can then utilize this newly declared type using:
- def fooA(x: ClassName) = x
- def fooB(x: TraitName) = x
- def fooC(x: ObjectName.type) = x
- def fooD(x: MyType) = x

- The .type for the objectName allows you to distinguish between a class and an object

Rules/Constraints

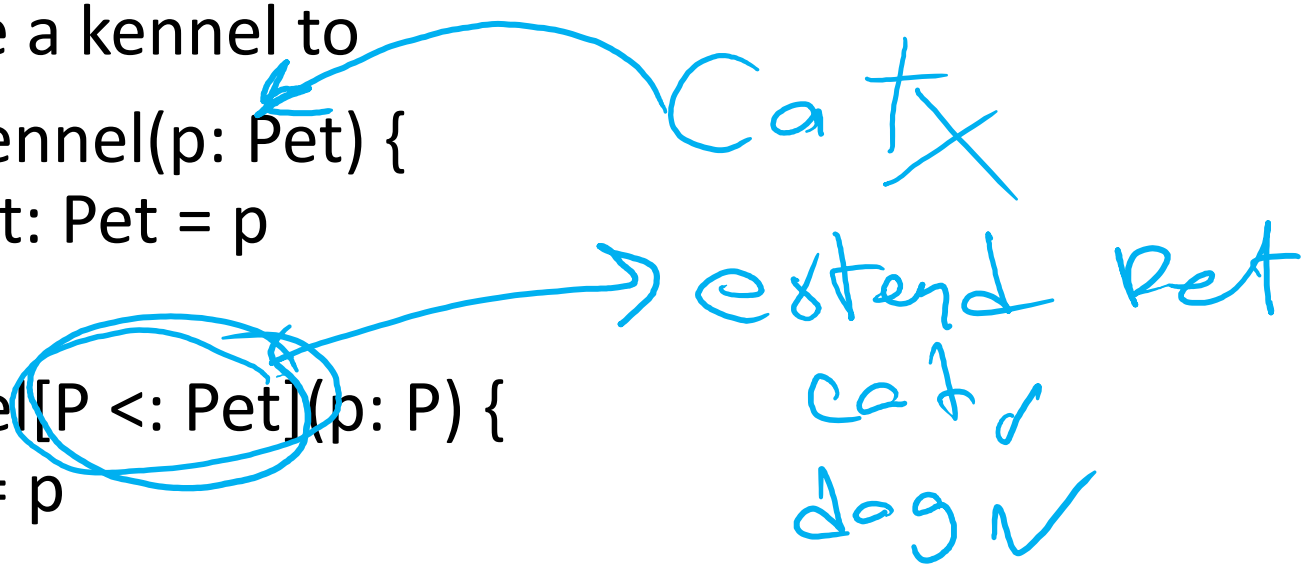
- Defining a type allows you to assert rules
- There are generally two kinds
- Upper bounds
  - A <: B means B → A in the type hierarchy (A extends B)
- Lower bounds
  - A >: B means A → B in the type hierarchy (B extends A)

Upperbound Examples

- Let’s first makes some nested declarations
- abstract class Animal {
  def name: String
  }
  abstract class Pet extends Animal {}
- class Cat extends Pet {
  override def name: String = "Cat“
  def meow: String = "Say meow“
  }
- class Dog extends Pet {
  override def name: String = "Dog“
  def bark: String = "Woof woof“
  }
- class Lion extends Animal {
  override def name: String = "Lion“
  def growl: String = "(muffled)“
  }

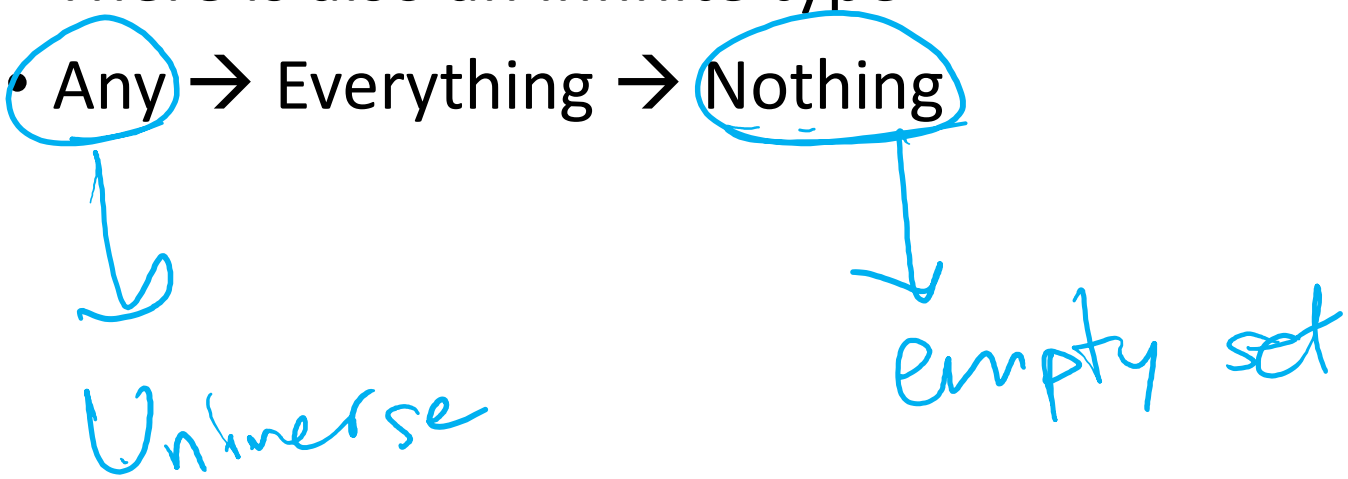
Upperbound Examples

- Let's create a kennel to
- class BadKennel(p: Pet) {
  def pet: Pet = p
  }
- class Kennel[P <: Pet](p: P) {
  def pet: P = p
  }
- Why is the second version better?
- val dogKennel = new Kennel[Dog](new Dog)
  - This create a new Kennel to keep one pet
- Can you do val lionKennel = new Kennel[Lion](new Lion)



Upperbound Limits

- There is also an infinite type
- Any → Everything → Nothing



Lowerbound

- The type selected must be equal or a supertype ***of the lower bound***
- class A {
  type B >: List[Int]
  def foo(a: B) = a
  }
- val x = new A { type B = Traversable[Int] }
  - This is ok because Traversable -> List
- Then you can use
  - x.foo(List(1,2)) // obviously
  - x.foo(Set(1,2)) // because Traversable[Int] -> Set[Int]
- How about val y = new A {type B = Set[Int]}

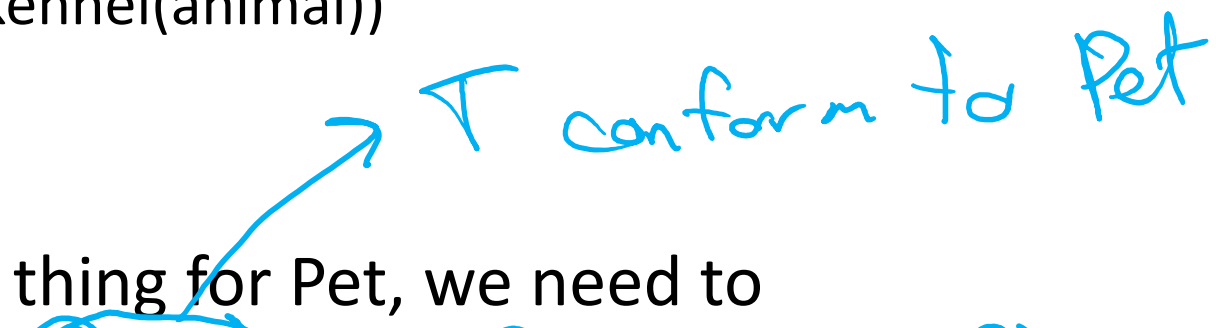
Type Parameters

- Let's look at
  - def pickRandom[T] (x: Seq[T]): T

- What does this do?

Type Parameters: Example

- Let's ties this to the animal/kennel example:
  - def kennelName(animal: …): (String, Kennel[…]) = {
    Val name = animal.name
    (name, new Kennel(animal))
    }



- To use the same thing for Pet, we need to
  - def kennelName[T <: Pet](animal: T): (String, Kennel[T]) = {
    val name = animal.name
    (name, new Kennel(animal))
    }

Variance

- Definition: The ability of type parameters to vary on high-kinded types
  - Say we can C[A]. A higher-kinded type C[A] is said to conform to C[B] if you can assign C[B] to C[A] with no error
- Three types of variance
  - Invariance: if A==B then C[A] will conform to C[B]
  - Covariance: if A→B, then C[A] → C[B]
  - Contravariance: if A→B, then C[B] → C[A]

Covariance vs. Contravariance

- Covarience
  - A list of cats is a subtype of a list of animal
  - Function that return a list of cat's type parameter is a covariance with a function that return a list of animal

- Contravarience
  - The parameter for a function that return a string given a cat is then contravariance with a parameter for a function that return a string given an animal
  - Basically the reversal of covariance

Using Variance in Scala

- Define covariance
  - Use +
  - Example: +T, List[+T]

- Define contravariance
  - Use –
  - Example: -T, List[-T]

Variance Example

- val catKennel = new Kennel(new Cat)
  val dogKennel = new Kennel(new Dog)
  def getName(k: Kennel[Pet]) = k.pet.name

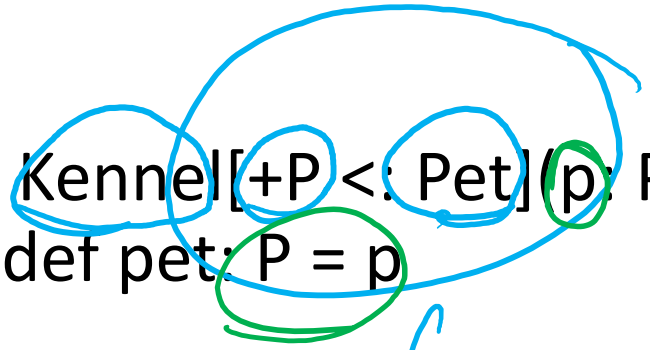
- Why is this not working?
  - The compiler **cannot make any assumptions you do tell them**
  - **But the language (scala) gives you that ability!**

- Anything missing?

Variance Example Cont.

- Let's fix the Kennel class

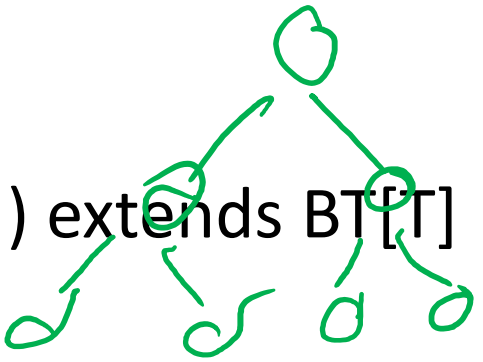
- class Kennel[+P <: Pet](p: P) {
  def pet: P = p
  }



tell Scala that anything that is a subtype of Pet (Cat, Dog) should be able to use this class

Binary Tree Example

- We can make the left passable/understandable to the binary tree
  - Basically passing the left as any tree type T
- sealed trait BT[T]
  - This must in fact be +T because the Leaf (i.e. BT[Nothing]) should be passable as BT[any T]
- object Leaf extends BT[Nothing]
- case class Node[T](l: BT[T],k: T,r: BT[T]) extends BT[T]

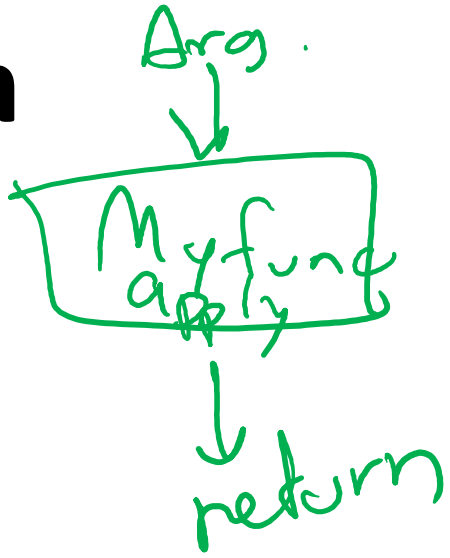


Create Your Own Function

- Let's say we have f: A→B
- What we learn so far is that it might be ok to pass in a ***wider range of input than A*** and produce and ***output that does not use all of B***

- What if we want to create a function that applies its argument

Create Your Own Function



- trait MyFunction[Arg, Return] {
  def apply(arg: Arg): Return
  }

- What is wrong with the code below?

```
val f: MyFunction[List[Int], Any] = new
MyFunction[Seq[Int], Double] {
    def apply(arg: Seq[Int]): Double = arg.sum
}
```

Relationship List[Int] → Seq[Int]
Any ⇒ Double

Create Your Own Function: Fix

- trait MyFunction**[-Arg, +Return]** {
  def apply(arg: Arg): Return
  }

- val f: MyFunction[List[Int], Any] = new MyFunction[Seq[Int], Double] {
  def apply(arg: Seq[Int]): Double = arg.sum
  }

Dependency

- What is a dependency?
- Why can this be useful?

- Can we manually “insert” this dependency?

Dependency Injection

- You might have heard this from software engineering
  - One object supplies the dependent item to another

obj1 → obje

- In this case, we can decouple the actual implementation away from the abstraction

- Let's use an example

Dependency Injection: Use Cases

- Let’s assume class X needs a database connection
  - val con = DBConnectionRepository.getByName("appDBConnection")
- Notice this creates the dependency, but the dependency is coupled to the repository
  - Basically you need the repo, and cannot really change the name

- Dependency injection aims to decouple this dependency
  - You can make the connection declaration
  - Then you can implement the repository later

What About Inheritance/Interface?

- Q: Why not just keep extending the traits to covers possible?
- trait FooAble {
  def foo = "this is an ordinary foo“
  }
- What if I create a code that depends on FooAble
- class BarUsingFooAble extends FooAble {
  def bar = "bar calls foo: " + foo
  }
- object Main extends App {
  val barWithFooAble = new BarUsingFooAble
  println(barWithFooAble.bar)
  }
- What is the problem here?

You Cannot Do This (Can't Compile)

- Using the "with" keyword
  - What is the "with" keyword?
- class BarUsingFooAble {
  def bar = "bar calls foo: " + foo
  }
- object Main extends App {
  val barWithFooAble = new BarUsingFooAble with FooAble
  println(barWithFooAble.bar)
  }
  - This is ''done'' at instantiation
- Why this does not compile?

What About Abstract Method?

- abstract class BarUsingFooAble {
  def foo: FooAble
  def bar = "bar calls foo: " + foo.foo
  }
- object Main extends App {
  val fooInstance = new FooAble {}
  val barWithFoo = new BarUsingFooAble {
  def foo = fooInstance
  }
  println(barWithFoo.bar)
  }
- This gets messy as you extends

Baking a Cake

- We can decouple the dependency using the cake method
  - What?
- Self-type annotation
- trait FooAble {
  def foo = "this is an ordinary foo“
  }
- class BarUsingFooAble {
  this: FooAble =>
  def bar = "bar calls foo: " + foo
  }

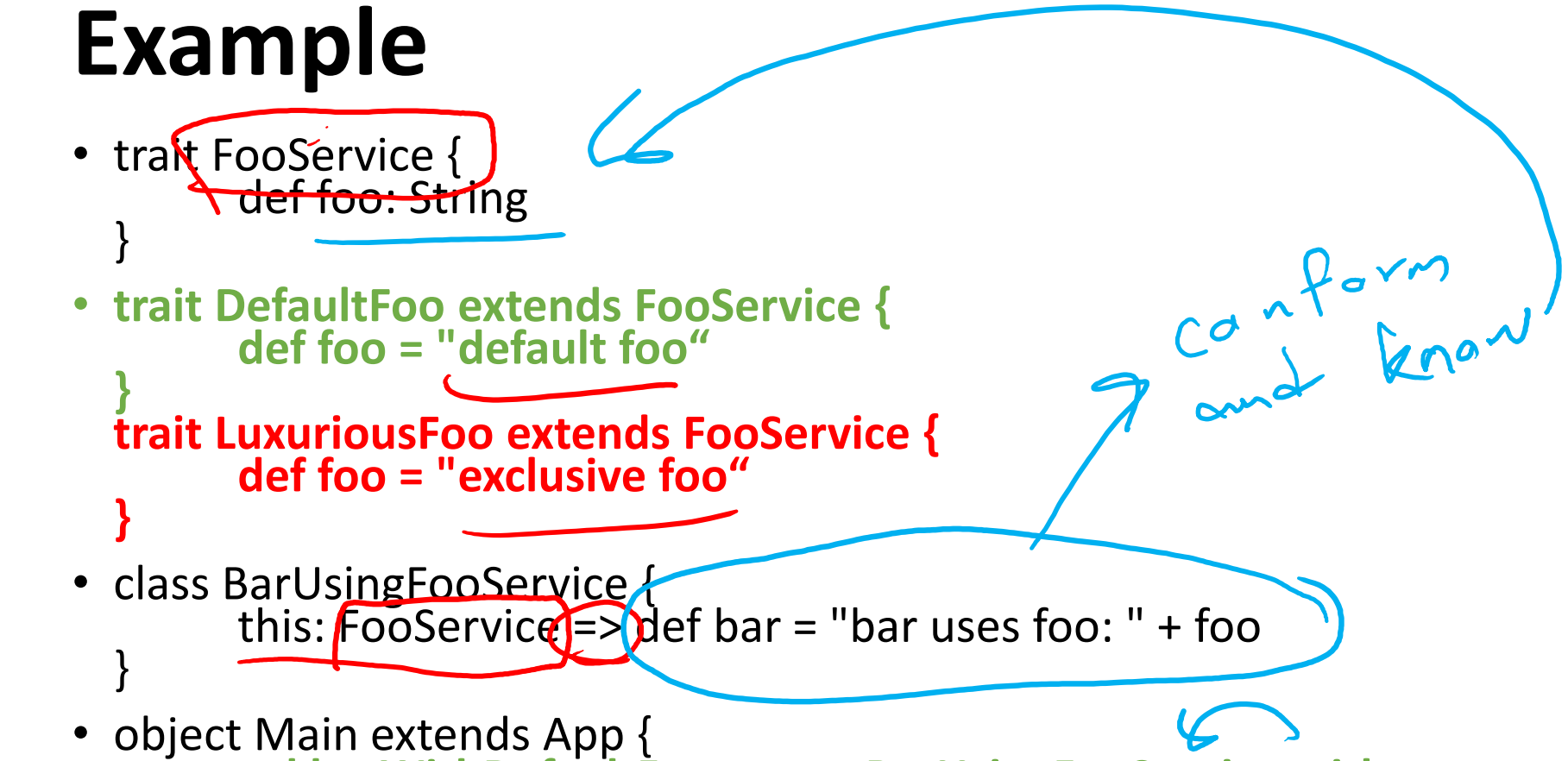
- Anything after the => can use ***methods and variables of FooAble***

Baking a Cake

- With the [trait] =>, …
  - We declare that the class depends on [trait]
- Difference between this and "extends":
  - extends is very type specific
    - You need to strictly have that exact type
  - Self-type annotation just say "I am declaring that whatever goes in the … will extend the trait"
    - It does not actually extend it yet
    - But it needs to conforms to [trait] (i.e., FooAble from the earlier example)
    - Hence, baking

Example

- trait FooService {
  def foo: String
  }
- trait DefaultFoo extends FooService {
  def foo = "default foo“
  }
  trait LuxuriousFoo extends FooService {
  def foo = "exclusive foo“
  }
- class BarUsingFooService {
  this: FooService => def bar = "bar uses foo: " + foo
  }
- object Main extends App {
  val barWithDefaultFoo = new BarUsingFooService with DefaultFoo
  val barWithBetterFoo = new BarUsingFooService with LuxuriousFoo
  println(barWithDefaultFoo.bar)
  println(barWithBetterFoo.bar)
- }



Before We Leave Today

In-class Exercise 9

- Please check our Canvas for the skeleton code
- We will try to implement a random draw using dependency injection
- Feels free to discuss your project idea

Next 2 Weeks Plan

- October 16th
  - Normal class + Review Session

- Your job after October 16th
  - Post questions on canvas
  - Prepare for the exam

- I will go through these questions on the following days:
  - October 20th Noon – 1 PM
  - October 22nd Noon – 1 PM
  - Recording will be posted on our Youtube channel
    - So you can watch these if you have conflicts

Next 2 Weeks Plan: Exam

- Exam: take-home
  - All material up until now
- But, October 23rd is a holiday and a long weekend

- Instead of me giving you 24 hours, I will
  - Send out the questions on October 22nd
  - You have until October 26th midnight to finish
  - Same length as if you are working on a 24-hour exam
  - Open everything
  - I will be online in Webex at some point
    - The time will be announced next week (I need to check my schedule)
    - This will be live Q&A, just don't ask me what's the answer :p
  - Then, additional questions is through email/discord/msging