


# FOUNDATION



Function

# Functional Programming



# First class function

```
def inc(x: Int): Int = x + 1
```

```
scala> inc(10)  
res0: Int = 11
```

```
val inc: Int => Int = (x: Int) => x + 1
```

```
scala> inc(10)  
res1: Int = 11
```



# First class function

```
val inc    : Int => Int = (x: Int) => x + 1  
val dec    : Int => Int = (x: Int) => x - 1  
val double: Int => Int = (x: Int) => x * 2
```

```
val list = List(inc, dec, double)
```

```
val map = Map(  
  "foo" -> inc,  
  "bar" -> dec,  
  "fizz" -> double,  
)
```



# First class function: higher order

```
def map(xs: List[Int])(f: Int => Int): List[Int] = ???
```



# First class function: higher order

```
def map(xs: List[Int])(f: Int => Int): List[Int] = ???
```

```
scala> List(1,2,3).map(inc)  
res2: List[Int] = List(2, 3, 4)
```

```
scala> List("hello", "world", "!").takeWhile(_.length > 2)  
res3: List[String] = List(hello, world)
```

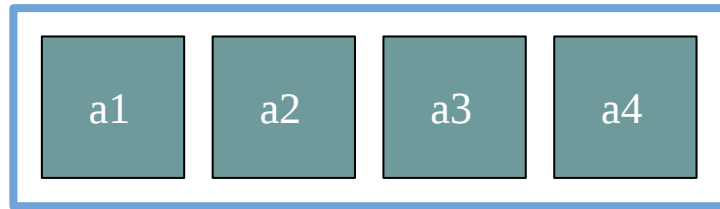


# Exercises 1, 2 and 3a-c

`exercises.function.FunctionExercises.scala`

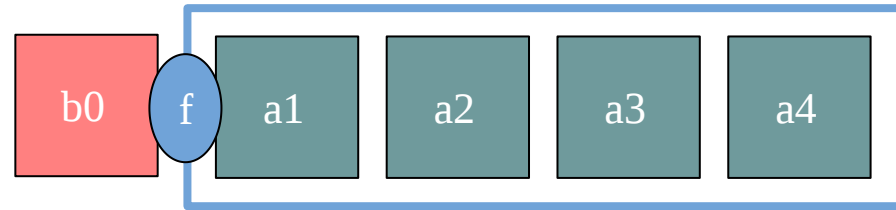


# Folding

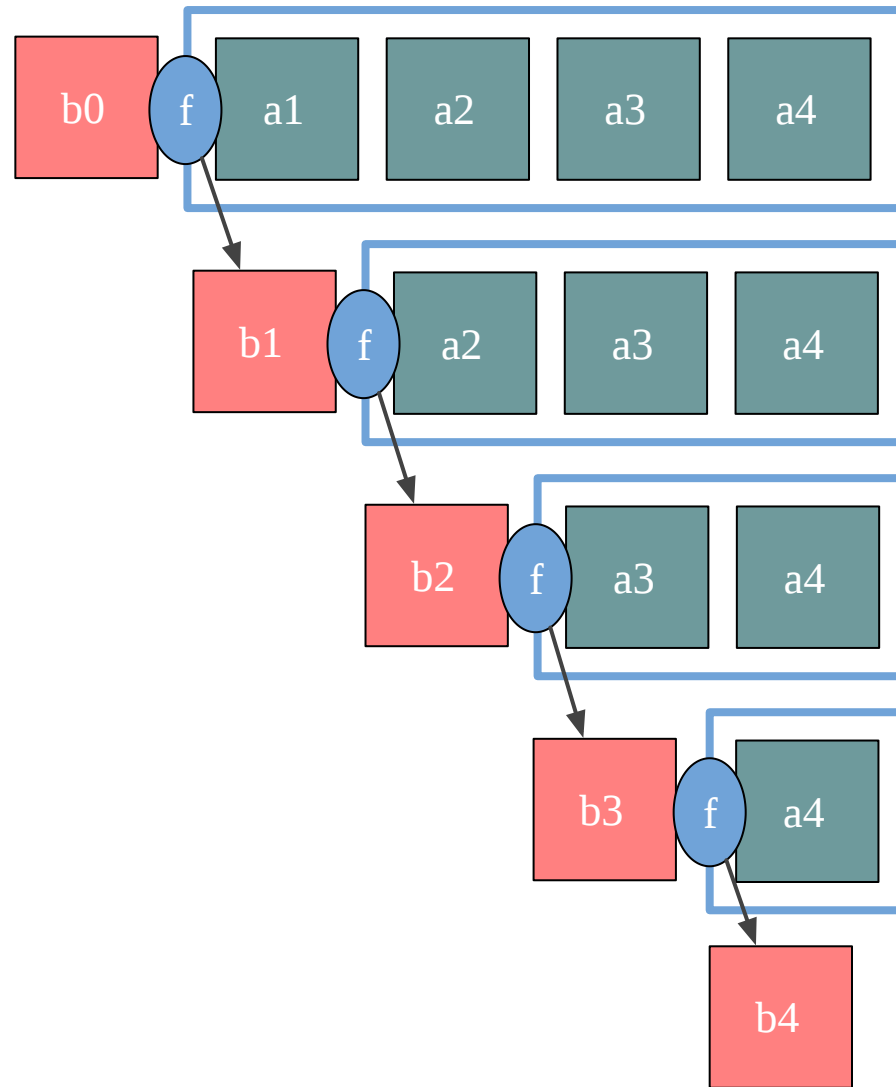




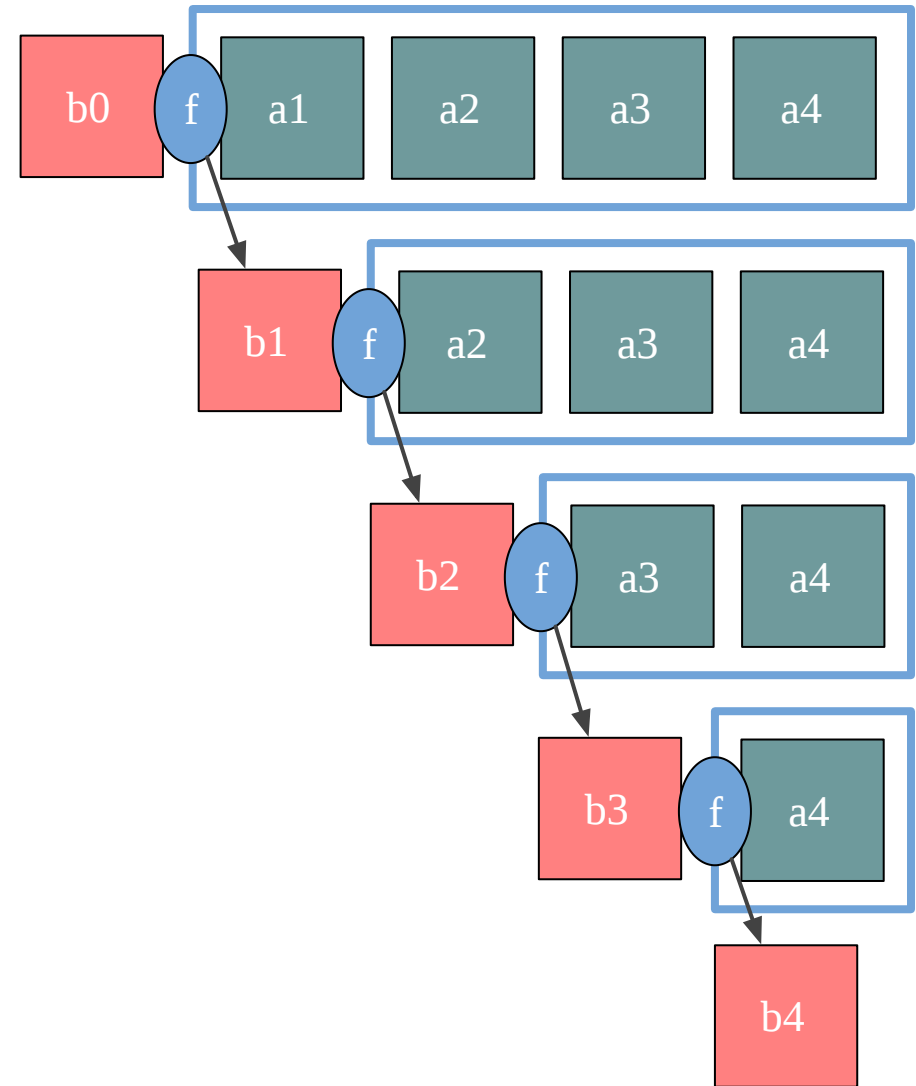
# Fold Left



# Fold Left



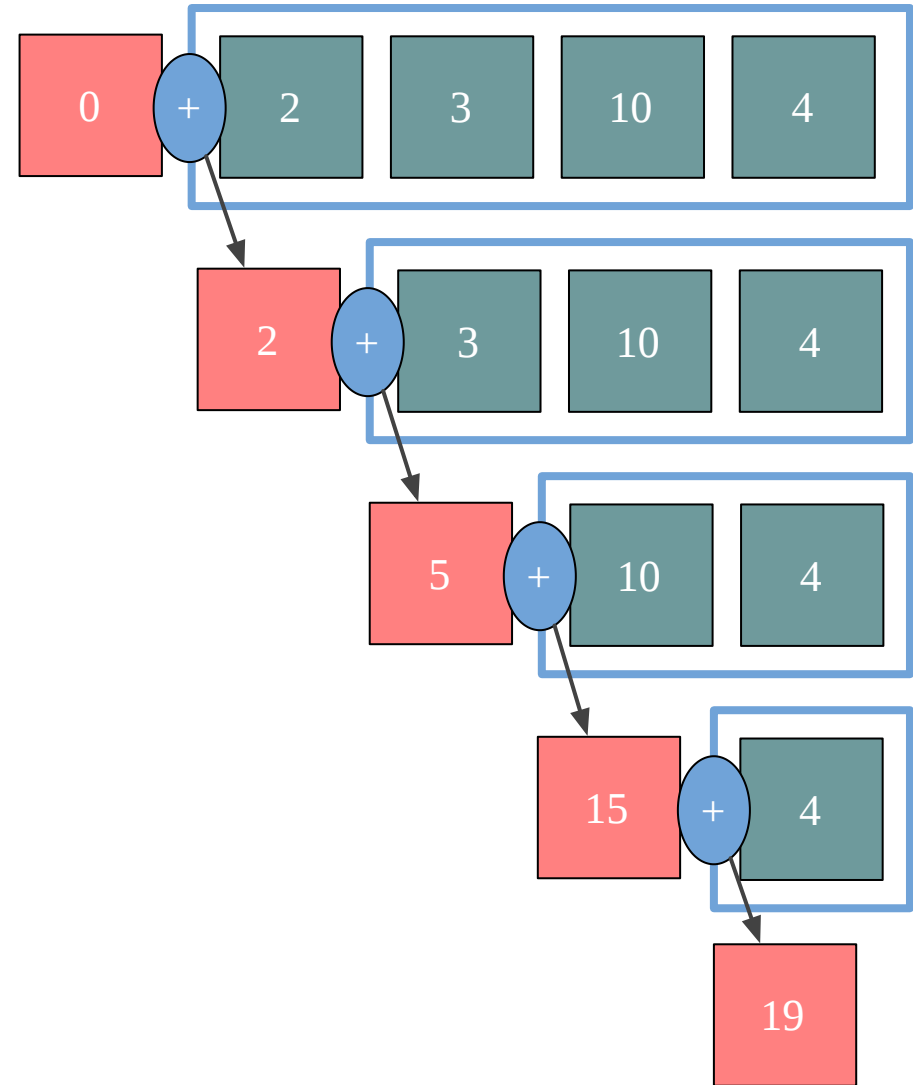
```
def foldLeft[A, B](fa: List[A], b: B)(f: (B, A) => B): B = {  
  var acc = b  
  val it = fa.iterator  
  
  while(it.hasNext) {  
    val current = it.next()  
    acc = f(acc, current)  
  }  
  
  acc  
}
```



# FoldLeft

```
def sum(xs: List[Int]): Int =  
  foldLeft(xs, 0)(_ + _)
```

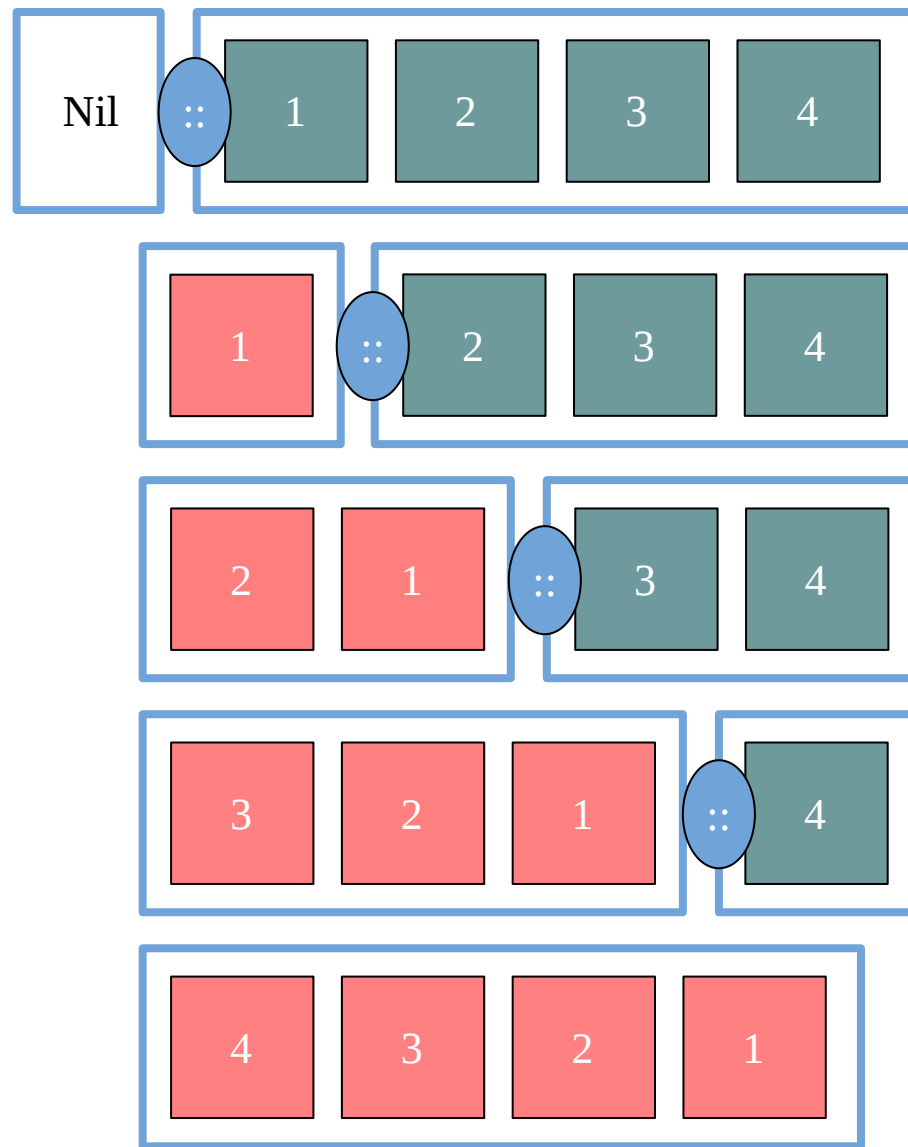
```
scala> sum(List(2,3,10,4))  
res4: Int = 19
```



# FoldLeft

```
def reverse[A](xs: List[A]): List[A] =  
  foldLeft(xs, List.empty[A])((acc, a) => a :: acc)
```

```
scala> reverse(List(1,2,3,4))  
res5: List[Int] = List(4, 3, 2, 1)
```

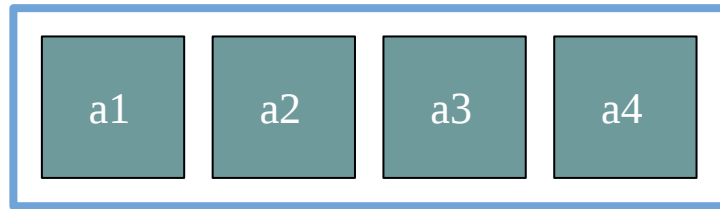


# Finish Exercise 3

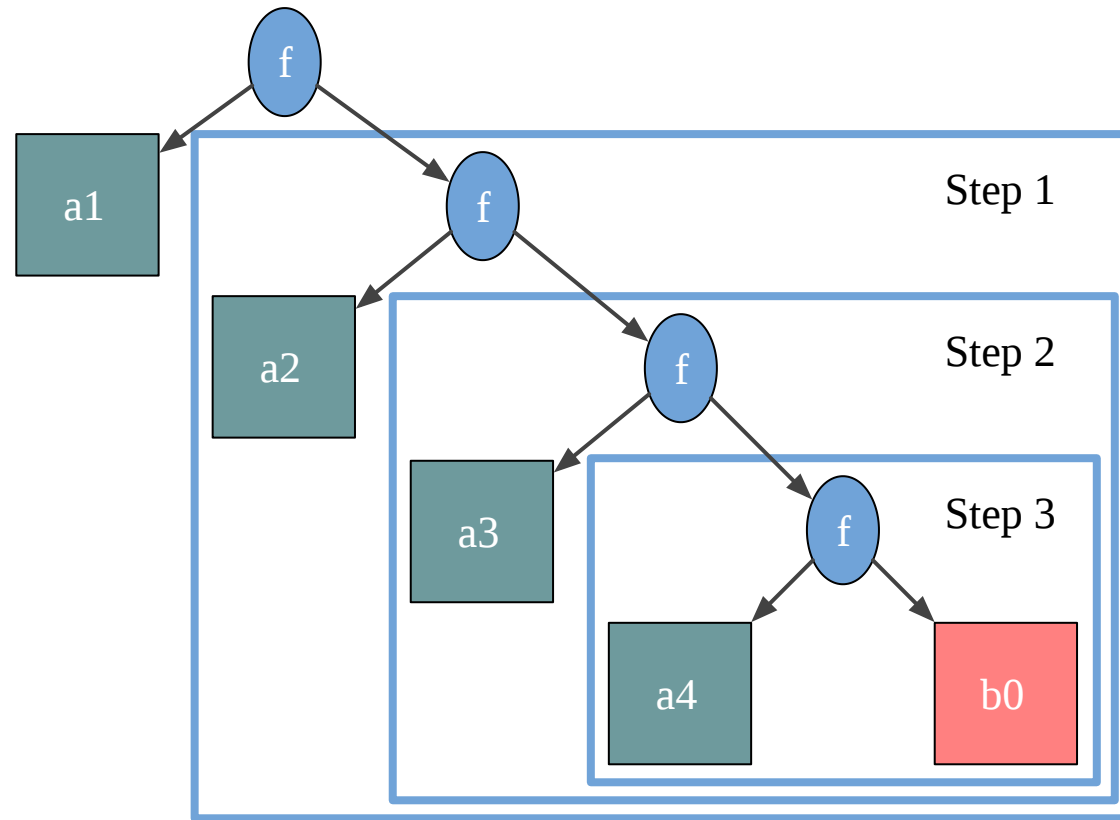
`exercises.function.FunctionExercises.scala`



# Folding

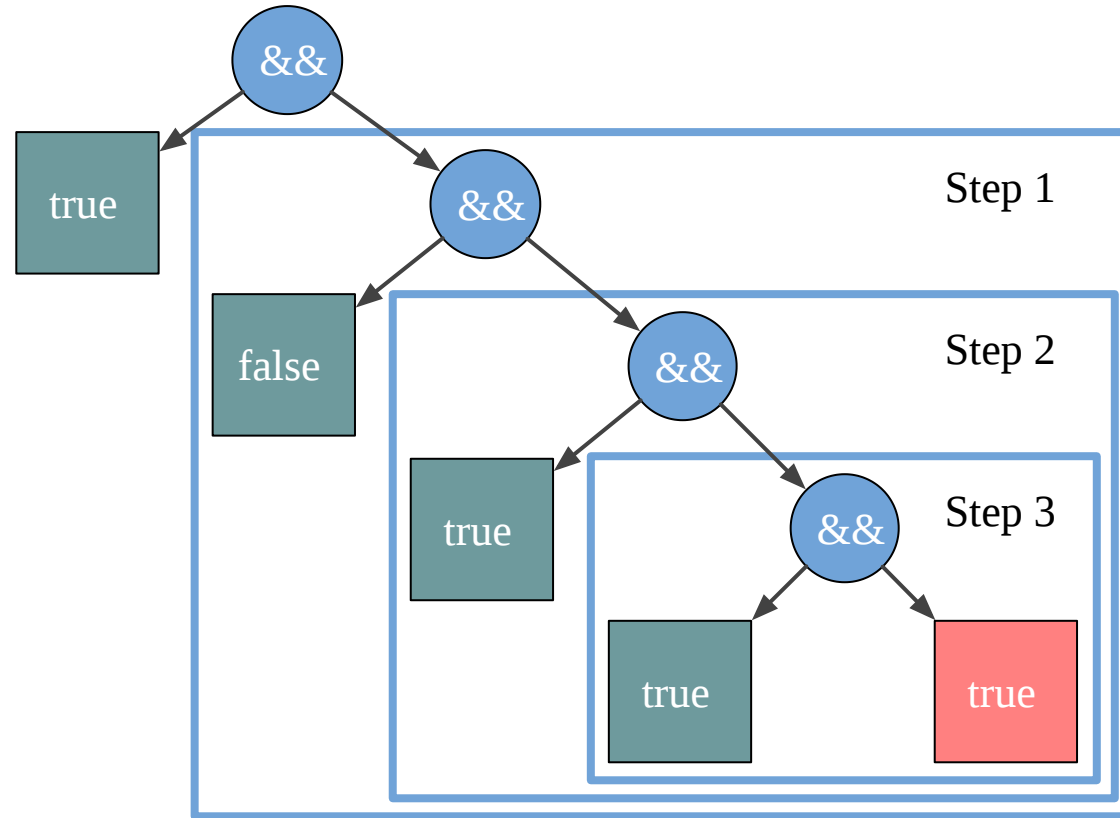


# Fold Right

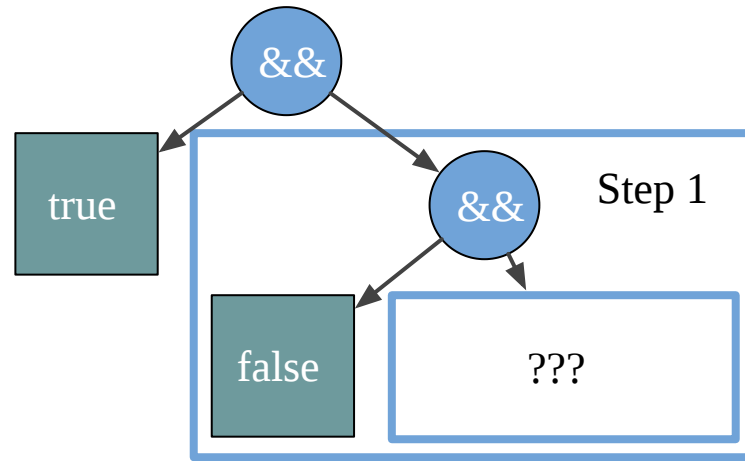




# Fold Right

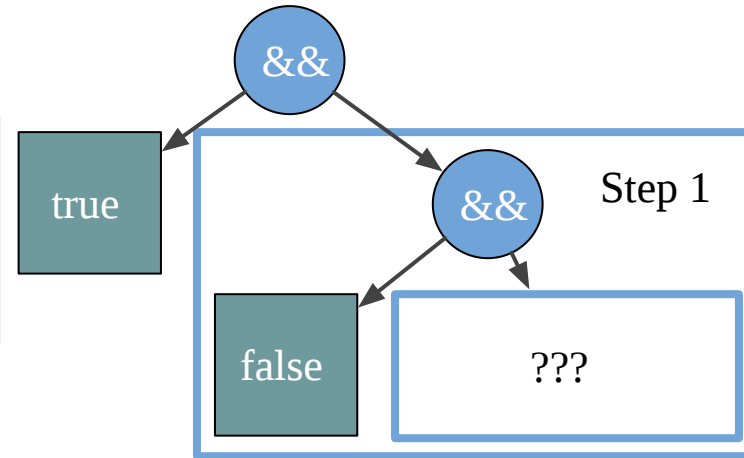


# Fold Right is lazy

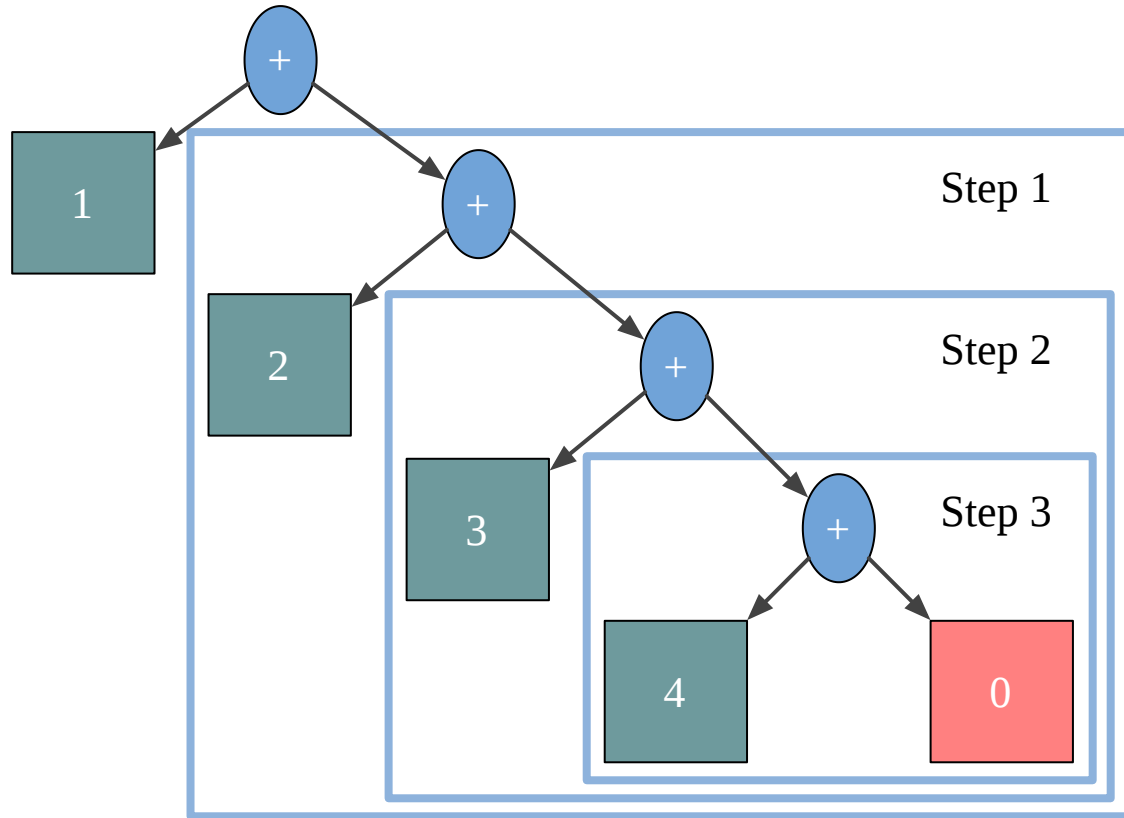


# Fold Right is lazy

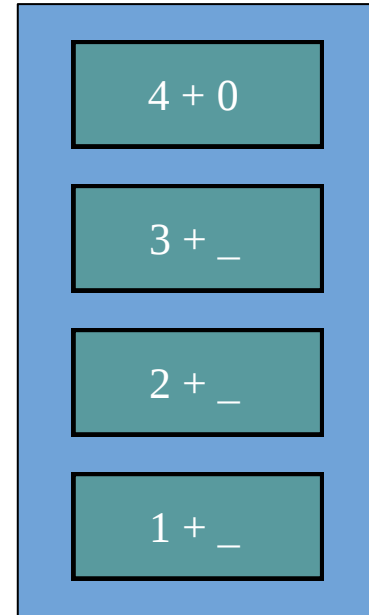
```
def foldRight[A, B](xs: List[A], z: B)(f: (A, => B) => B): B =  
  xs match {  
    case Nil      => z  
    case h :: t => f(h, foldRight(t, z)(f))  
  }
```



# Fold Right is NOT always stack safe



Stack



# Fold Right replaces constructors

```
sealed trait List[A]  
  
case class Nil[A]() extends List[A]  
case class Cons[A](head: A, tail: List[A]) extends List[A]
```

```
scala> val xs: List[Int] = Cons(1, Cons(2, Cons(3, Nil())))  
xs: List[Int] = Cons(1,Cons(2,Cons(3,Nil())))
```



# Fold Right replaces constructors

```
sealed trait List[A]

case class Nil[A]() extends List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
```

```
scala> val xs: List[Int] = Cons(1, Cons(2, Cons(3, Nil())))
xs: List[Int] = Cons(1,Cons(2,Cons(3,Nil())))
```

```
def foldRight[A, B](list: List[A], z: B)(f: (A, => B) => B): B

foldRight(xs, z)(f) == foldRight(Cons(1, Cons(2, Cons(3, Nil()))), z)(f)
                    ==          f  (1, f  (2, f  (3, z  )))
```



# Fold Right replaces constructors

```
sealed trait List[A]

case class Nil[A]() extends List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
```

```
scala> val xs: List[Int] = Cons(1, Cons(2, Cons(3, Nil())))
xs: List[Int] = Cons(1,Cons(2,Cons(3,Nil())))
```

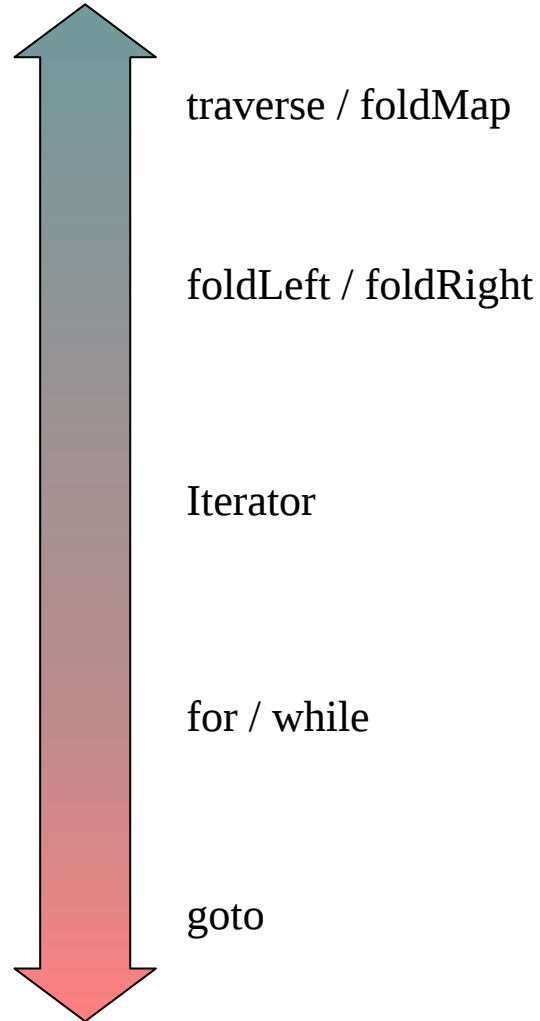
```
def foldRight[A, B](list: List[A], z: B)(f: (A, => B) => B): B

foldRight(xs, z)(f) == foldRight(Cons(1, Cons(2, Cons(3, Nil()))), z)(f)
                    ==          f  (1, f  (2, f  (3, z      )))
```

Exercise: How would you "replace constructors" for Option or Binary Tree?



# Different level of abstractions

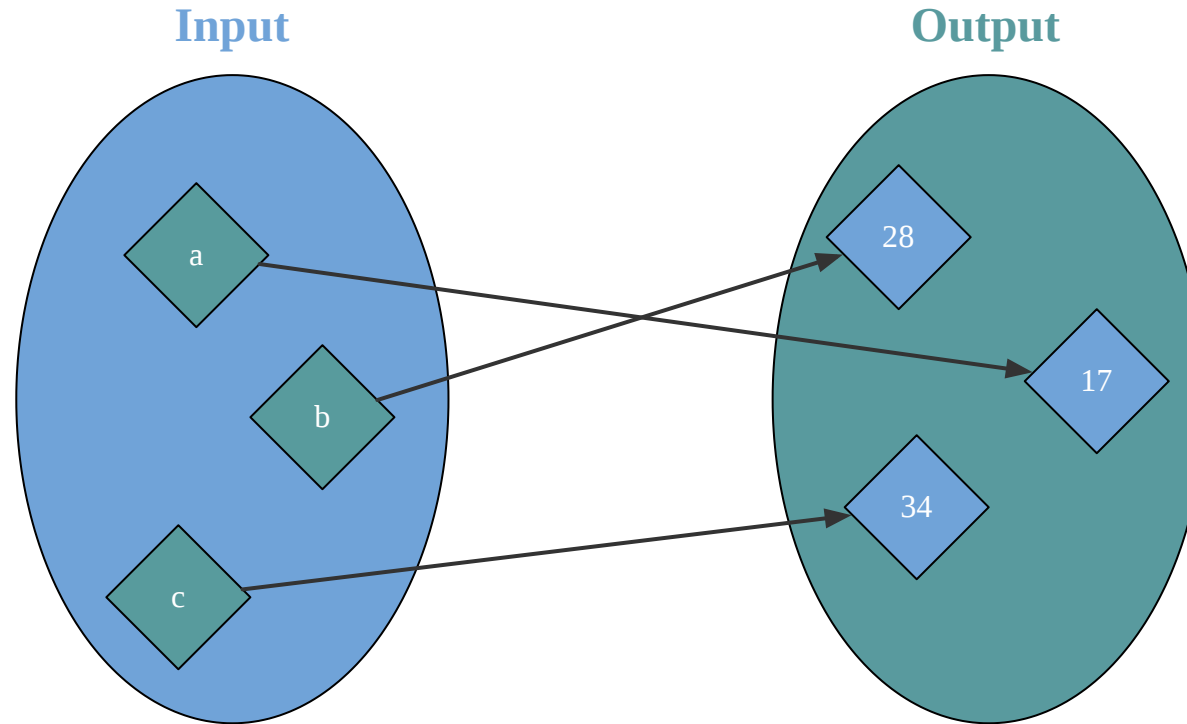




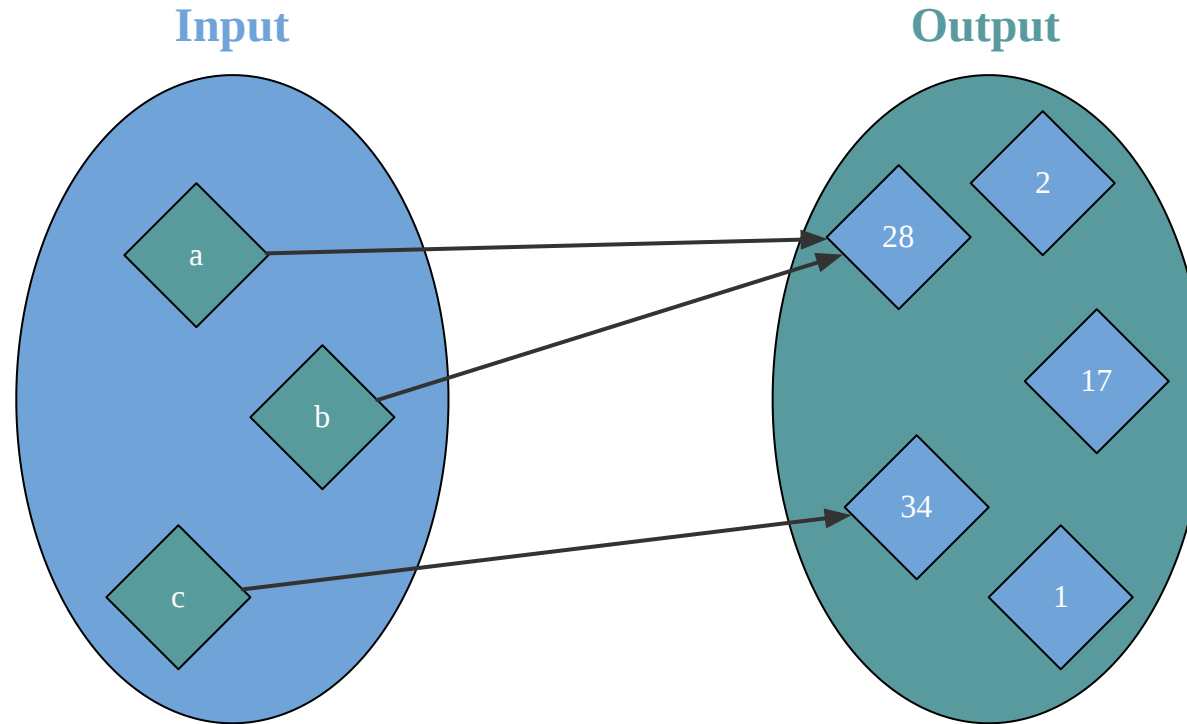
# Pure function



# Pure function is a mapping



# Pure function is a mapping



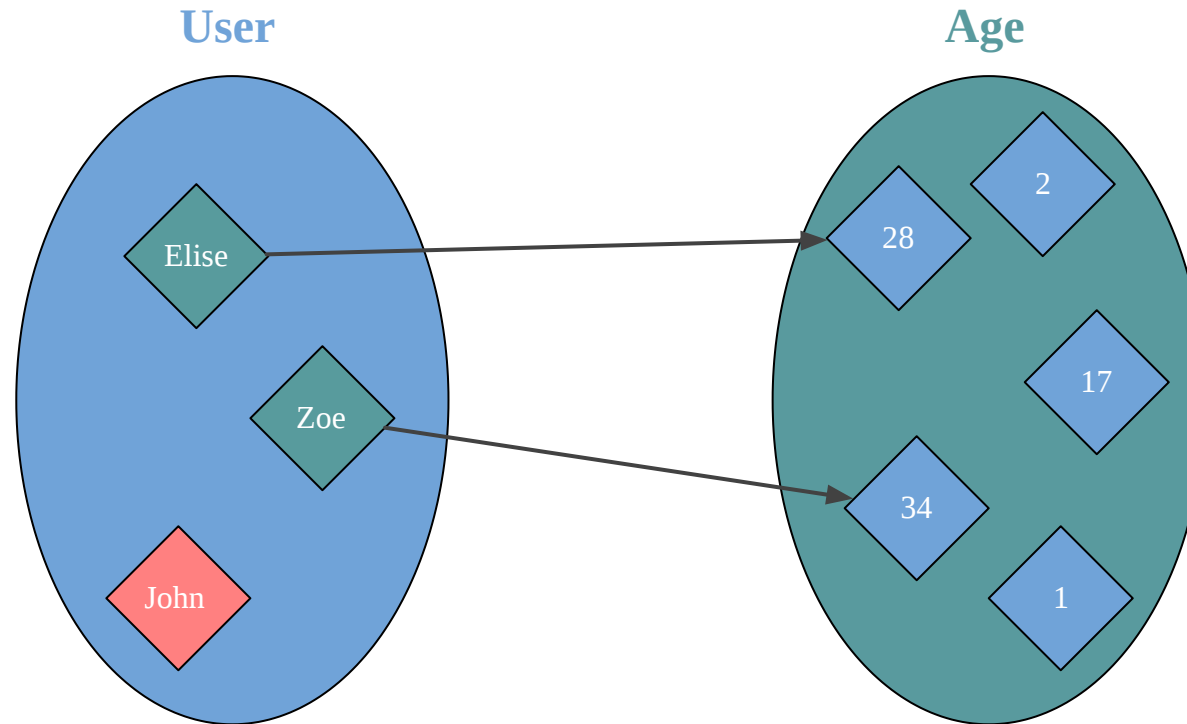
# Programming function

!=

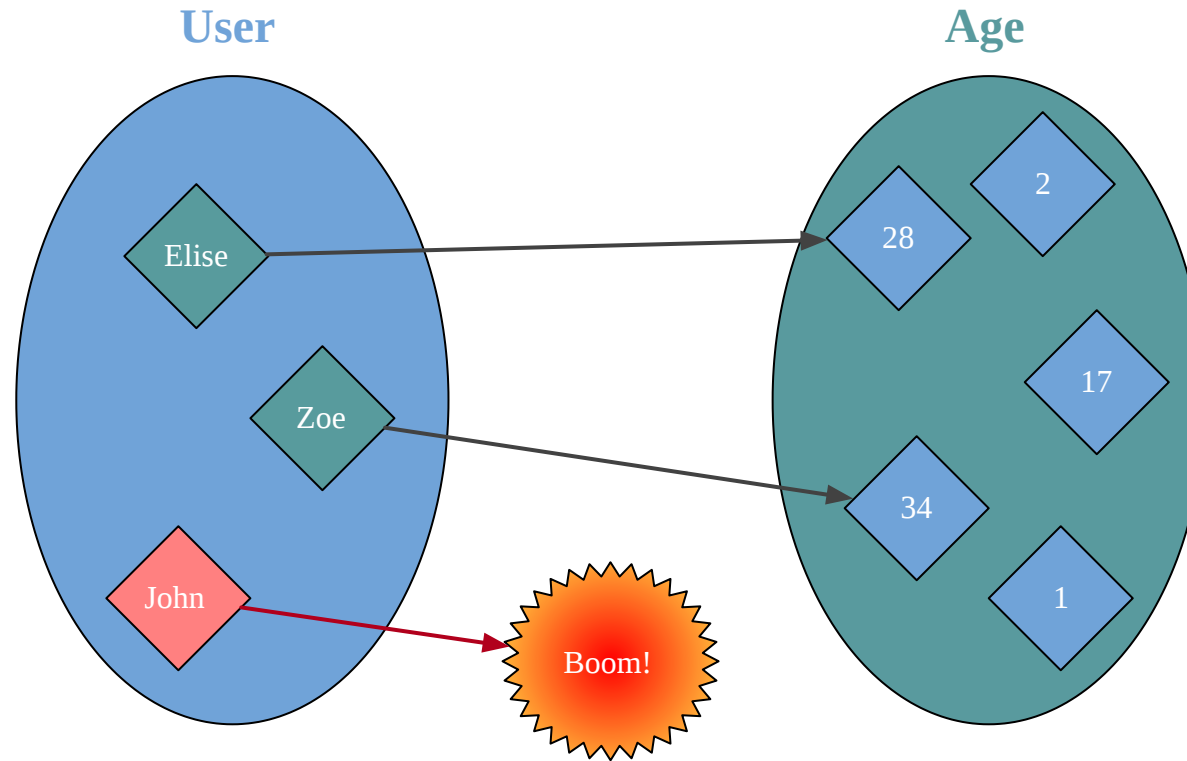
# Pure function



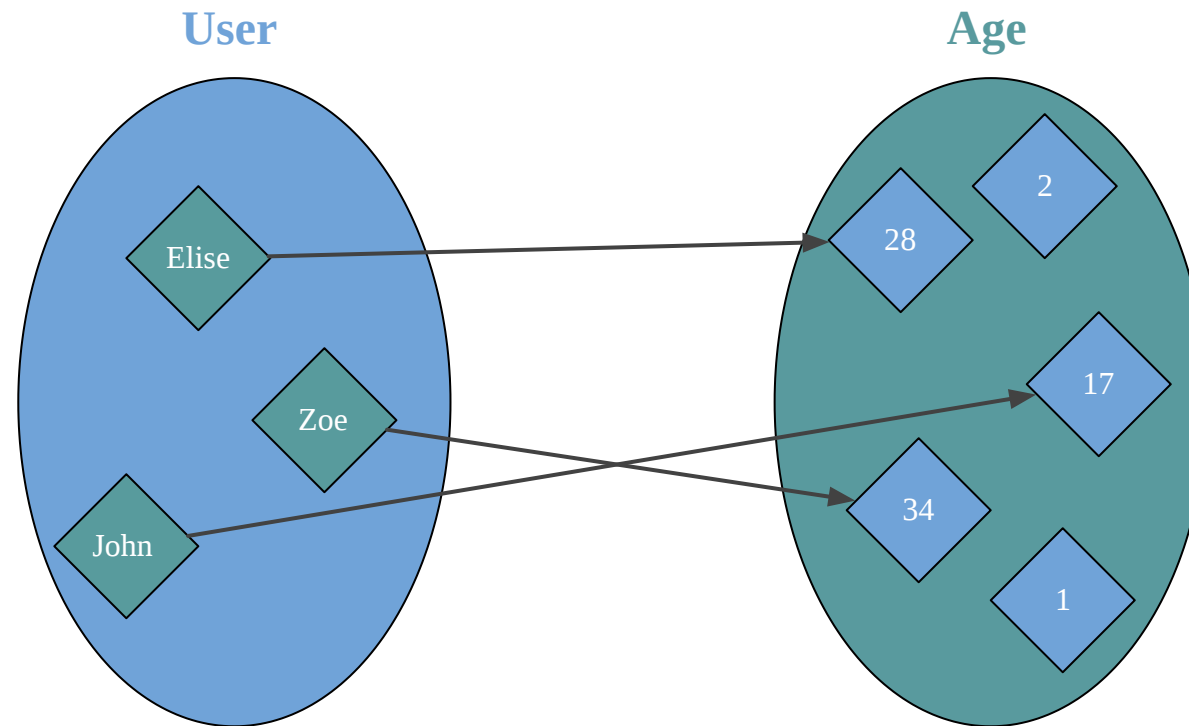
# Partial function



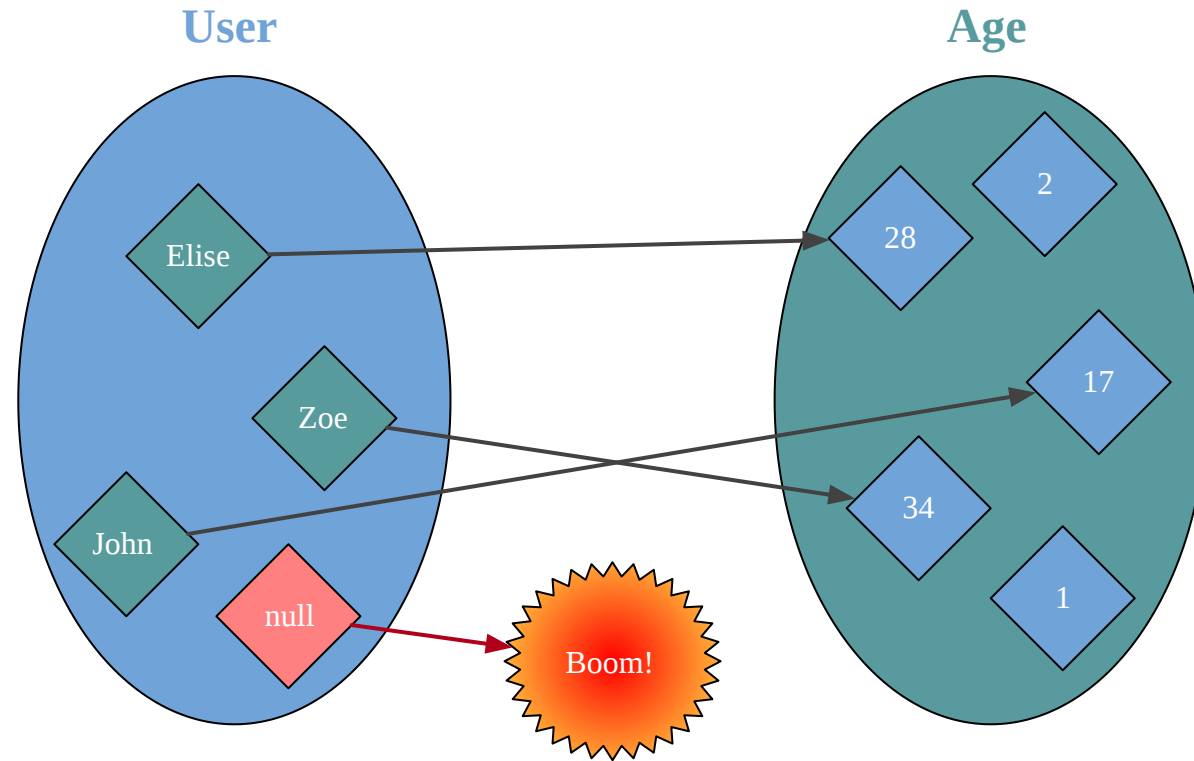
# Partial function



# Null



# Null





# Null handling

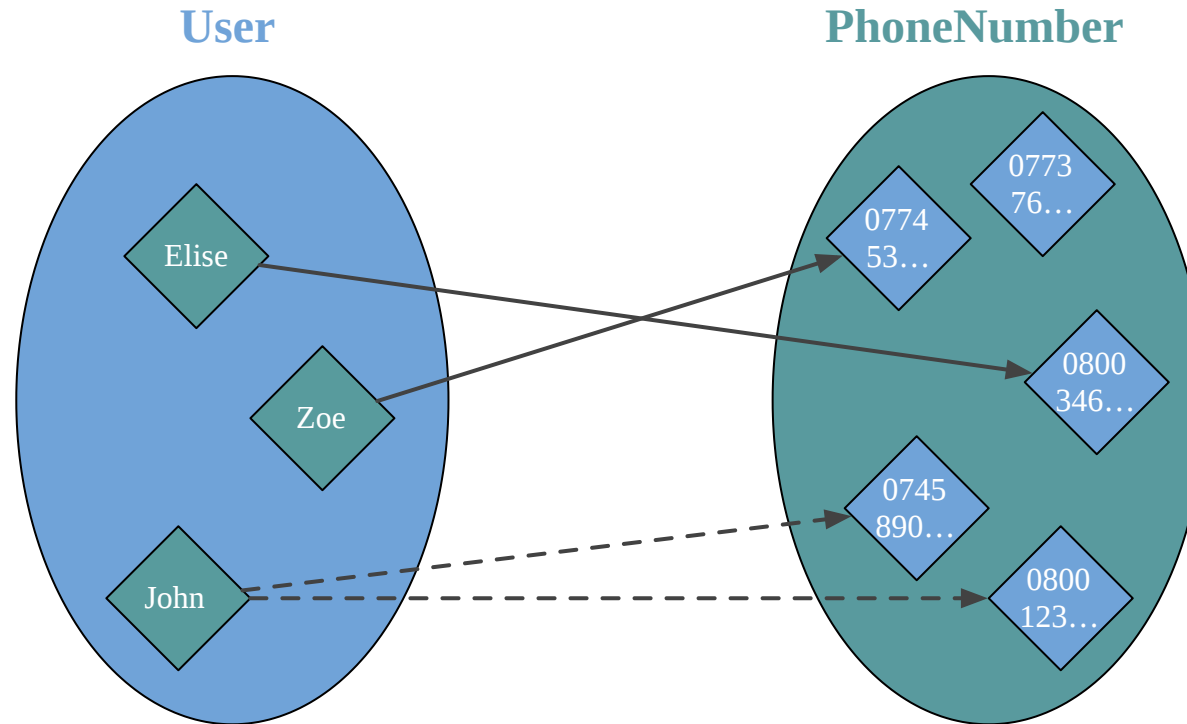
We cannot remove `null` from the language (maybe in 3.0)

We cannot handle it for all functions

So we ignore null: don't return it, don't handle it



# Nondeterministic



# Nondeterministic

```
import scala.util.Random
```

```
scala> Random.nextInt(100)  
res6: Int = 2
```

```
scala> Random.nextInt(100)  
res7: Int = 68
```

```
scala> Random.nextInt(100)  
res8: Int = 56
```

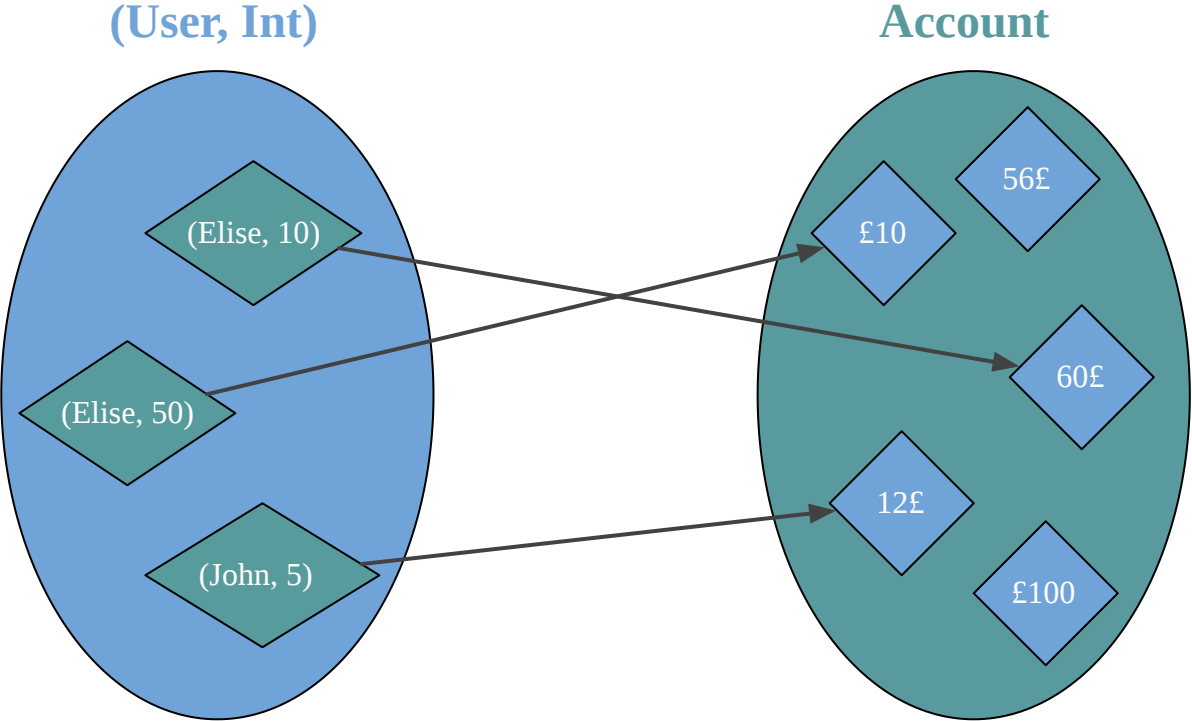
```
import java.time.Instant
```

```
scala> Instant.now()  
res9: java.time.Instant = 2019-09-04T08:45:29.905881Z
```

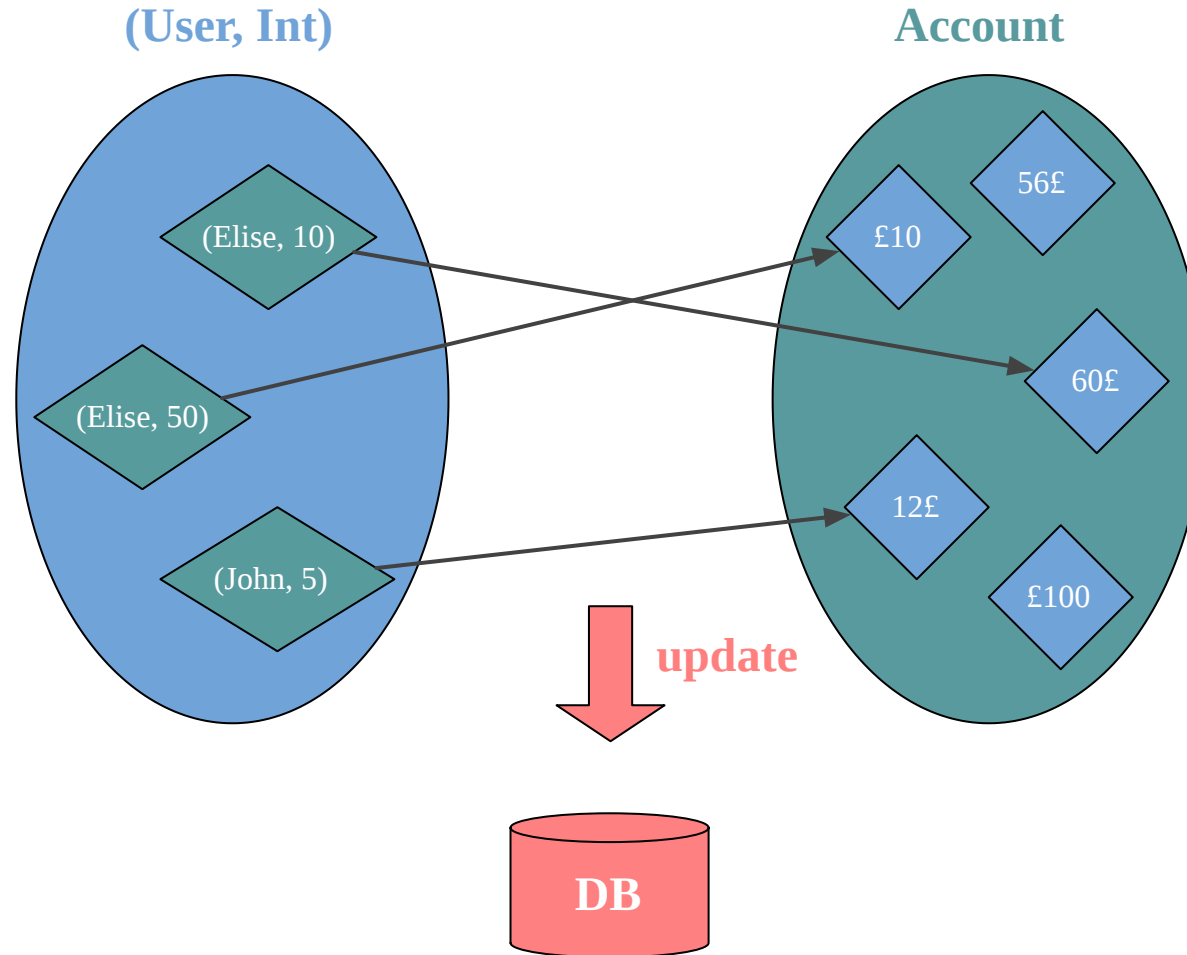
```
scala> Instant.now()  
res10: java.time.Instant = 2019-09-04T08:45:29.982358Z
```



# Side effect



# Side effect



# Side effect

```
def println(message: String): Unit = ...
```

```
scala> val x = println("Hello")  
Hello  
x: Unit = ()
```



# Side effect

```
def println(message: String): Unit = ...
```

```
scala> val x = println("Hello")  
Hello  
x: Unit = ()
```

```
scala> scala.io.Source.fromURL("http://google.com")("ISO-8859-1").take(100).mkString  
res11: String = <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en"><head><meta content
```



# Side effect

```
def println(message: String): Unit = ...
```

```
scala> val x = println("Hello")  
Hello  
x: Unit = ()
```

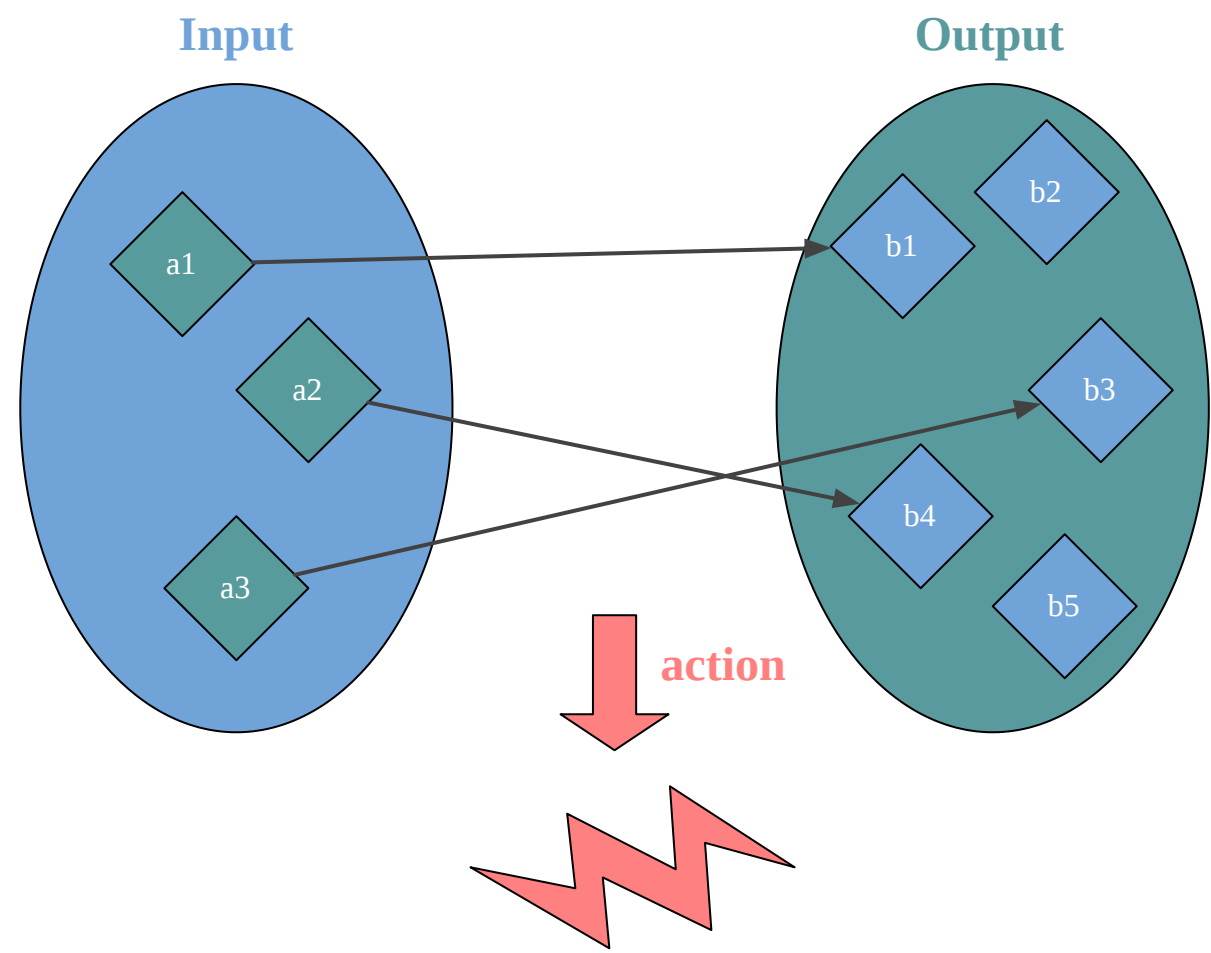
```
scala> scala.io.Source.fromURL("http://google.com")("ISO-8859-1").take(100).mkString  
res11: String = <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en"><head><meta content
```

```
var x: Int = 0  
  
def count(): Int = {  
  x = x + 1  
  x  
}
```





# Side effect



# Reflection

```
def foo[A](a: A): A = a match {  
  case x: Int    => (x + 1).asInstanceOf[A]  
  case x: String => x.reverse.asInstanceOf[A]  
  case _         => a  
}
```

```
scala> foo(5)  
res12: Int = 6
```

```
scala> foo("Hello")  
res13: String = olleH
```

```
scala> foo(true)  
res14: Boolean = true
```



# Reflection is not reliable

```
def foo[A](a: A): Int = a match {  
  case _: List[Int]    => 0  
  case _: List[String] => 1  
  case _               => 2  
}
```

```
scala> foo(List(1,2,3))  
res15: Int = 2
```

```
scala> foo(List("abc"))  
res16: Int = 2
```



# Reflection makes signature unreliable

```
def foo[A](a: A): A = ???
```



# Reflection makes signature unreliable

```
def foo[A](a: A): A = ???
```

## Without Reflection

```
def foo[A](a: A): A = a
```

and without infinite recursion

```
def foo[A](a: A): A = foo(a)
```



## Also apply to Any and non sealed trait

```
def foo(a: Any): Int = ???
```

## Without Reflection

```
def foo[A](a: Any): Int = 5
```



# Pure Function (aka Scalazzi subset)

- deterministic
- total (not partial)
- no mutation
- no exception
- no null
- no side effect
- no reflection



# Exercise 4

`exercises.function.FunctionExercises.scala`





# Why pure function?



# 1. Refactoring



# Refactoring

```
def hello_1(foo: Foo, bar: Bar) = {  
  val x = f(foo)  
  val y = g(bar)  
  h(y)  
  y  
}
```



# Refactoring: remove unused code

```
def hello_1(foo: Foo, bar: Bar) = {  
  val x = f(foo)  
  val y = g(bar)  
  h(y)  
  y  
}
```



# Refactoring: remove unused code

```
def hello_1(foo: Foo, bar: Bar) = {  
  val x = f(foo)  
  val y = g(bar)  
  h(y)  
  y  
}
```

should be equivalent to

```
def hello_2(foo: Foo, bar: Bar): Int =  
  g(bar)
```



# Refactoring: remove unused code

```
def hello_1(foo: Foo, bar: Bar) = {  
  val x = f(foo)  
  val y = g(bar)  
  h(y)  
  y  
}
```

should be equivalent to

```
def hello_2(foo: Foo, bar: Bar): Int =  
  g(bar)
```

## Counter example

```
def f(foo: Foo): Unit = upsertToDb(foo)  
  
def h(id: Int): Unit = globalVar += 1
```



# Refactoring: reorder variables

```
def hello_1(foo: Foo, bar: Bar) = {  
  val x = f(foo)  
  val y = g(bar)  
  h(x, y)  
}
```

```
def hello_2(foo: Foo, bar: Bar): Int = {  
  val y = g(bar)  
  val x = f(foo)  
  h(x, y)  
}
```



# Refactoring: reorder variables

```
def hello_1(foo: Foo, bar: Bar) = {  
  val x = f(foo)  
  val y = g(bar)  
  h(x, y)  
}
```

```
def hello_2(foo: Foo, bar: Bar): Int = {  
  val y = g(bar)  
  val x = f(foo)  
  h(x, y)  
}
```

## Counter example

```
def f(foo: Foo): Unit = print("foo")  
def g(bar: Bar): Unit = print("bar")  
  
hello_1(foo, bar) // print foobar  
hello_2(foo, bar) // print barfoo
```





# Refactoring: extract - inline

```
def hello_extract(foo: Foo, bar: Bar) = {  
  val x = f(foo)  
  val y = g(bar)  
  h(x, y)  
}
```

```
def hello_inline(foo: Foo, bar: Bar) = {  
  h(f(foo), g(bar))  
}
```



# Refactoring: extract - inline

```
def hello_extract(foo: Foo, bar: Bar) = {  
  val x = f(foo)  
  val y = g(bar)  
  h(x, y)  
}
```

```
def hello_inline(foo: Foo, bar: Bar) = {  
  h(f(foo), g(bar))  
}
```

## Counter example

```
def f(foo: Foo): Boolean = false  
  
def g(bar: Bar): Boolean = throw new Exception("Boom!")  
  
def h(b1: Boolean, b2: => Boolean): Boolean = b1 && b2  
  
hello_inline (foo, bar) // false  
hello_extract(foo, bar) // throw Exception
```



# Refactoring: extract - inline

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def doSomethingExpensive(x: Int): Future[Int] =
  Future { ??? }

for {
  x <- doSomethingExpensive(5)
  y <- doSomethingExpensive(8) // sequential, only starts when first Future is complete
                                // will not be called if first Future fails
} yield x + y
```

```
val fx = doSomethingExpensive(5)
val fy = doSomethingExpensive(8) // both Future are started in parallel

for {
  x <- fx
  y <- fy
} yield x + y
```



# Refactoring: de-duplicate

```
def hello_duplicate(foo: Foo) = {  
  val x = f(foo)  
  val y = f(foo)  
  h(x, y)  
}
```

```
def hello_simplified(foo: Foo) = {  
  val x = f(foo)  
  h(x, x)  
}
```



# Refactoring: de-duplicate

```
def hello_duplicate(foo: Foo) = {  
  val x = f(foo)  
  val y = f(foo)  
  h(x, y)  
}
```

```
def hello_simplified(foo: Foo) = {  
  val x = f(foo)  
  h(x, x)  
}
```

## Counter example

```
def f(foo: Foo): Unit = print("foo")  
  
hello_duplicate(foo) // print foofoo  
hello_simplified(foo) // print foo
```



Pure function  
means  
fearless refactoring



## 2. Local reasoning



# Local reasoning

```
def hello(foo: Foo, bar: Bar): Int = {  
  ??? // only depends on foo, bar  
}
```





# Local reasoning

```
class HelloWorld(fizz: Fizz) {  
  val const = 12.3  
  def hello(foo: Foo, bar: Bar): Int = {  
    ??? // only depends on foo, bar, const and fizz  
  }  
}
```



# Local reasoning

```
class HelloWorld(fizz: Fizz) {  
  var secret = null // ☐  
  
  def hello(foo: Foo, bar: Bar): Int = {  
    FarAwayObject.mutableMap += "foo" -> foo // ☐  
    publishMessage>Hello(foo, bar)) // ☐  
    ???  
  }  
}  
  
object FarAwayObject {  
  val mutableMap = ??? // ☐  
}
```



### 3. Easier testing



## 4. Better documentation



# Better documentation

```
def getAge(user: User): Int = ???  
  
def getOrElse[A](fa: Option[A])(orElse: => A): A = ???  
  
def parseJson(x: String): Either[ParsingError, Json] = ???  
  
def void[A](fa: Option[A]): Option[Unit] = ???  
  
def none: Option[Nothing] = ???  
  
def forever[A](fa: IO[A]): IO[Nothing] = ???
```



# 5. Potential compiler optimisations

See Exercise 5

`exercises.function.FunctionExercises.scala`



# What's the catch?



With pure function, you cannot **DO** anything





# Resources and further study

- [Explain List Folds to Yourself](#)
- [Constraints Liberate, Liberties Constrain](#)



## Module 2: Side Effect

