

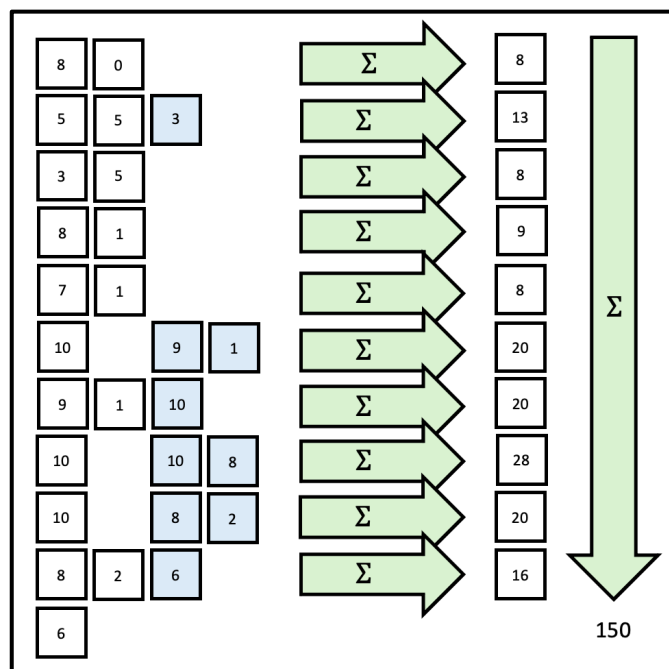


The Bowling Game

From Imperative to Functional Programming

Part 3

λ Functional Programming 



 **Scala**

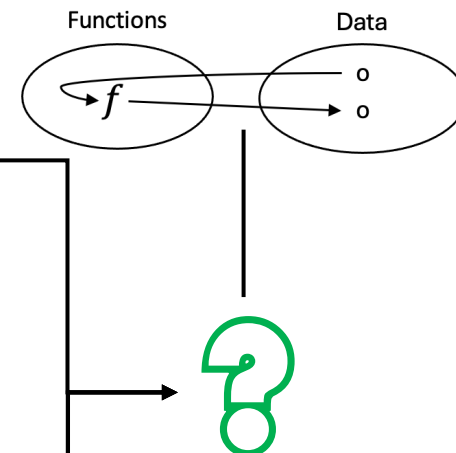


 **Haskell**

 **Clojure**

Monoid [A]	→ combine, empty
Functor [F[_]]	→ map
Foldable [F[_]]	→ fold, foldMap
Nested [F[_], G[_], A]	→ Functor [F] & Functor [G] ⇒ Functor [Nested [F,G,*]] Foldable [F] & Foldable [G] ⇒ Foldable [Nested [F,G,*]]
Monoid a	→ mappend, mempty
Functor f	→ fmap
Foldable f	→ fold, foldMap
Compose f g a	→ Functor f & Functor g ⇒ Functor (Compose f g) Foldable f & Foldable g ⇒ Foldable (Compose f g)

reduce



slides by



@philip_schwarz



<https://fpilluminated.org/>

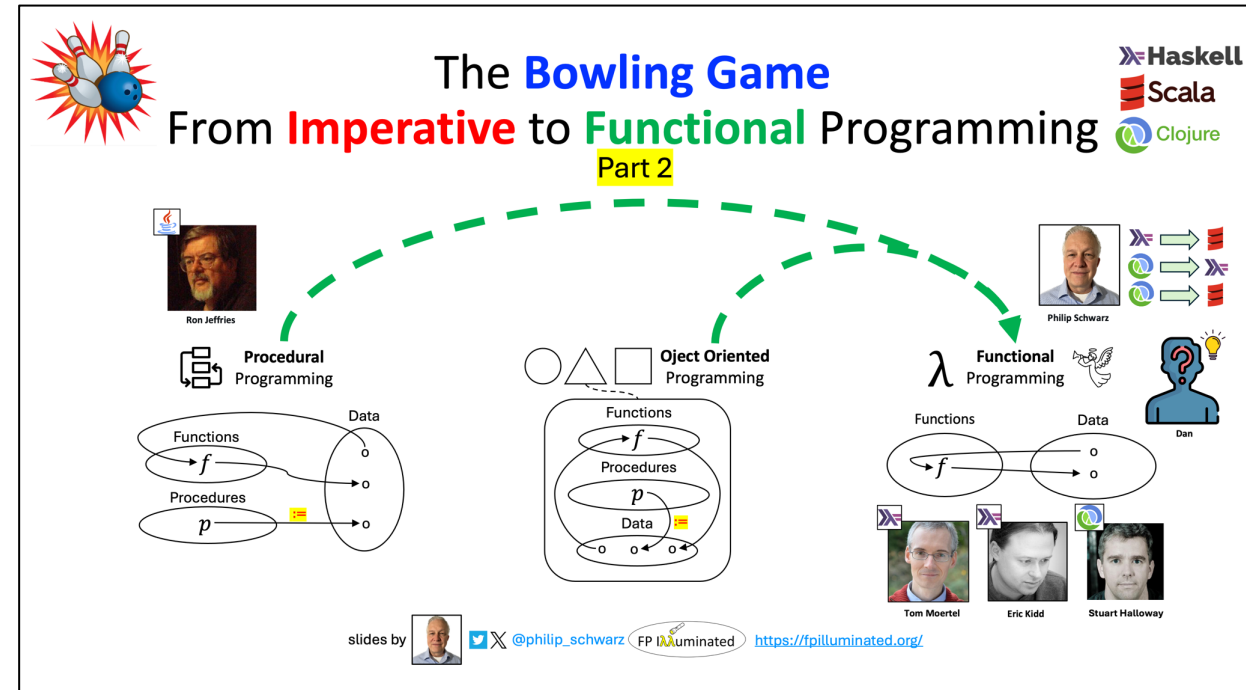
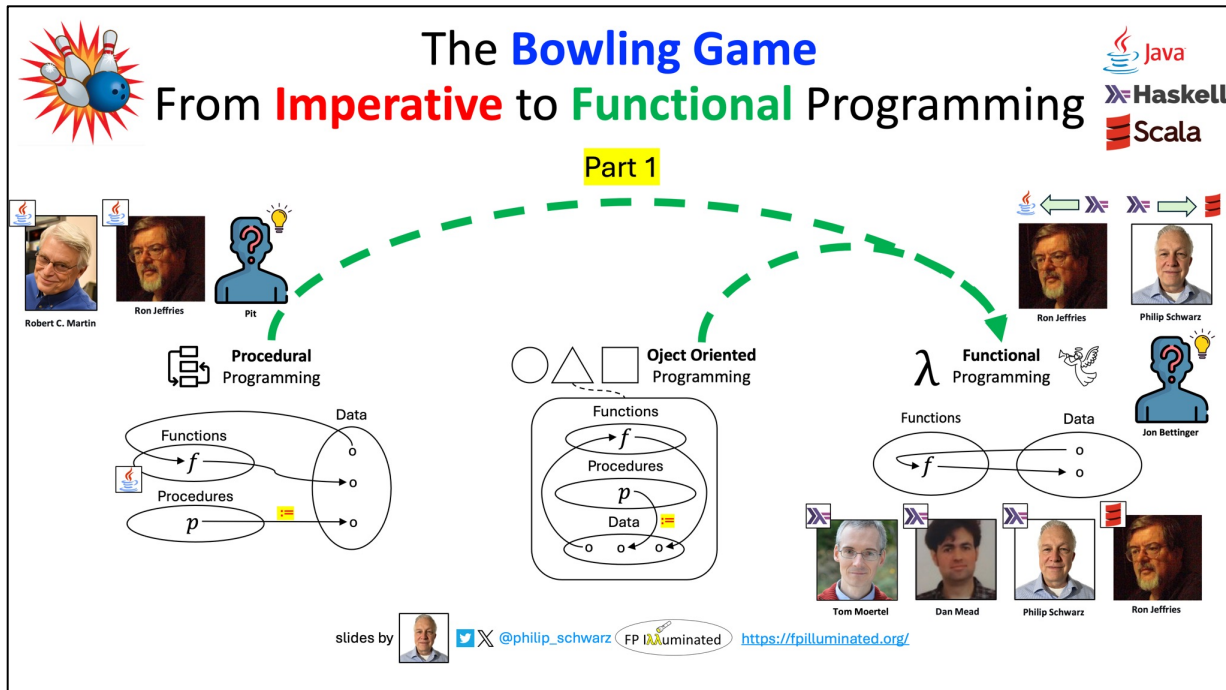
Welcome to **part 3** of this **series** of **decks**.



@philip_schwarz



<https://fpilluminated.org>



Here is the **last bowling program** that we saw in **part 2**.

I think its following **traits** make it a **great version** of the **program**:

- **domain rich**
- **simple functions**
- **good naming**
- **useful comments**



 [@philip_schwarz](#)



Stuart Halloway

```
(ns bowling-game (:use clojure.contrib.seq-utils))

(defn strike? [rolls]
  (= 10 (first rolls)))

(defn spare? [rolls]
  (= 10 (apply + (take 2 rolls))))

(defn balls-to-score
  "How many balls contribute to this frame's score?"
  [rolls]
  (cond
    (strike? rolls) 3
    (spare? rolls) 3
    :else 2))

(defn frame-advance
  "How many rolls should be consumed to advance to the next frame?"
  [rolls]
  (if (strike? rolls) 1 2))

(defn frames
  "Converts a sequence of rolls to a sequence of frames"
  [rolls]
  (when-let [rolls (seq rolls)]
    (lazy-seq (cons (take (balls-to-score rolls) rolls)
                    (frames (drop (frame-advance rolls) rolls))))))

(defn score-frame
  [frame]
  (reduce + frame))

(defn score-game
  "Score a bowling game, passed as a sequence of rolls."
  [rolls]
  (reduce + (map score-frame (take 10 (frames rolls)))))
```

```
(ns bowling-game (:use clojure.contrib.seq-utils))
```

```
(defn strike? [rolls]
  (= 10 (first rolls)))
```

```
(defn spare? [rolls]
  (= 10 (apply + (take 2 rolls))))
```

```
(defn balls-to-score
  "How many balls contribute to this frame's score?"
  [rolls]
  (cond
    (strike? rolls) 3
    (spare? rolls) 3
    :else 2))
```

```
(defn frame-advance
  "How many rolls should be consumed to advance to the next frame?"
  [rolls]
  (if (strike? rolls) 1 2))
```

```
(defn frames
  "Converts a sequence of rolls to a sequence of frames"
  [rolls]
  (when-let [rolls (seq rolls)]
    (lazy-seq (cons (take (balls-to-score rolls) rolls)
                    (frames (drop (frame-advance rolls) rolls))))))
```

```
(defn score-frame
  [frame]
  (reduce + frame))
```

```
(defn score-game
  "Score a bowling game, passed as a sequence of rolls."
  [rolls]
  (reduce + (map score-frame (take 10 (frames rolls)))))
```



Stuart Halloway



Here is the same **Clojure** program again, and next to it, its **Scala translation**, which we also saw in **part two**. To ease comparison, the **Scala program** has here been **enhanced** with the same **comments** as the **Clojure** one.

```
def isStrike(rolls: List[Int]): Boolean =
  rolls.take(1).sum == 10
```

```
def isSpare(rolls: List[Int]): Boolean =
  rolls.take(2).sum == 10
```

```
// How many balls contribute to this frame's score?
def ballsToScore(rolls: List[Int]): Int =
  if isStrike(rolls) || isSpare(rolls) then 3 else 2
```

```
// How many rolls should be consumed to advance to the next frame?
def frameAdvance(rolls: List[Int]): Int =
  if isStrike(rolls) then 1 else 2
```

```
// Converts a sequence of rolls to a sequence of frames
def frames(rolls: List[Int]): LazyList[List[Int]] =
  rolls.take(ballsToScore(rolls))
    #:: frames(rolls.drop(frameAdvance(rolls)))
```

```
def scoreFrame(frame: List[Int]): Int =
  frame.sum
```

```
// Score a bowling game, passed as a sequence of rolls.
def scoreGame(rolls: List[Int]): Int =
  frames(rolls).take(10).map(scoreFrame).sum
```



```
def isStrike(rolls: List[Int]): Boolean =
  rolls.take(1).sum == 10

def isSpare(rolls: List[Int]): Boolean =
  rolls.take(2).sum == 10

def ballsToScore(rolls: List[Int]): Int =
  if isStrike(rolls) || isSpare(rolls) then 3 else 2

def frameAdvance(rolls: List[Int]): Int =
  if isStrike(rolls) then 1 else 2

def frames(rolls: List[Int]): LazyList[List[Int]] =
  rolls.take(ballsToScore(rolls))
  #:: frames(rolls.drop(frameAdvance(rolls)))

def scoreFrame(frame: List[Int]): Int =
  frame.sum

def scoreGame(rolls: List[Int]): Int =
  frames(rolls).take(10).map(scoreFrame).sum
```

The **scoreGame** function currently takes a list of **integer numbers** called **rolls**.

Each such **number** is the **number** of **pins knocked down** by one **roll** of the **ball**, i.e. it is the **score** of the **roll**.

Let's introduce a couple of **types** to model the **score** of a **roll**:

```
type PinCount = Int
type RollScore = PinCount
```

Let's also rename **rolls** parameters to **rollScores**, and change their **type** from **List[Int]** to **List[RollScore]**.

See next slide for the changes.





	<>	type PinCount = Int type RollScore = PinCount
def isStrike(rolls: List[Int]): Boolean = rolls.take(1).sum == 10		def isStrike(rollScores: List[RollScore]): Boolean = rollScores.take(1).sum == 10
	=	
def isSpare(rolls: List[Int]): Boolean = rolls.take(2).sum == 10	<>	def isSpare(rollScores: List[RollScore]): Boolean = rollScores.take(2).sum == 10
	=	
def ballsToScore(rolls: List[Int]): Int = if isStrike(rolls) isSpare(rolls) then 3 else 2	<>	def ballsToScore(rollScores: List[RollScore]): Int = if isStrike(rollScores) isSpare(rollScores) then 3 else 2
	=	
def frameAdvance(rolls: List[Int]): Int = if isStrike(rolls) then 1 else 2	<>	def frameAdvance(rollScores: List[RollScore]): Int = if isStrike(rollScores) then 1 else 2
	=	
def frames(rolls: List[Int]): LazyList[List[Int]] = rolls.take(ballsToScore(rolls)) #:: frames(rolls.drop(frameAdvance(rolls)))	<>	def frames(rollScores: List[RollScore]): LazyList[List[RollScore]] = rollScores.take(ballsToScore(rollScores)) #:: frames(rollScores.drop(frameAdvance(rollScores)))
	=	
def scoreFrame(frame: List[Int]): Int = frame.sum	<>	def scoreFrame(frame: List[RollScore]): Int = frame.sum
	=	
def scoreGame(rolls: List[Int]): Int = frames(rolls).take(10).map(scoreFrame).sum	<>	def scoreGame(rollScores: List[RollScore]): Int = frames(rollScores).take(10).map(scoreFrame).sum

```
type PinCount = Int
type RollScore = PinCount
```

```
def isStrike(rollScores: List[RollScore]): Boolean =
  rollScores.take(1).sum == 10

def isSpare(rollScores: List[RollScore]): Boolean =
  rollScores.take(2).sum == 10

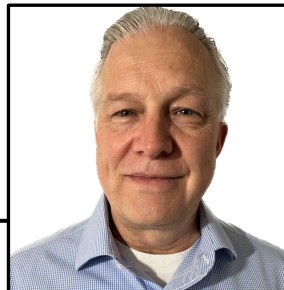
def ballsToScore(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) || isSpare(rollScores) then 3 else 2

def frameAdvance(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) then 1 else 2

def frames(rollScores : List[RollScore]): LazyList[List[RollScore]] =
  rollScores.take(ballsToScore(rollScores))
  #:: frames(rollScores.drop(frameAdvance(rollScores)))

def scoreFrame(frame: List[RollScore]): Int =
  frame.sum

def scoreGame(rollScores: List[RollScore]): Int =
  frames(rollScores).take(10).map(scoreFrame).sum
```



The **frames function** is not really creating **frames**, because a **frame** consists of either **one** or **two rolls**, whereas **frames** returns a **list** containing the **scores** of the **frame's rolls** and the **scores** of the **frame's bonus roll(s)**. i.e. the **frames function** can be said to return **frame scores** (lists of **roll scores**).

Let's make the following changes (shown on the next slide):

- Introduce a **type** indicating that the **score** of a **frame** is determined by the **scores** of a number of **rolls**.

```
type FrameScore = List[RollScore]
```

- **Rename** the **frames function** to **frameScores**, and change its **return type** from **LazyList[List[RollScore]]** to **LazyList[FrameScore]**.
- **Rename** the type of the **parameter** of the **scoreFrame function** to indicate that the function is operating on **FrameScore**.
- Introduce an unused yet explanatory **type** indicating that the **score** of a **game** is determined by the **scores** of a number of **frames**:

```
type GameScore = List[FrameScore]
```



type PinCount = Int type RollScore = PinCount	=	type PinCount = Int type RollScore = PinCount
	->	type FrameScore = List[RollScore] type GameScore = LazyList[FrameScore]
def isStrike(rollScores: List[RollScore]): Boolean = rollScores.take(1).sum == 10 def isSpare(rollScores: List[RollScore]): Boolean = rollScores.take(2).sum == 10 def ballsToScore(rollScores: List[RollScore]): Int = if isStrike(rollScores) isSpare(rollScores) then 3 else 2 def frameAdvance(rollScores: List[RollScore]): Int = if isStrike(rollScores) then 1 else 2	=	def isStrike(rollScores: List[RollScore]): Boolean = rollScores.take(1).sum == 10 def isSpare(rollScores: List[RollScore]): Boolean = rollScores.take(2).sum == 10 def ballsToScore(rollScores: List[RollScore]): Int = if isStrike(rollScores) isSpare(rollScores) then 3 else 2 def frameAdvance(rollScores: List[RollScore]): Int = if isStrike(rollScores) then 1 else 2
def frames(rollScores: List[RollScore]): LazyList[List[RollScore]] = rollScores.take(ballsToScore(rollScores)) #:: frames(rollScores.drop(frameAdvance(rollScores)))	<>	def frameScores(rollScores: List[RollScore]): LazyList[FrameScore] = rollScores.take(ballsToScore(rollScores)) #:: frameScores(rollScores.drop(frameAdvance(rollScores)))
	=	
def scoreFrame(frame: List[RollScore]): Int = frame.sum	<>	def scoreFrame(frame: FrameScore): Int = frame.sum
	=	
def scoreGame(rollScores: List[RollScore]): Int = frames(rollScores).take(10).map(scoreFrame).sum	<>	def scoreGame(rollScores: List[RollScore]): Int = frameScores(rollScores).take(10).map(scoreFrame).sum

```
type PinCount    = Int
type RollScore   = PinCount
type FrameScore  = List[RollScore]
type GameScore   = List[FrameScore]
```

```
def isStrike(rollScores: List[RollScore]): Boolean =
  rollScores.take(1).sum == 10

def isSpare(rollScores: List[RollScore]): Boolean =
  rollScores.take(2).sum == 10

def ballsToScore(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) || isSpare(rollScores) then 3 else 2

def frameAdvance(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) then 1 else 2

def frameScores(rollScores : List[RollScore]): LazyList[FrameScore] =
  rollScores.take(ballsToScore(rollScores))
  #:: frameScores(rollScores.drop(frameAdvance(rollScores)))

def scoreFrame(frame: FrameScore): Int =
  frame.sum

def scoreGame(rollScores: List[RollScore]): Int =
  frameScores(rollScores).take(10).map(scoreFrame).sum
```

The **scoreFrame** function now has a **split personality**: on the one hand it is **scoring** a **frame**, but on the other it is **summing** (!?) a **frame score**. Let's inline it and see if doing so creates some **refactoring opportunities**.

See the next slide for the changes.





<pre>type PinCount = Int type RollScore = PinCount type FrameScore = List[RollScore] type GameScore = LazyList[FrameScore] def isStrike(rollScores: List[RollScore]): Boolean = rollScores.take(1).sum == 10 def isSpare(rollScores: List[RollScore]): Boolean = rollScores.take(2).sum == 10 def ballsToScore(rollScores: List[RollScore]): Int = if isStrike(rollScores) isSpare(rollScores) then 3 else 2 def frameAdvance(rollScores: List[RollScore]): Int = if isStrike(rollScores) then 1 else 2 def frameScores(rollScores: List[RollScore]): LazyList[FrameScore] = rollScores.take(ballsToScore(rollScores)) #:: frameScores(rollScores.drop(frameAdvance(rollScores)))</pre>	=	<pre>type PinCount = Int type RollScore = PinCount type FrameScore = List[RollScore] type GameScore = LazyList[FrameScore] def isStrike(rollScores: List[RollScore]): Boolean = rollScores.take(1).sum == 10 def isSpare(rollScores: List[RollScore]): Boolean = rollScores.take(2).sum == 10 def ballsToScore(rollScores: List[RollScore]): Int = if isStrike(rollScores) isSpare(rollScores) then 3 else 2 def frameAdvance(rollScores: List[RollScore]): Int = if isStrike(rollScores) then 1 else 2 def frameScores(rollScores: List[RollScore]): LazyList[FrameScore] = rollScores.take(ballsToScore(rollScores)) #:: frameScores(rollScores.drop(frameAdvance(rollScores)))</pre>
<pre>def scoreFrame(frame: FrameScore): Int = frame.sum</pre>	+−	
<pre>def scoreGame(rollScores: List[RollScore]): Int = frameScores(rollScores).take(10).map(scoreFrame).sum</pre>	=	<pre>def scoreGame(rollScores: List[RollScore]): Int = frameScores(rollScores).take(10).map(_.sum).sum</pre>
	<>	

```
type PinCount    = Int
type RollScore   = PinCount
type FrameScore  = List[RollScore]
type GameScore   = List[FrameScore]
```

```
def isStrike(rollScores: List[RollScore]): Boolean =
  rollScores.take(1).sum == 10

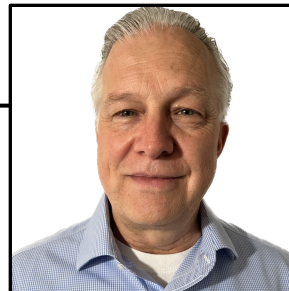
def isSpare(rollScores: List[RollScore]): Boolean =
  rollScores.take(2).sum == 10

def ballsToScore(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) || isSpare(rollScores) then 3 else 2

def frameAdvance(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) then 1 else 2

def frameScores(rollScores : List[RollScore]): LazyList[FrameScore] =
  rollScores.take(ballsToScore(rollScores))
  #:: frameScores(rollScores.drop(frameAdvance(rollScores)))

def scoreGame(rollScores: List[RollScore]): Int =
  frameScores(rollScores).take(10).map(_.sum).sum
```



The **scoreGame** function first turns **roll scores** into **frame scores**, then takes the **first ten frame scores**, and finally executes the logic necessary to **fold** such **frame scores** into an **integer game score**.

Let's **extract** that **logic** into an **extension function** that turns a **LazyList[FrameScore]** into an **integer score**.

To avoid some **compilation errors** down the line (due to **clashes** with existing **standard library functions** and/or **cats functions**), let's call the new function **folded** rather than **fold**, and since the **logic** of such a **function** is **generic**, let's make the type of the **function's parameter more generic**, by repeatedly **substituting domain types** with **more generic types**.

```
LazyList[FrameScore]
  ↓
LazyList[List[RollScore]]
  ↓
LazyList[List[PinCount]]
  ↓
LazyList[List[Int]]
```

See the next slide for the above changes.



<pre>type PinCount = Int type RollScore = PinCount type FrameScore = List[RollScore] type GameScore = LazyList[FrameScore] def isStrike(rollScores: List[RollScore]): Boolean = rollScores.take(1).sum == 10 def isSpare(rollScores: List[RollScore]): Boolean = rollScores.take(2).sum == 10 def ballsToScore(rollScores: List[RollScore]): Int = if isStrike(rollScores) isSpare(rollScores) then 3 else 2 def frameAdvance(rollScores: List[RollScore]): Int = if isStrike(rollScores) then 1 else 2 def frameScores(rollScores: List[RollScore]): LazyList[FrameScore] = rollScores.take(ballsToScore(rollScores)) #:: frameScores(rollScores.drop(frameAdvance(rollScores)))</pre>	=	<pre>type PinCount = Int type RollScore = PinCount type FrameScore = List[RollScore] type GameScore = LazyList[FrameScore] def isStrike(rollScores: List[RollScore]): Boolean = rollScores.take(1).sum == 10 def isSpare(rollScores: List[RollScore]): Boolean = rollScores.take(2).sum == 10 def ballsToScore(rollScores: List[RollScore]): Int = if isStrike(rollScores) isSpare(rollScores) then 3 else 2 def frameAdvance(rollScores: List[RollScore]): Int = if isStrike(rollScores) then 1 else 2 def frameScores(rollScores: List[RollScore]): LazyList[FrameScore] = rollScores.take(ballsToScore(rollScores)) #:: frameScores(rollScores.drop(frameAdvance(rollScores)))</pre>
	-+	<pre>extension (xss: LazyList[List[Int]]) def folded: Int = xss.map(_._sum).sum</pre>
<pre>def scoreGame(rollScores: List[RollScore]): Int = frameScores(rollScores).take(10).map(_._sum).sum</pre>	=	<pre>def scoreGame(rollScores: List[RollScore]): Int =</pre>
	<>	<pre>frameScores(rollScores).take(10).folded</pre>

```
type PinCount    = Int
type RollScore   = PinCount
type FrameScore  = List[RollScore]
type GameScore   = List[FrameScore]
```

```
def isStrike(rollScores: List[RollScore]): Boolean =
  rollScores.take(1).sum == 10

def isSpare(rollScores: List[RollScore]): Boolean =
  rollScores.take(2).sum == 10

def ballsToScore(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) || isSpare(rollScores) then 3 else 2

def frameAdvance(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) then 1 else 2

def frameScores(rollScores : List[RollScore]): LazyList[FrameScore] =
  rollScores.take(ballsToScore(rollScores))
  #:: frameScores(rollScores.drop(frameAdvance(rollScores)))

extension (xss: LazyList[List[Int]])
  def folded: Int =
    xss.map(_.sum).sum

def scoreGame(rollScores: List[RollScore]): Int =
  frameScores(rollScores).take(10).folded
```

On the next slide we turn to the **task** of **simplifying** the **folding** of the new **extension function**'s parameter.





In **part two** of this **series** we saw that **Tom Moertel**'s updated version of his **program** has a **comment** in which he links to a **web site defining** the **United States Bowling Congress rules**.

```
-- | Computes the score for a player's game having a list of `rolls`.  
--   Uses US Bowling Congress rules: https://bowl.com/keeping-score.
```

```
scoreGame :: [Int] -> Int
```

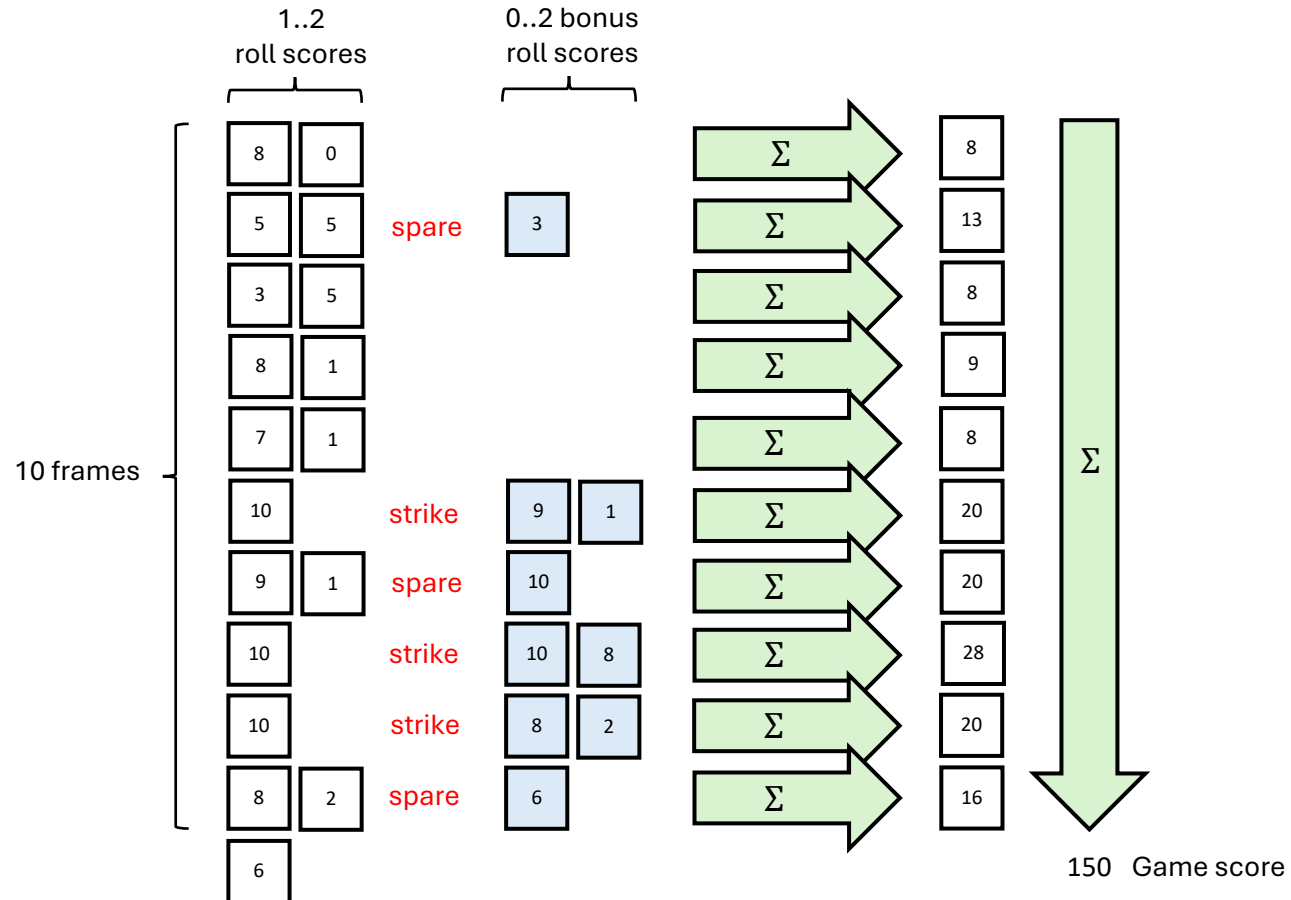
In what follows, let's use this **bowling game example** provided by the site.

1	2	3	4	5	6	7	8	9	10
8/-	5/	3/5	8/1	7/1	X	9/	X	X	8/6
8	21	29	38	46	66	86	114	134	150

Let's have a go at **visualising**
the **scoring** of the **game**.



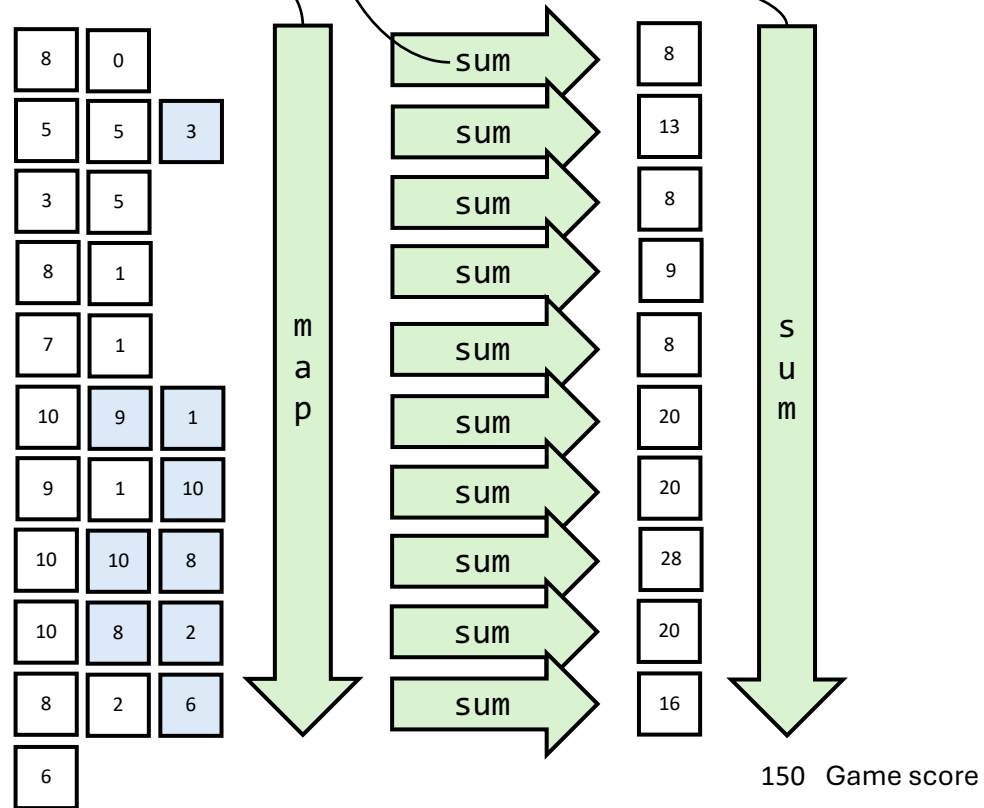
1	2	3	4	5	6	7	8	9	10
8-	5/	3/5	8/1	7/1	X	9/	X	X	8/6
8	21	29	38	46	66	86	114	134	150



And now let's **visualise** how our **'folded'** **function** currently **computes** the **score**.



```
extension (xss: LazyList[List[Int]])  
def folded: Int =  
  xss.map(_._sum)._sum
```





On the next two slides we have a go at **simplifying** the **folded function** using the **concept** of an **integer monoid**.

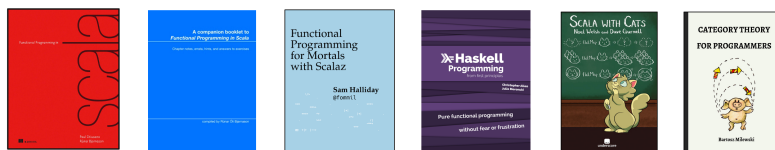
While the two slides are sufficient for our purposes, if you could do with a **more in depth refresher** on (or **introduction to**) **monoids**, consider checking out some of the decks below.

```
extension (xss: LazyList[List[Int]])
def folded: Int =
  xss.map(_._sum).sum
```

Monoids

with examples using Scalaz and Cats

based on



Part 1

slides by  @philip_schwarz

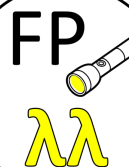
Monoids

with examples using Scalaz and Cats

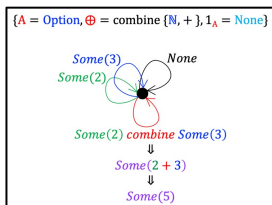
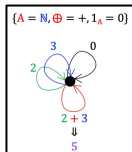
Part II - based on



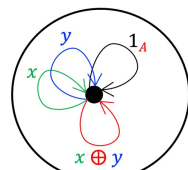
slides by  @philip_schwarz



Nat, **List** and **Option Monoids**
from scratch
Combining and **Folding**
an example



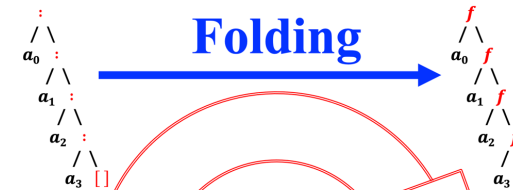
Nat



List

Option

slides by  @philip_schwarz  <http://fpilluminated.com/>

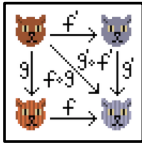


Folding

CHEAT-SHEET
#8

slides by  @philip_schwarz  <https://fpilluminated.com/>

```
import cats.implicit.*
import cats.Foldable
import cats.Monoid
```



Cats

```
val m = Monoid[Int]

assert(m.combine(2,3) == 5)
assert(m.empty == 0)

val ints = List(1,2,3,4,5)
```

```
assert(Foldable[List].fold[Int](Nil)(using m) == 0)
assert(Foldable[List].fold(ints)(using m) == 15)

assert(Foldable[List].fold[Int](Nil) == 0)
assert(Foldable[List].fold(ints) == 15)
```

```
assert(Nil.combineAll(using m) == 0)
assert(ints.combineAll(using m) == 15)

assert(List.empty[Int].combineAll == 0)
assert(ints.combineAll == 15)
```

```
assert(ints.foldMap(_ * 10) == 150)

val intLists = List(List(1,2),List(3,4,5))

assert(intLists.foldMap(_.combineAll) == 15)
```

```
def fold[A](using Monoid[A]) = Foldable[Seq].fold[A]

assert(intLists.foldMap(fold) == 15)
```

Cats provides a **'given'** **Monoid typeclass instance** for **Int**. Here we bind it to **val m** and then show the following:

- the **monoid's combine function** can be used to **sum** two integers
- the **monoid's empty value** is **zero**

Cats provides a **Foldable typeclass instance** for **List**. Here we show the following:

- The **Foldable** provides a **fold function** that can **explicitly use** the **monoid** to **fold** a **list** of **integers** into the **integers' sum**
- The **fold function** can also use the **monoid implicitly**



To make usages of **Foldable's fold function** less **verbose**, we could define an **extension function** called **folded**

```
extension [A](as: Seq[A])(using Monoid[A])
  def folded: A = Foldable[Seq].fold[A](as)

assert(List.empty[Int].folded == 0)
assert(ints.folded == 15)
```

But we don't need to do that, because **Cats** already provides **combineAll syntactic sugar** for invoking **Foldable's fold function**.

The **Foldable typeclass instance** for a **collection** also provides a **foldMap function** such that **foldMapping** a function **f** over a **collection** amounts to first **mapping f** over the **collection** and then **folding** the resulting **collection** using **Foldable's fold function**. We can **combine** the use of **foldMap** and **combineAll** to **sum** the **integers** in a **collection** of **collections** of **integers**.

To **emphasize** that we are doing **nested folds**, and to **maximise conciseness**, we can define a **fold function** that delegates to **Foldable's fold function**.

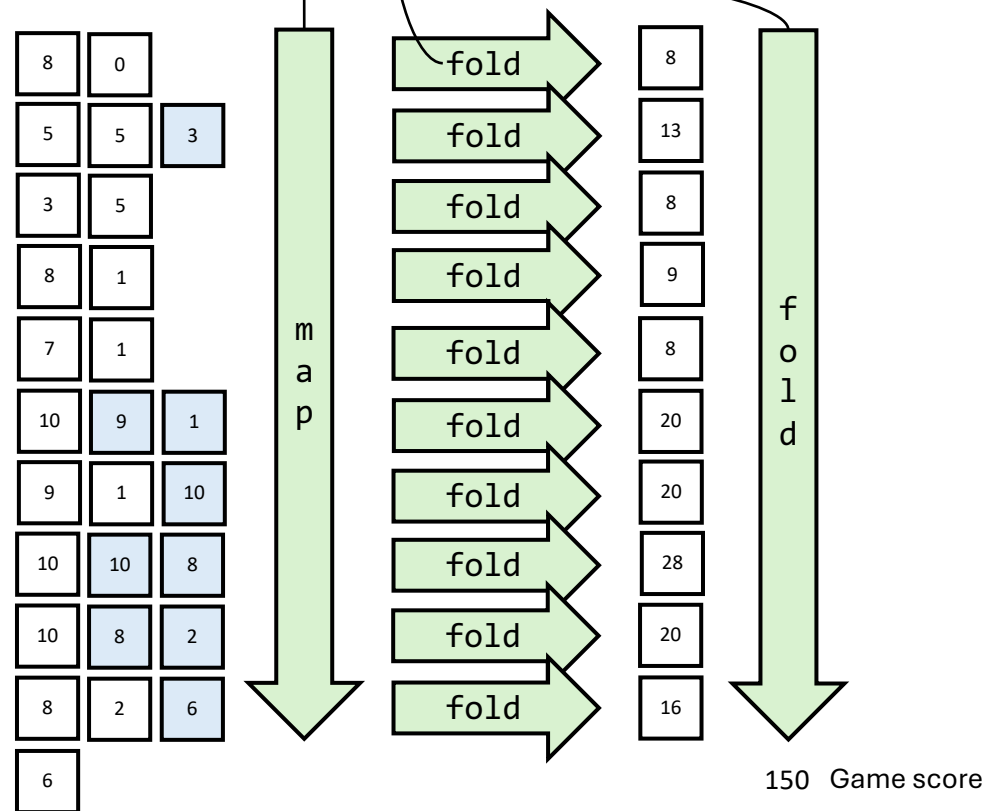
On the next slide we use **foldMap** and **fold** to **simplify** the **bowling program's folded function**.



```
extension (xss: LazyList[List[Int]])
  def folded: Int =
    xss.map(_._sum).sum
```

simplify

```
extension (xss: LazyList[List[Int]])
  def folded: Int =
    xss.foldMap(fold)
```



```
def fold[A](using Monoid[A]) =
  Foldable[List].fold[A]
```



It is possible to **simplify** the **folded function** even further by exploiting the **concept** of **Foldable composition**.

To help us set the context for examples of the latter, on the next slide we work our way through some examples of **Functor composition**.

While the next two slides suffice for our purposes, if you could do with a **more in depth refresher** on (or **introduction to**) **Functor composition**, consider checking out the first deck below. If you end up doing so, and later think you might also be interested in **Applicative composition** and **Monad composition**, then consider checking out the second deck below.

Functor Composition

including (starting from) the definition in

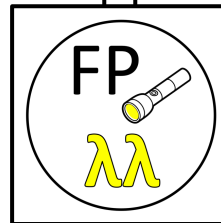


[@BartoszMilewski](#)

slides by



[@philip_schwarz](#)



Monads do not Compose

not in a generic way - there is no general way of composing monads

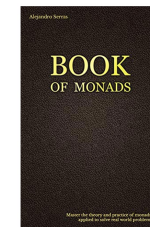
inspired by (and with excerpts from)



Functional Programming in Scala



A companion booklet to
Functional Programming in Scala

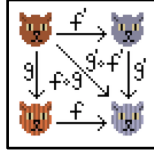


slides by



[@philip_schwarz](#)

```
import cats.*
import cats.data.Nested
import cats.implicit.*
```



Cats

```
assert(List(2,3,4) == Functor[List].map(List(1,2,3))(_ + 1))
```

Cats provides a **Functor** for **List**. Here we use it to **map** an **increment function** over a **list of integers**.



```
assert(List(2,3,4) == List(1,2,3).map(_ + 1))
```

Of course there is no need to use the **functor** when we can simply invoke **map** on the **list of integers**.



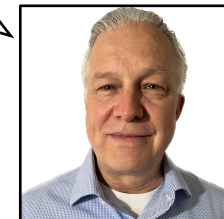
What if we want to **map** a **desired function**, not simply over a **flat data structure**, e.g. **List[Int]**, for which there is a **functor**, e.g. **Functor[List]**, but over two **nested data structures**, e.g. **List[Option[Int]]** or **Option[List[Int]]**, for both of which there is a **functor**, e.g. **Functor[List]** and **Functor[Option]**?

The obvious way to do that is to use one **functor** to **map**, over the **outer data structure**, a **function** that in turn uses the other **functor** to **map** the **desired function** over each **inner data structure**.

```
assert(List(2.some, 3.some, 4.some) == Functor[List].map(List(1.some, 2.some, 3.some))(Functor[Option].map(_)(_ + 1)))
assert(Some(List(2,3,4)) == Functor[Option].map(Some(List(1,2,3)))(Functor[List].map(_)(_ + 1)))
assert(List(List(2,3), List(4,5,6)) == Functor[List].map(List(List(1,2), List(3,4,5)))(Functor[List].map(_)(_ + 1)))
```

Of course there is no need to use the **functors** when we can simply invoke **map** on the **data structures**.

```
assert(List(2.some, 3.some, 4.some) == List(1.some, 2.some, 3.some).map(_._map(_ + 1)))
assert(Some(List(2,3,4)) == Some(List(1,2,3)).map(_._map(_ + 1)))
assert(List(List(2,3), List(4,5,6)) == List(List(1,2), List(3,4,5)).map(_._map(_ + 1)))
```





The other way is to rely on the fact that for any two given **functors**, e.g. **Functor**[**List**] and **Functor**[**Option**], there exist composite functors, e.g. **Functor**[**List**[**Option**[_]]] or **Functor**[**Option**[**List**[_]]], that can be used to **map** the **desired function** over a **nested data structure**.

```
assert(List(2.some, 3.some, 4.some) == Functor[List].compose[Option].map(List(1.some, 2.some, 3.some))(_ + 1))
assert(Some(List(2, 3, 4))          == Functor[Option].compose[List].map(Some(List(1, 2, 3)))(_ + 1))
assert(List(List(2, 3), List(4, 5, 6)) == Functor[List].compose[List].map(List(List(1, 2), List(3, 4, 5)))(_ + 1))
```



Instead of doing a **nested mapping** as shown above, i.e. by explicitly using a **composite functor**, **Cats** gives us the **convenient alternative** of simply **wrapping** the **nested data structure** in **Nested**.

```
final case class Nested[F[_], G[_], A](value: F[G[A]]) {...}
```

```
assert(List(2.some, 3.some, 4.some) == Nested(List(1.some, 2.some, 3.some)).map(_ + 1).value)
assert(Some(List(2, 3, 4))          == Nested(Option(List(1, 2, 3))).map(_ + 1).value)
assert(List(List(2, 3), List(4, 5, 6)) == Nested(List(List(1, 2), List(3, 4, 5))).map(_ + 1).value)
```



Even better: **Cats** allows us to **wrap** something in **Nested** simply by invoking **extension function** **nested**.

```
assert(List(2.some, 3.some, 4.some) == List(1.some, 2.some, 3.some).nested.map(_ + 1).value)
assert(Option(List(2, 3, 4))         == Option(List(1, 2, 3)).nested.map(_ + 1).value)
assert(List(List(2, 3), List(4, 5, 6)) == List(List(1, 2), List(3, 4, 5)).nested.map(_ + 1).value)
```



In the next three slides we are going to look at the **Foldable** equivalent of what we have just seen for **Functor**.



```
assert(List(2.some, 3.some, 4.some) == Functor[List].compose[Option].map(List(1.some, 2.some, 3.some))(_ + 1))
assert(Some(List(2, 3, 4)) == Functor[Option].compose[List].map(Some(List(1, 2, 3)))(_ + 1))
assert(List(List(2, 3), List(4, 5, 6)) == Functor[List].compose[List].map(List(List(1, 2), List(3, 4, 5)))(_ + 1))
```



Just like **functors**, **foldables** are **composable**.

Just like it is possible to **map** a **function** over **nested data structures** using a **composite functor**, it is possible to **fold nested data structures** (for whose **leaf values** there is a **monoid**) using a **composite foldable**.

```
val m = Monoid[Int]
```

```
assert(15 == Foldable[List].compose[Option].fold(List(1.some, 2.some, 3.some, 4.some, 5.some))(using m))
assert(15 == Foldable[Option].compose[List].fold(Some(List(1, 2, 3, 4, 5)))(using m))
assert(15 == Foldable[List].compose[List].fold(List(List(1, 2), List(3, 4, 5)))(using m))
```

Monoid[Int]
passed in
explicitly

```
assert(15 == Foldable[List].compose[Option].fold(List(1.some, 2.some, 3.some, 4.some, 5.some)))
assert(15 == Foldable[Option].compose[List].fold(Some(List(1, 2, 3, 4, 5))))
assert(15 == Foldable[List].compose[List].fold(List(List(1, 2), List(3, 4, 5))))
```

Monoid[Int]
passed in
implicitly



```
assert(List(2.some, 3.some, 4.some) == Nested(List(1.some, 2.some, 3.some)).map(_ + 1).value)
assert(Some(List(2,3,4)) == Nested(Option(List(1, 2, 3))).map(_ + 1).value)
assert(List(List(2,3),List(4,5,6)) == Nested(List(List(1,2),List(3,4,5))).map(_ + 1).value)
```



Just as we can do a **nested mapping** by **wrapping** the **nested data structures** in **Nested**, rather than by explicitly using a **composite functor**, we can do a **nested fold** by **wrapping** the **nested data structures** in **Nested**, rather than by explicitly using a **composite foldable**.

Notice that while after a **nested mapping** we have to ask for the **value** of the resulting **Nested**, we don't have to do that after a **nested fold** because the latter does not return a **Nested**, but rather a **value** of the same **type** as that of the **leaves** of the **nested data structures** being **folded**.

```
val m = Monoid[Int]
```

```
assert(15 == Nested(List(1.some,2.some,3.some,4.some,5.some)).fold(using m))
assert(15 == Nested(Some(List(1,2,3,4,5))).fold(using m))
assert(15 == Nested(List(List(1,2),List(3,4,5))).fold(using m))
```

Monoid[Int]
passed in
explicitly

```
assert(15 == Nested(List(1.some,2.some,3.some,4.some,5.some)).fold)
assert(15 == Nested(Some(List(1,2,3,4,5))).fold)
assert(15 == Nested(List(List(1,2),List(3,4,5))).fold)
```

Monoid[Int]
passed in
implicitly



```
assert(List(2.some, 3.some, 4.some) == List(1.some, 2.some, 3.some).nested.map(_ + 1).value)
assert(Option(List(2, 3, 4)) == Option(List(1, 2, 3)).nested.map(_ + 1).value)
assert(List(List(2, 3), List(4, 5, 6)) == List(List(1, 2), List(3, 4, 5)).nested.map(_ + 1).value)
```



Whether we intend to **map** a **function** over **nested data structures** or **fold** them, we of course have the ability to **wrap** them in **Nested** simply by invoking **extension function** `nested`.

```
val m = Monoid[Int]
```

```
assert(15 == List(1.some, 2.some, 3.some, 4.some, 5.some).nested.fold(using m))
assert(15 == Some(List(1, 2, 3, 4, 5)).nested.fold(using m))
assert(15 == List(List(1, 2), List(3, 4, 5)).nested.fold(using m))
```

`Monoid[Int]`
passed in
explicitly

```
assert(15 == List(1.some, 2.some, 3.some, 4.some, 5.some).nested.fold)
assert(15 == Some(List(1, 2, 3, 4, 5)).nested.fold)
assert(15 == List(List(1, 2), List(3, 4, 5)).nested.fold)
```

`Monoid[Int]`
passed in
implicitly



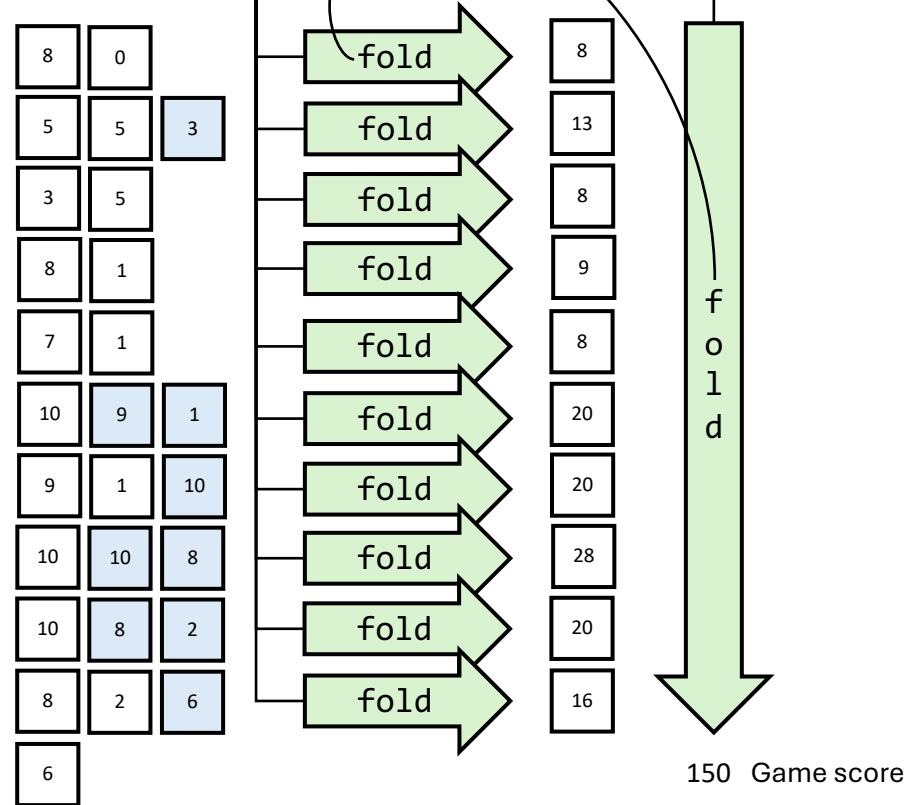
On the next slide we **simplify** the **folded function** of our **bowling game program** by using what we have just learnt about **folding nested data structures**.

```
extension (xss: LazyList[List[Int]])
  def folded: Int =
    xss.foldMap(fold)
```

```
def fold[A](using Monoid[A]) =
  Foldable[List].fold[A]
```

simplify

```
extension (xss: LazyList[List[Int]])
  def folded: Int =
    xss.nested.fold
```

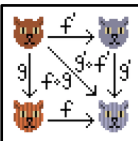




In the next three slides we **simplify** the **folded function** of our **bowling game program** even further in that we can do **nested folds** by relying on the **Numeric typeclass** rather than on the **Monoid typeclass**.



```
import cats.implicit.*
import cats.Foldable
import cats.Monoid
```



```
val m = Monoid[Int]

assert(m.combine(2,3) == 5)
assert(m.empty == 0)

val ints = List(1,2,3,4,5)

assert(Foldable[List].fold[Int](Nil)(using m) == 0)
assert(Foldable[List].fold(ints)(using m) == 15)

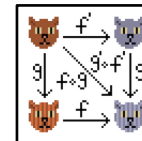
assert(Foldable[List].fold[Int](Nil) == 0)
assert(Foldable[List].fold(ints) == 15)

val intLists = List(List(1,2),List(3,4,5))

assert(intLists.nested.fold == 15)
```



```
import cats.implicit.*
import cats.Foldable
```



```
val n = Numeric[Int]

assert(n.plus(2,3) == 5)
assert(n.zero == 0)

val ints = List(1,2,3,4,5)

assert(Foldable[List].sumAll[Int](Nil)(using n) == 0)
assert(Foldable[List].sumAll(ints)(using n) == 15)

assert(Foldable[List].sumAll[Int](Nil) == 0)
assert(Foldable[List].sumAll(ints) == 15)

val intLists = List(List(1,2),List(3,4,5))

assert(intLists.nested.sumAll == 15)
```

import cats.implicits.* import cats.Foldable	=	import cats.implicits.* import cats.Foldable
import cats.Monoid	<>	
val m = Monoid[Int]		val n = Numeric[Int]
	=	
assert(m.combine(2,3) == 5) assert(m.empty == 0)	<>	assert(n.plus(2,3) == 5) assert(n.zero == 0)
	=	
val ints = List(1,2,3,4,5)		val ints = List(1,2,3,4,5)
assert(Foldable[List].fold[Int](Nil)(using m) == 0) assert(Foldable[List].fold(ints)(using m) == 15)	<>	assert(Foldable[List].sumAll[Int](Nil)(using n) == 0) assert(Foldable[List].sumAll(ints)(using n) == 15)
	=	
assert(Foldable[List].fold[Int](Nil) == 0) assert(Foldable[List].fold(ints) == 15)	<>	assert(Foldable[List].sumAll[Int](Nil) == 0) assert(Foldable[List].sumAll(ints) == 15)
	=	
val intLists = List(List(1,2),List(3,4,5))		val intLists = List(List(1,2),List(3,4,5))
assert(intLists.nested.fold == 15)	<>	assert(intLists.nested.sumAll == 15)

```

extension (xss: LazyList[List[Int]])
def folded: Int =
  xss.nested.fold

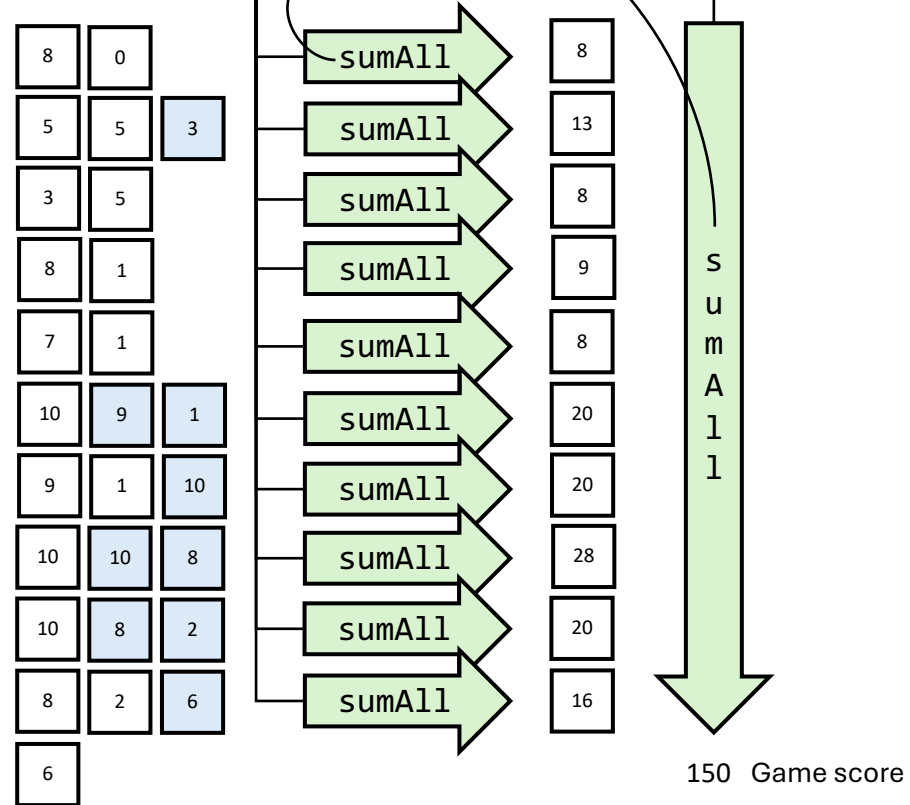
```

simplify

```

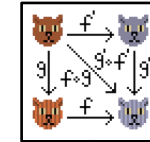
extension (xss: LazyList[List[Int]])
def folded: Int =
  xss.nested.sumAll

```





To **recap**, the next slide shows both the **original Scala program** and the **refactored** one.



```
import cats.implicits.*
```

```
type PinCount = Int
type RollScore = PinCount
type FrameScore = List[RollScore]
type GameScore = List[FrameScore]
```

```
def isStrike(rolls: List[Int]): Boolean =
  rolls.take(1).sum == 10

def isSpare(rolls: List[Int]): Boolean =
  rolls.take(2).sum == 10

def ballsToScore(rolls: List[Int]): Int =
  if isStrike(rolls) || isSpare(rolls) then 3 else 2

def frameAdvance(rolls: List[Int]): Int =
  if isStrike(rolls) then 1 else 2

def frames(rolls: List[Int]): LazyList[List[Int]] =
  rolls.take(ballsToScore(rolls))
  #:: frames(rolls.drop(frameAdvance(rolls)))

def scoreFrame(frame: List[Int]): Int =
  frame.sum

def scoreGame(rolls: List[Int]): Int =
  frames(rolls).take(10).map(scoreFrame).sum
```



```
def isStrike(rollScores: List[RollScore]): Boolean =
  rollScores.take(1).sum == 10

def isSpare(rollScores: List[RollScore]): Boolean =
  rollScores.take(2).sum == 10

def ballsToScore(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) || isSpare(rollScores) then 3 else 2

def frameAdvance(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) then 1 else 2

def frameScores(rollScores: List[RollScore]): LazyList[FrameScore] =
  rollScores.take(ballsToScore(rollScores))
  #:: frameScores(rollScores.drop(frameAdvance(rollScores)))

extension (xss: LazyList[List[Int]])
  def folded: Int = xss.nested.sumAll

def scoreGame(rollScores: List[RollScore]): Int =
  frameScores(rollScores).take(10).folded
```

ORIGINAL

REFACTORED



What about the **Clojure program**: can we **refactor** it in a similar way?

Is there a **library** that supports **folding** a **list** of **lists** of **elements** for which there is a **monoid**?

I had a quick look at the **library below**, but while it provides **monoid-related functionality**, it seems not to support **folding** with **monoids**.



<https://github.com/funcool/cats>

cats

404 badge not found chat on gitter



And there are my cats, engaged in a ritual that goes back thousands of years, tranquilly licking themselves after the meal. Practical animals, they prefer to have others provide the food ... some of them do. There must have been a split between the cats who accepted domestication and those who did not.

-- William S. Burroughs, The Cat Inside

[funcool/cats "2.4.3-beta.2"] @clojars.org

Category Theory and algebraic abstractions for Clojure and ClojureScript.

Stable:

- Documentation: <http://funcool.github.io/cats/latest/>
- API reference: <http://funcool.github.io/cats/latest/api/>

Monoid

A Monoid is a Semigroup with an identity element (mempty). For the collection types the mempty function is analogous to Clojure's empty.

Given that the values it contains form a Semigroup, we can mappend multiple *Maybe*, with Nothing being the identity element.

```
(require '[cats.core :as m])
(require '[cats.builtin])
(require '[cats.monad.maybe :as maybe])

(m/mappend (maybe/just [1 2 3])
            (maybe/nothing)
            (maybe/just [4 5 6])
            (maybe/nothing))
;; => #<Just [1 2 3 4 5 6]>
```



But then I realised that in **Clojure**, if we **reduce** a **collection** of items with an **operator** with which we normally associate a **monoid**, and the **collection** is **empty**, the result is that **monoid's zero**...

```
=>(reduce + `(1 2 3 4 5))
15
=>(reduce * `(1 2 3 4 5))
120
=>(reduce concat
  `((1 2) (3) (4 5)))
(1 2 3 4 5)
```

```
=>(reduce + `())
0
=>(reduce * `())
1
=>(reduce concat `())
()
```



...which I think is possible because in **Clojure**, given such a **'monoidal' operator** (let's call it so), it is possible to find out what the associated **monoid's zero** is by invoking the **operator** without arguments...

```
=>(+)
0
=>(*)
1
=>(concat)
()
```

... so instead of having **multiple almost identical chunks of code**, each of which **folds** a collection of collections of items of a specific type with a different **monoidal operator**...

```
=>(reduce + (map #(reduce + %) `((1 2) (3) (4 5))))
15

=>(reduce * (map #(reduce * %) `((1 2) (3) (4 5))))
120

=>(reduce concat (map #(reduce concat %)
  `(((1 2) (3)) ((4)) ((5 6)))))
(1 2 3 4 5 6)
```

...we can write a **generic fold function** that does the **folding** for **any monoidal operator**. On the next slide we use this **fold** function to **refactor** the **Clojure program** the same way we **refactored** the **Scala program**.

```
=>(defn fold
  [op xss]
  (reduce op (map #(reduce op %) xss)))
#'user/fold

=>(fold + `((1 2) (3) (4 5)))
15

=>(fold * `((1 2) (3) (4 5)))
120

=>(fold concat `(((1 2) (3)) ((4)) ((5 6)))))
(1 2 3 4 5 6)
```

```
(ns bowling-game (:use clojure.contrib.seq-utils))
```

```
(defn strike? [rolls]
  (= 10 (first rolls)))
```

```
(defn spare? [rolls]
  (= 10 (apply + (take 2 rolls))))
```

```
(defn balls-to-score
  "How many balls contribute to this frame's score?"
  [rolls]
  (cond
    (strike? rolls) 3
    (spare? rolls) 3
    :else 2))
```

```
(defn frame-advance
  "How many rolls should be consumed to advance to the next frame?"
  [rolls]
  (if (strike? rolls) 1 2))
```

```
(defn frames
  "Converts a sequence of rolls to a sequence of frames"
  [rolls]
  (when-let [rolls (seq rolls)]
    (lazy-seq (cons (take (balls-to-score rolls) rolls)
                    (frames (drop (frame-advance rolls) rolls))))))
```

```
(defn score-frame
  [frame]
  (reduce + frame))
```

```
(defn score-game
  "Score a bowling game, passed as a sequence of rolls."
  [rolls]
  (reduce + (map score-frame (take 10 (frames rolls))))))
```



Stuart Halloway



```
(ns bowling-game (:use clojure.contrib.seq-utils))
```

```
(defn strike? [rolls]
  (= 10 (first rolls)))
```

```
(defn spare? [rolls]
  (= 10 (apply + (take 2 rolls))))
```

```
(defn balls-to-score
  "How many balls contribute to this frame's score?"
  [rolls]
  (cond
    (strike? rolls) 3
    (spare? rolls) 3
    :else 2))
```

```
(defn frame-advance
  "How many rolls should be consumed to advance to the next frame?"
  [rolls]
  (if (strike? rolls) 1 2))
```

```
(defn frame-scores
  "Converts a sequence of rolls to a sequence of frames"
  [rolls]
  (when-let [rolls (seq rolls)]
    (lazy-seq (cons (take (balls-to-score rolls) rolls)
                    (frame-scores (drop (frame-advance rolls) rolls))))))
```

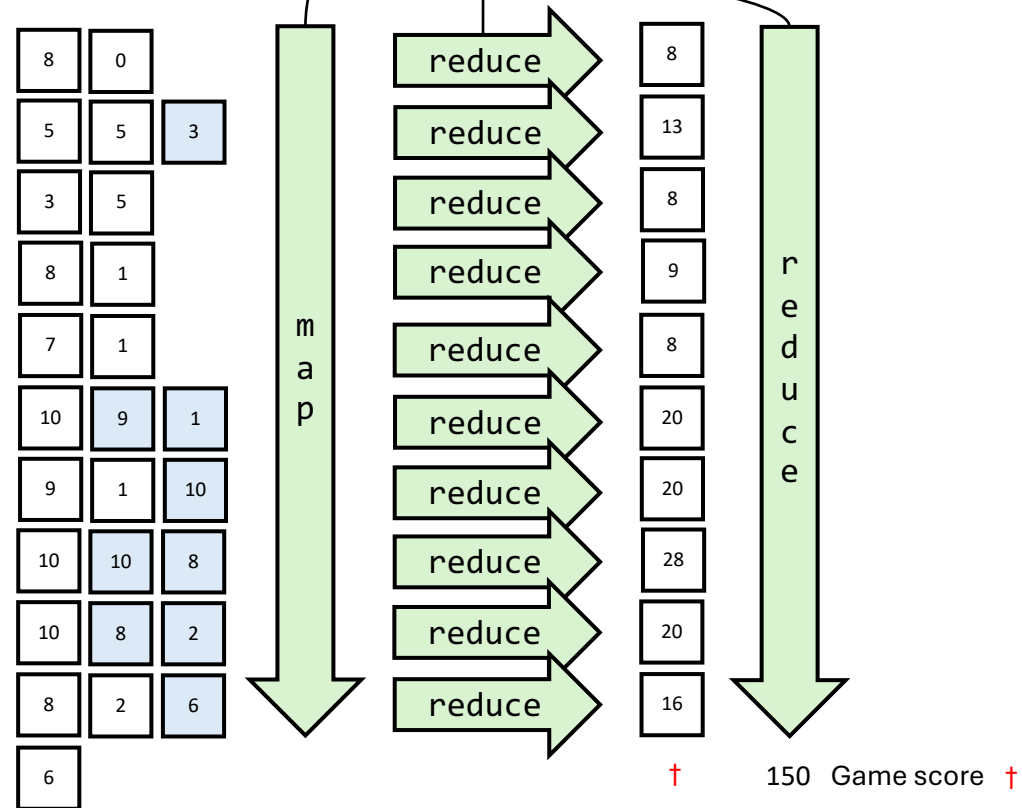
```
(defn fold
  [op xss]
  (reduce op (map #(reduce op %) xss)))
```

```
(defn score-game
  "Score a bowling game, passed as a sequence of rolls."
  [rolls]
  (fold + (take 10 (frame-scores rolls))))
```

ORIGINAL

REFACTORED

```
(defn fold
  [op xss]
  (reduce op (map #(reduce op %) xss)))
```



† this illustration shows the behaviour of the function when `op` is bound to `+` and so **reduction** is **summation**



So far in this deck we have **refactored** the following:

- The **Scala translation** of **Stuart Halloway's Clojure program**
- **Stuart Halloway's Clojure program**

Let's now turn to **refactoring** the **Haskell translation** of the **Clojure program**, in the same way that we **refactored** the **Scala translation**.

The next slide shows the bulk of the **refactoring** in a **single step**.

The subsequent two slides show the **Haskell** equivalent of the **Monoid basics** that we used when **refactoring** the **Scala program**.



ORIGINAL

```

isStrike :: [Int] -> Bool
isStrike rolls = sum (take 1 rolls) == 10

isSpare :: [Int] -> Bool
isSpare rolls = sum (take 2 rolls) == 10

ballsToScore :: [Int] -> Int
ballsToScore rolls =
    if isStrike rolls || isSpare rolls then 3 else 2

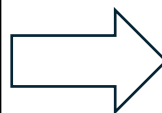
frameAdvance :: [Int] -> Int
frameAdvance rolls =
    if isStrike rolls then 1 else 2

frames :: [Int] -> [[Int]]
frames rolls =
    (take (ballsToScore rolls) rolls)
    : frames (drop (frameAdvance rolls) rolls)

scoreFrame :: [Int] -> Int
scoreFrame = sum

scoreGame :: [Int] -> Int
scoreGame rolls = sum (map scoreFrame (take 10 (frames rolls)))

```



```

type PinCount = Int
type RollScore = PinCount
type FrameScore = [RollScore]
type GameScore = [FrameScore]

```

REFACTORED - V1

```

isStrike :: [RollScore] -> Bool
isStrike rollScores = sum (take 1 rollScores) == 10

isSpare :: [RollScore] -> Bool
isSpare rollScores = sum (take 2 rollScores) == 10

ballsToScore :: [RollScore] -> Int
ballsToScore rollScores =
    if isStrike rollScores || isSpare rollScores then 3 else 2

frameAdvance :: [RollScore] -> Int
frameAdvance rollScores =
    if isStrike rollScores then 1 else 2

frameScores :: [RollScore] -> [FrameScore]
frameScores rollScores =
    (take (ballsToScore rollScores) rollScores)
    : frameScores (drop (frameAdvance rollScores) rollScores)

† reduce :: [[Int]] -> Int
reduce xss = sum (fmap sum xss)

scoreGame :: [RollScore] -> Int
scoreGame rollScores = reduce (take 10 (frameScores rollScores))

```



is there a haskell analogue to scala cats Nested?



AI Mode

All

Forums

Videos

Short videos

Shopping

Images

More ▾

Tools ▾

✦ AI Overview

Yes, the direct Haskell analogue to Scala Cats' `Nested` is `Data.Functor.Compose`, which is built into Haskell's `base` standard library.  Haskell Language

Just like `Nested[F, G, A]` in Cats wrapping an underlying `F[G[A]]`, Haskell's `Compose` `f g a` wraps a nested type constructor `f (g a)`.  YouTube · Tsoding +1

Direct Comparison

Feature	Scala (Cats)	Haskell (<code>base</code>)
Type Definition	<code>Nested[F[_], G[_], A]</code>	<code>newtype Compose f g a</code>
Constructor	<code>Nested(v)</code>	<code>Compose v</code>
Unwrapping	<code>.value</code>	<code>getCompose</code>

```
> import cats.implicit.*
> import cats.syntax.*
> import cats.Monoid
```

```
> val empty = Monoid[Int].empty
val empty: Int = 0
```

```
> 2 |+| empty
val res1: Int = 2
```

```
> empty |+| 3
val res2: Int = 3
```

```
> 2 |+| 3
val res3: Int = 5
```

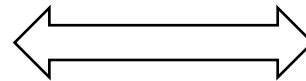
```
> empty |+| empty
val res4: Int = 0
```

```
> List.empty[Int].combineAll
val res5: Int = 0
```

```
> List(1, 2, 3, 4).combineAll
val res6: Int = 10
```

```
> List(1, 2, 3, 4).foldMap(_ + 10)
val res7: Int = 50
```

```
empty
combine
|+|
combineAll
foldMap
```



```
newtype Sum a = Sum {getSum :: a}
```

```
> import Data.Monoid
> import Data.Foldable
```

```
> getSum (mempty::Sum Int)
0
```

```
> getSum (Sum 2 <> mempty)
2
```

```
> getSum (mempty <> Sum 3)
3
```

```
> getSum (Sum 2 <> Sum 3)
5
```

```
> getSum (mempty <> mempty::Sum Int)
0
```

```
> getSum (fold ([]::[Sum Int]))
0
```

```
> getSum (fold (fmap Sum [1, 2, 3, 4]))
10
```

```
> getSum (foldMap Sum [1, 2, 3, 4])
10
```

```
mempty
mappend
<>
fold
foldMap
```





The next slide shows the **Haskell equivalent** of the **subsequent steps** that we took when **refactoring** the **Scala translation**.

The **dot** seen in the **Haskell** code on that slide is the **higher-order function** for **function composition**.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Right to left function composition.

```
(f . g) x = f (g x)
```

```
f . id = f = id . f
```

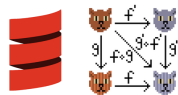


The **Scala equivalent** is the **compose** function (**backward composition**), as opposed to the **andThen** function (**forward composition**).

If you could do with a few more details, see the deck below.



Function Composition - forward composition versus backward composition 



```
extension (xss: LazyList[List[Int]])  
  def folded: Int = xss.map(_.sum).sum
```



```
extension (xss: LazyList[List[Int]])  
  def folded: Int = xss.foldMap(fold)  
  
def fold[A](using Monoid[A]) =  
  Foldable[List].fold[A]
```



```
extension (xss: LazyList[List[Int]])  
  def folded: Int = xss.nested.fold
```



```
extension (xss: LazyList[List[Int]])  
  def folded: Int = xss.nested.sumAll
```



```
reduce :: [[Int]] -> Int  
reduce = sum (fmap sum)
```



```
reduce :: [[Int]] -> Int  
reduce = getSum.(foldMap (foldMap Sum))
```



```
reduce :: [[Int]] -> Int  
reduce = getSum.(foldMap Sum).Compose †
```



```
reduce :: [[Int]] -> Int  
reduce = sum.Compose ††
```

† foldMap f (Compose t) = foldMap (foldMap f) t †† sum :: (Foldable t, Num a) => t a -> a
sum (Compose t) = getSum (foldMap' (Sum . sum) t)
foldMap' f = foldl' (\ acc a -> acc <> f a) mempty



The next slide shows the **Haskell program** at the end of the **refactoring**.

The subsequent slide shows the **refactored Scala program** next to the **refactored Haskell program**.

The slide after that shows the **refactored Scala program** next to the **refactored Clojure program**.



```
type PinCount    = Int
type RollScore   = PinCount
type FrameScore  = [RollScore]
type GameScore   = [FrameScore]
```

```
import Data.Functor.Compose
```

REFACTORED - V2

```
isStrike :: [RollScore] -> Bool
isStrike rollScores = sum (take 1 rollScores) == 10

isSpare :: [RollScore] -> Bool
isSpare rollScores = sum (take 2 rollScores) == 10

ballsToScore :: [RollScore] -> Int
ballsToScore rollScores =
  if isStrike rollScores || isSpare rollScores then 3 else 2

frameAdvance :: [RollScore] -> Int
frameAdvance rollScores =
  if isStrike rollScores then 1 else 2

frameScores :: [RollScore] -> [FrameScore]
frameScores rollScores =
  (take (ballsToScore rollScores) rollScores)
  : frameScores (drop (frameAdvance rollScores) rollScores)

reduce :: [[Int]] -> Int
reduce = sum.Compose

scoreGame :: [RollScore] -> Int
scoreGame rollScores =
  reduce (take 10 (frameScores rollScores))
```



```

type PinCount = Int
type RollScore = PinCount
type FrameScore = List[RollScore]
type GameScore = List[FrameScore]

```

```
import cats.implicit
```



```

def isStrike(rollScores: List[RollScore]): Boolean =
  rollScores.take(1).sum == 10

def isSpare(rollScores: List[RollScore]): Boolean =
  rollScores.take(2).sum == 10

// How many balls contribute to this frame's score?
def ballsToScore(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) || isSpare(rollScores) then 3 else 2

// How many rolls should be consumed to advance to the next frame?
def frameAdvance(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) then 1 else 2

// Converts a sequence of roll scores to a sequence of frame scores
def frameScores(rollScores : List[RollScore]): LazyList[FrameScore] =
  rollScores.take(ballsToScore(rollScores))
  #:: frameScores(rollScores.drop(frameAdvance(rollScores)))

extension (xss: LazyList[List[Int]])
  def folded: Int = xss.nested.sumAll

// Score a bowling game, passed as a sequence of roll scores.
def scoreGame(rollScores: List[RollScore]): Int =
  frameScores(rollScores).take(10).folded

```



```

type PinCount = Int
type RollScore = PinCount
type FrameScore = [RollScore]
type GameScore = [FrameScore]

```

```
import Data.Functor.Compose
```

```

isStrike :: [RollScore] -> Bool
isStrike rollScores = sum (take 1 rollScores) == 10

isSpare :: [RollScore] -> Bool
isSpare rollScores = sum (take 2 rollScores) == 10

-- How many balls contribute to this frame's score?
ballsToScore :: [RollScore] -> Int
ballsToScore rollScores = if isStrike rollScores || isSpare rollScores then 3 else 2

-- How many rolls should be consumed to advance to the next frame?
frameAdvance :: [RollScore] -> Int
frameAdvance rollScores = if isStrike rollScores then 1 else 2

-- Converts a sequence of roll scores to a sequence of frame scores
frameScores :: [RollScore] -> [FrameScore]
frameScores rollScores = (take (ballsToScore rollScores) rollScores)
  : frameScores (drop (frameAdvance rollScores) rollScores)

reduce :: [[Int]] -> Int
reduce = sum.Compose

// Score a bowling game, passed as a sequence of roll scores.
scoreGame :: [RollScore] -> Int
scoreGame rollScores = reduce (take 10 (frameScores rollScores))

```



```

type PinCount    = Int
type RollScore   = PinCount
type FrameScore  = List[RollScore]
type GameScore   = List[FrameScore]

```

```
import cats.implicits
```



```

def isStrike(rollScores: List[RollScore]): Boolean =
  rollScores.take(1).sum == 10

def isSpare(rollScores: List[RollScore]): Boolean =
  rollScores.take(2).sum == 10

// How many balls contribute to this frame's score?
def ballsToScore(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) || isSpare(rollScores) then 3 else 2

// How many rolls should be consumed to advance to the next frame?
def frameAdvance(rollScores: List[RollScore]): Int =
  if isStrike(rollScores) then 1 else 2

// Converts a sequence of roll scores to a sequence of frame scores
def frameScores(rollScores : List[RollScore]): LazyList[FrameScore] =
  rollScores.take(ballsToScore(rollScores))
  #:: frameScores(rollScores.drop(frameAdvance(rollScores)))

extension (xss: LazyList[List[Int]])
  def folded: Int = xss.nested.sumAll

// Score a bowling game, passed as a sequence of roll scores.
def scoreGame(rollScores: List[RollScore]): Int =
  frameScores(rollScores).take(10).folded

```



```
(ns bowling-game (:use clojure.contrib.seq-utils))
```

```

(defn strike? [rolls]
  (= 10 (first rolls)))

(defn spare? [rolls]
  (= 10 (apply + (take 2 rolls))))

(defn balls-to-score
  "How many balls contribute to this frame's score?"
  [rolls]
  (cond
    (strike? rolls) 3
    (spare? rolls) 3
    :else 2))

(defn frame-advance
  "How many rolls should be consumed to advance to the next frame?"
  [rolls]
  (if (strike? rolls) 1 2))

(defn frame-scores
  "Converts a sequence of rolls to a sequence of frames"
  [rolls]
  (when-let [rolls (seq rolls)]
    (lazy-seq (cons (take (balls-to-score rolls) rolls)
                    (frame-scores (drop (frame-advance rolls) rolls))))))

(defn fold
  [op xss]
  (reduce op (map #(reduce op %) xss)))

(defn score-game
  "Score a bowling game, passed as a sequence of rolls."
  [rolls]
  (fold + (take 10 (frame-scores rolls))))

```



That's all for **part three**.

I hope you liked it.

See you in **part four**.