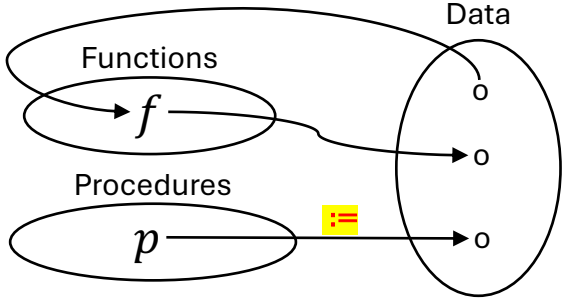
# The Bowling Game
# From Imperative to Functional Programming

Haskell Scala Clojure

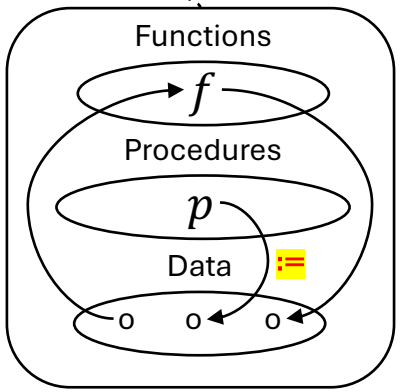
Part 2

Ron Jeffries

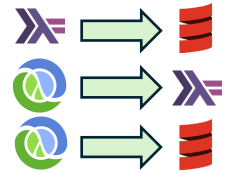
**Procedural** Programming




**Oject Oriented** Programming

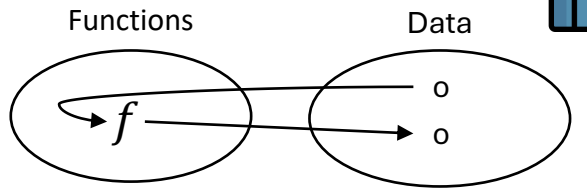



Philip Schwarz



**Functional** Programming

Dan




Tom Moertel

Eric Kidd

Stuart Halloway

slides by  @philip_schwarz FP Iλλuminated https://fpilluminated.org/

@philip_schwarz

Welcome to part two of this **series**.

Remember how among the **bowling game programs** that we looked at in **part 1** there was a **Haskell** program written by **Tom Moertel** back in 2006?

See next slide for a brief reminder of the **blog post** in which **Tom** wrote about the **program**.

2006

https://blog.moertel.com/posts/2006-04-05-the-bowling-game-kata-in-haskell.html

Tom Moertel

@tmoertel



# The Bowling Game Kata in Haskell

*By Tom Moertel*

*Posted on April 5, 2006*
*Tags: haskell, kata, bowling*

On the WPLUG mailing list I came across a post about the formation of a Pittsburgh Coding Dojo. The idea is to get a bunch of hackers together and have them work on solving a challenge problem with the goal of sharpening their programming skills and learning from each other.

There was a trial meeting on 31 March that focused on The Bowling Game Kata. The challenge was essentially to write some code that scores a full (ten-frame) game of bowling. A game is represented by a series of "rolls," each being the number of pins knocked down by a roll of the bowling ball. The scoring function must determine frame boundaries from the sequence of rolls and score all ten frames according to the rules of bowling, i.e., taking into account spares and strikes and the final frame.

The challenge sounded like a fun lunch-break problem, and so I whipped up the following solution in Haskell. (You might find it interesting to compare this solution to the Java-based solutions on the web.)

```haskell
{-
   My solution to "The Bowling Game Kata"
   Tom Moertel <tom@moertel.com>
   2006-04-05

   See http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata
-}

module Bowling (score) where

import Test.HUnit

-- | Compute the score for the list of rolls 'rs'

score rs = sc 0 1 rs

-- accumulate the score 's' and frame count 'f' while consuming a
-- list of rolls 'rs' one frame at a time

sc s 11 _  = s              -- frame 11 means all done; return score
sc s f rs  = case rs of     -- otherwise, consume the frame & recurse
    10:rs'                   -> sc' 3 rs'   -- strike
    x:y:rs' | x + y == 10    -> sc' 3 rs'   -- spare
            | otherwise      -> sc' 2 rs'   -- normal
    _                        -> error "ill-formed sequence of rolls"
  where
    -- accumulate the next 'n' rolls into the score and recurse
    sc' n rs' = sc (s + sum (take n rs)) (f + 1) rs'
```

Well, it turns out that **20 years later**, inspired by **part 1** of this **series**, **Tom** has published a follow up **blog post** in which he writes about a **revised version** of his **program**!

See next slide for a brief intro to **Tom**'s **new blog post**.

2026

https://blog.moertel.com/posts/2026-07-30-how-i-would-do-the-bowling-game-kata-in-haskell-today.html

**Tom Moertel**

@tmoertel



# How I would do the "Bowling Game Kata" in Haskell today

*By Tom Moertel*

*Posted on July 30, 2026*

*Tags: code, comments, clean code, good code, source code, haskell, kata, bowling, tdd*

Recently, Philip Schwarz from fpilluminated.org wrote to let me know about a new slide deck he had published: The Bowling Game - From Imperative to Functional Programming - Part 1. In the deck, he revisits the "Bowling Game Kata" and summarizes some of the discussion (controversy?) spurred by this exercise over the last two decades. Philip tells this story in a rather clever fashion that blends code, graphics, and comic-book storytelling techniques, so consider checking out the deck.

Anyway, he wrote in particular to kindly let me know that one of the code examples he examines is from my blog, way back when I was learning Haskell in 2006: The Bowling Game Kata in Haskell. This got me thinking about how I might write this code today. So I revised the code, and here it is. (Be sure to read my rationale for each change, which follows the code.)

```haskell
{-
   My solution to "The Bowling Game Kata"
   Tom Moertel <tom@moertel.com>
   Created 2006-04-05. Revised 2026-07-30 to show "production style."

   See http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata.
-}

module Bowling (scoreGame) where

import Test.HUnit

-- | 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
scoreGame rolls = go 0 1 rolls
  where
    go :: Int -> Int -> [Int] -> Int
    go !score !frame rs = case rs of
        _       | frame == 11 -> score              -- 10th frame ends game.
        10:rs'                -> accumFrame 3 rs'   -- Strike scores 3 rolls.
        x:y:rs' | x + y == 10 -> accumFrame 3 rs'   -- Spare scores 3 rolls.
                | otherwise   -> accumFrame 2 rs'   -- Others score 2 rolls.
        _                     -> error "ill-formed sequence of rolls"
      where
        accumFrame n rs' = go (score + sum (take n rs)) (frame + 1) rs'
```

The next slide shows **Tom**'s **two versions** of the **program side by side**, and the subsequent slide shows the **minor change** that **Tom** made to his **test suite**.

```haskell
-- | Compute the score for the list of rolls 'rs'

score rs = sc 0 1 rs

-- accumulate the score 's' and frame count 'f' while consuming a
-- list of rolls 'rs' one frame at a time

sc s 11 _  = s              -- frame 11 means all done; return score
sc s f rs  = case rs of     -- otherwise, consume the frame & recurse
    10:rs'                   -> sc' 3 rs'  -- strike
    x:y:rs' | x + y == 10    -> sc' 3 rs'  -- spare
            | otherwise      -> sc' 2 rs'  -- normal
    _                        -> error "ill-formed sequence of rolls"
  where
    -- accumulate the next 'n' rolls into the score and recurse
    sc' n rs' = sc (s + sum (take n rs)) (f + 1) rs'
```

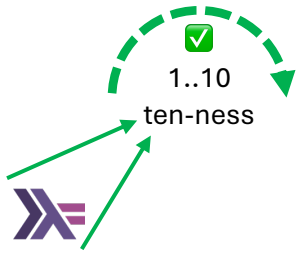
Version 6 - Haskell – domain-free - Tom Moertel

2006

So who was the **intended audience** for my **2006 bowling code**?

Then, I was writing the **logic** mainly for **myself** and was **golfing** the solution **somewhat** in a **challenge** to see how **simply** and **compactly** I could represent it.

But my code's **commentary** was for **people new to Haskell**, which at the time was something of a curiosity among programming communities.




**Tom Moertel**

@tmoertel

```haskell
-- | 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
scoreGame rolls = go 0 1 rolls
  where
    go :: Int -> Int -> [Int] -> Int
    go !score !frame rs = case rs of
        _          | frame == 11 -> score              -- 10th frame ends game.
        10:rs'                   -> accumFrame 3 rs'   -- Strike scores 3 rolls.
        x:y:rs'  | x + y == 10   -> accumFrame 3 rs'   -- Spare scores 3 rolls.
                 | otherwise     -> accumFrame 2 rs'   -- Others score 2 rolls.
        _                        -> error "ill-formed sequence of rolls"
      where
        accumFrame n rs' = go (score + sum (take n rs)) (frame + 1) rs'
```

2026

Production grade revision of 'Version 6 - Haskell – domain-rich - Tom Moertel'

This time around, I am writing my **code** for **Haskell programmers** who are reasonably **"skilled in the art"**: they are **competent** in the **language**, its **standard libraries**, and its **idioms**.

That is, I'm writing the **code** the way I probably would in a **production shop** that used **Haskell**.

(Be sure to read my rationale for each change, which follows the code.)

```
{-
                    *** Unit tests ***

            *Bowling> runTestTT tests
            Cases: 9  Tried: 9  Errors: 0  Failures: 0
-}

tests = test
    [ "gutters"       ~: score  (rep 20  0)          ~?=   0
    , "ones"          ~: score  (rep 20  1)          ~?=  20
    , "fives"         ~: score  (rep 22  5)          ~?= 150
    , "strikes"       ~: score  (rep 12 10)          ~?= 300
    , "1 + gutters"   ~: score  (1 : rep 19 0)       ~?=   1
    , "first spare"   ~: score  (5:5:5 : rep 17 0)   ~?=  20
    , "first strike"  ~: score  (10:5:5 : rep 17 0)  ~?=  30
    , "last spare"    ~: rscore (5:5:5 : rep 18 0)   ~?=  15
    , "last strike"   ~: rscore (5:5:10 : rep 18 0)  ~?=  20
    ]
  where
    rep    = replicate
```

=

```
{-
                    *** Unit tests ***

            *Bowling> runTestTT tests
            Cases: 9  Tried: 9  Errors: 0  Failures: 0
-}

tests = test
    [ "gutters"       ~: score  (rep 20  0)          ~?=   0
    , "ones"          ~: score  (rep 20  1)          ~?=  20
    , "fives"         ~: score  (rep 22  5)          ~?= 150
    , "strikes"       ~: score  (rep 12 10)          ~?= 300
    , "1 + gutters"   ~: score  (1 : rep 19 0)       ~?=   1
    , "first spare"   ~: score  (5:5:5 : rep 17 0)   ~?=  20
    , "first strike"  ~: score  (10:5:5 : rep 17 0)  ~?=  30
    , "last spare"    ~: rscore (5:5:5 : rep 18 0)   ~?=  15
    , "last strike"   ~: rscore (5:5:10 : rep 18 0)  ~?=  20
    ]
  where
    rep    = replicate
```

<>

```
    rscore = score . reverse  -- reverse list and then score it
```

```
    score  = scoreGame
    rscore = score . reverse  -- Scores a reversed list of rolls.
```


**Tom Moertel**

@tmoertel

I would like to **thank Philip Schwarz** for **spurring** this **discussion**. It caused me to think about why I believe the things I do about code and to put those beliefs in writing. (I can't trust that I understand something unless I can write about it clearly.)

If you are **interested** in reading these **writings**, on my **blog** they are tagged with **"source code"**.

I am grateful to **Tom**, not only for his **original 2006 post**, which together with his **contributions** to related blog posts, e.g. those written by **Ron Jeffries**, played an **inspirational role** in the writing of the **first part** of this **series**, but also for his **follow up 2026 post**, which plays a role in this **second part** of the series.

See below for **Tom**'s **"source code" blog posts** at the time of writing this **deck**.

Comments Are Inevitable – August 14, 2026

How I would do the "Bowling Game Kata" in Haskell today – July 30, 2026

Beyond "Clean Code": Why Your Comments Matter – July 27, 2026

What Makes Good Code Good – July 24, 2026

On the Nature and Purpose of Code – July 23, 2026

In light of Tom's blog posts arguing that comments play a more significant role in making code understandable than may have been accorded to them by some pundits, it is time to amend the definition of domain-free code by getting it to mention comments:

- domain-free code: code whose naming and comments reveal a negligible amount of domain knowledge
- domain-rich code: code rich in domain knowledge (written as if the domain matters)

While it is tempting to also add an intermediate definition, half-way between domain-free and domain-rich (domain-aware? domain-neutral?), I am going to refrain from doing that, at least for now, because I think these kinds of imprecise definitions are of limited use, and adding another one would also lead to us spending more time classifying programs than is worthwhile given our current purposes.

On the next slide we **translate** the **latest revision** of **Tom**'s **program** into **Scala**, and on the subsequent one we write a **slightly simplified** version of the latter that uses **default parameters**, so as to eliminate the need for a **go function**. After that there is a slide showing the **test suite** for the two **Scala programs**.

```haskell
-- | 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
scoreGame rolls = go 0 1 rolls
  where
    go :: Int -> Int -> [Int] -> Int
    go !score !frame rs = case rs of
        _          | frame == 11 -> score              -- 10th frame ends game.
        10:rs'                   -> accumFrame 3 rs'   -- Strike scores 3 rolls.
        x:y:rs' | x + y == 10 -> accumFrame 3 rs'   -- Spare scores 3 rolls.
                | otherwise   -> accumFrame 2 rs'   -- Others score 2 rolls.
        _                        -> error "ill-formed sequence of rolls"
      where
        accumFrame n rs' = go (score + sum (take n rs)) (frame + 1) rs'
```

Production grade revision of ‘Version 6 - **Haskell** – **domain-free** - **Tom Moertel**‘

```scala
def scoreGame(rolls: List[Int]): Int =
  def go(score: Int, frame: Int, rolls: List[Int]): Int =
    def accumFrame(n: Int, restOfRolls: List[Int]): Int =
      go(score + rolls.take(n).sum, frame + 1, restOfRolls)
    rolls match
      case _ if frame == 11 =>          score                // 10th frame ends game.
      case 10::rest =>                  accumFrame(3, rest)  // Strike scores 3 rolls.
      case x::y::rest if x + y == 10 => accumFrame(3, rest)  // Spare scores 3 rolls
      case _::_::rest =>                accumFrame(2, rest)  // Others score 2 rolls.
      case _ =>                         throw IllegalArgumentException("ill-formed sequence of rolls")
  go(score = 0, frame = 1, rolls)
```

**Scala** translation of Production grade revision of ‘Version 6 - **Haskell** – **domain-rich** - **Tom Moertel**‘

```scala
def scoreGame(rolls: List[Int]): Int =
  def go(score: Int, frame: Int, rolls: List[Int]): Int =
    def accumFrame(n: Int, restOfRolls: List[Int]): Int =
      go(score + rolls.take(n).sum, frame + 1, restOfRolls)
    rolls match
      case _ if frame == 11 =>          score               // 10th frame ends game.
      case 10::rest =>                  accumFrame(3, rest) // Strike scores 3 rolls.
      case x::y::rest if x + y == 10 => accumFrame(3, rest) // Spare scores 3 rolls
      case _::_::rest =>                accumFrame(2, rest) // Others score 2 rolls.
      case _ =>                         throw IllegalArgumentException("ill-formed sequence of rolls")
  go(score = 0, frame = 1, rolls)
```

simplify

**Scala** translation of Production grade revision of 'Version 6 - **Haskell** – **domain-rich** - **Tom Moertel**'

```scala
def scoreGame(rolls: List[Int], currentScore: Int = 0, frame: Int = 1, ): Int =
  def accumFrame(n: Int, restOfRolls: List[Int]): Int =
    scoreGame(restOfRolls, currentScore + rolls.take(n).sum, frame + 1)
  rolls match
    case _ if frame == 11 =>          currentScore        // 10th frame ends game.
    case 10::rest =>                  accumFrame(3, rest) // Strike scores 3 rolls.
    case x::y::rest if x + y == 10 => accumFrame(3, rest) // Spare scores 3 rolls
    case _::_::rest =>                accumFrame(2, rest) // Others score 2 rolls.
    case _ =>                         throw IllegalArgumentException("ill-formed sequence of rolls")
```

Simplified **Scala** translation of Production grade revision of 'Version 6 - **Haskell** – **domain-rich** - **Tom Moertel**'

Robert C. Martin

Ron Jeffries

Pit

```scala
import org.scalatest.funsuite.AnyFunSuite
import scala.util.{Failure, Try}

class BowlingTest extends AnyFunSuite {

  test("gutter"):
    assert(0 == scoreGame(List.fill(20)(0)))

  test("allOnes"):
    assert(20 == scoreGame(List.fill(20)(1)))

  test("oneSpare"):
    assert(24 == scoreGame(5::5::7::List.fill(17)(0)))

  test("oneStrike"):
    assert(20 == scoreGame(10::2::3::List.fill(16)(0)))

  test("perfect game"):
    assert(300 == scoreGame(List.fill(12)(10)))

  ////////////////////////////////////////

  test("alternating strike spare"):
    assert(200 == scoreGame(List(10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10)))

  test("alternating spare strike"):
    assert(200 == scoreGame(List(5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5)))

  test("pit strike final frame"):
    assert(15 == scoreGame(List(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,2,3)))

  test("pit strike ninth frame"):
    assert(20 == scoreGame(List(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 2,3)))

  ////////////////////////////////////////

  test("ill-formed sequence of rolls"): assert(
    Failure(IllegalArgumentException("ill-formed sequence of rolls")).toString
    == Try{ scoreGame(List.empty) }.toString)
}
```

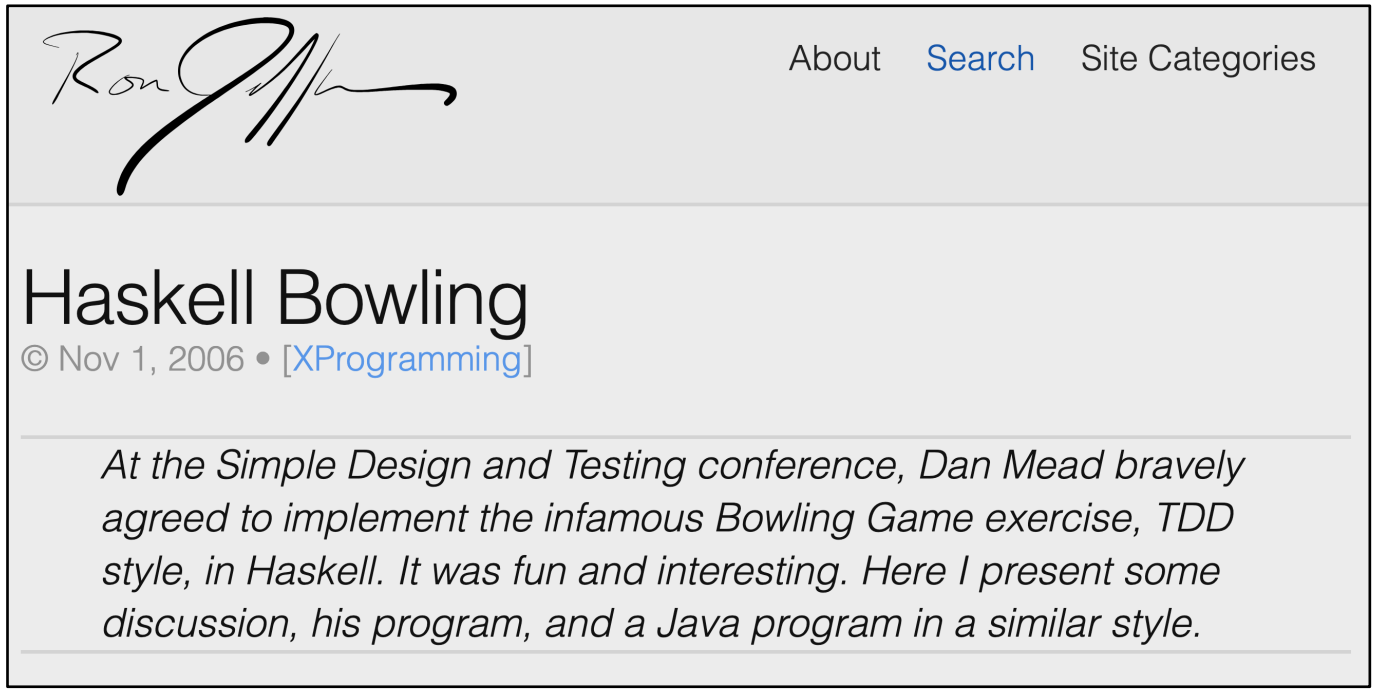
All three of the **Haskell programs** that we have seen so far (see next slide) are related to **Ron Jeffries**' 2006 **Haskell Bowling** series of blog posts:

- **Dan Mead**'s **program** was the subject of the series
- Both my **program** and **Dan**'s were found to lack the **ten-ness** defined in the series
- The series referred to **Tom**'s **program** as having the advantage of being the one that worked

The same is true for the **Haskell program** that we are turning to next, which is the subject of a **blog post** that is **Eric Kidd**'s response to **Ron Jeffries**' **series**.

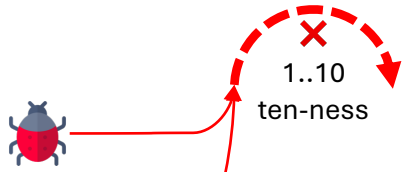
See the **slide after next** for a brief **introduction** to **Eric**'s **post**, and the **subsequent two slides** for his **program** and a **brief summary** of his **explanation** of the **program**, based on excerpts from his post.

**Ron Jeffries**




https://ronjeffries.com/xprog/articles/dbchaskellbowling/

```haskell
score [x, y]                          = x + y      -- Normal Frame
score [10, x, y]                      = 10 + x + y -- Strike
score [x, y, z]                       = 10 + z     -- Spare
score (10:x:y:rest)                   = 10 + x + y + score (x:y:rest) -- Strike
score (x:y:z:rest)  | (x + y) == 10   = 10 + z + score (z:rest)       -- Spare
score (x:y:rest)                      = x + y + score rest           -- Normal Frame
```




Version 2 - **Haskell** - **Philip Schwarz**

```haskell
score ([]) = 0
score (x:[]) = x
score (x:y:[]) = x + y
score (x:y:z:[]) = x + y + z
score (x:y:z:xs) = if  (x == 10)             then x + y + z +  score(y:z:xs)
                   else if (((x + y) == 10)) then x + y + z +  score(z:xs)
                   else x + y + score(z:xs)
```

Version 3 - **Haskell** - **domain-free** - **Dan Mead**

```haskell
-- | 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
scoreGame rolls = go 0 1 rolls
  where
    go :: Int -> Int -> [Int] -> Int
    go !score !frame rs = case rs of
            _         | frame == 11 -> score             -- 10th frame ends game.
        10:rs'                      -> accumFrame 3 rs'  -- Strike scores 3 rolls.
        x:y:rs' | x + y == 10       -> accumFrame 3 rs'  -- Spare scores 3 rolls.
                | otherwise         -> accumFrame 2 rs'  -- Others score 2 rolls.
            _                       -> error "ill-formed sequence of rolls"
      where
        accumFrame n rs' = go (score + sum (take n rs)) (frame + 1) rs'
```


Production grade revision of 'Version 6 - **Haskell** – **domain-rich** - **Tom Moertel**'

# Bowling in Haskell: A response to Ron Jeffries

Apr 28, 2007 • by Eric Kidd

Bowling is a tricky game to score. It's just complicated enough to act as a good programming exercise. And Ron Jeffries has performed this exercise many times, in C#, Smalltalk, and other languages. He's been searching for a tidy and elegant solution, one which makes the rules of bowling as clear as possible.

In the past, though, Jeffries has been a bit skeptical of Haskell implementations of bowling:

> *The recursive [Haskell] solution, however, is questionable on more fundamental grounds. A game of bowling consists of ten frames, not less or more, and the "ten-ness" of the game is not represented in the recursive solutions at all. Even if we let that slide, the recursive solutions make it a bit hard to understand what's going on.*

Let's see if we can do better. No knowledge of bowling is required–if we do this right, our program should be at least as clear as an English-language version of the rules.

Along the way, we'll encounter lazy lists, an interesting recursion combinator, and Hoogle, the Haskell search engine.

**Eric Kidd**

https://github.com/emk

2007

https://www.randomhacks.net/2007/04/28/bowling-in-haskell/

Eric Kidd

Do you have a **more elegant solution**? Please feel free to **share it**!

I think **Jeffries was right** when he said that:

A **game** of **bowling** consists of **ten frames**, **not less or more**, and the **"ten-ness" of the game** is **not represented** in the **recursive solutions** at all.

While I'm sure it's possible to write a (**correct**) **recursive bowling program** that doesn't mention the **number 10**, you would have to add more **complexity** to **scoreFrame**, or add **sentinel values** at the end of the game.

So that's why I went with **take 10**—it's the **shortest way** to build **"ten-ness"** into the program, and it keeps any trickiness out of **scoreFrame** or **reduce**.

```haskell
-- Pins knocked down by each ball.
type Balls = [Int]

-- Number of points scored.
type Score = Int

-- Given the number of pins knocked down
-- by each ball, score the game.
scoreGame :: Balls -> Score
scoreGame balls =
  sum (take 10 (reduce scoreFrame balls))

-- A slightly messier cousin of 'foldr'.
reduce :: ([a] -> (b,[a])) -> [a] -> [b]
reduce f ys = x:reduce f ys'
  where (x,ys') = f ys

-- Score one frame of a bowling game,
-- and calculate the starting point for
-- the next frame.
scoreFrame :: Balls -> (Score, Balls)
scoreFrame (x1:    y1:y2:ys) | x1 == 10 =
  (x1+y1+y2, y1:y2:ys)  -- Strike
scoreFrame (x1:x2: y1:ys) | x1+x2 == 10 =
  (x1+x2+y1, y1:ys)     -- Spare
scoreFrame (x1:x2: ys) =
  (x1+x2,    ys)        -- Open frame
```

1..10 ten-ness

**Eric**'s **`recursion combinator`**, which he mentioned on the previous slide and explains on the next slide.

In practice it turns out not to be a problem that **scoreFrame**'s **pattern matching** is **non-exhaustive** in that it doesn't cover the **empty list** and **singleton list**, because by **`taking`** 10 **frame scores**, **scoreGame** ensures that **scoreFrame** is never called with such lists.

Version 7 - **Haskell** – **domain-rich** - **Eric Kidd**

https://web.archive.org/web/20080907225526/http://www.randomhacks.net/darcs/bowling/Bowling.hs

```haskell
-- Pins knocked down by each ball.
type Balls = [Int]

-- Number of points scored.
type Score = Int

-- Given the number of pins knocked down
-- by each ball, score the game.
scoreGame :: Balls -> Score
scoreGame balls =
  sum (take 10 (reduce scoreFrame balls))

-- A slightly messier cousin of 'foldr'.
reduce :: ([a] -> (b,[a])) -> [a] -> [b]
reduce f ys = x:reduce f ys'
  where (x,ys') = f ys

-- Score one frame of a bowling game,
-- and calculate the starting point for
-- the next frame.
scoreFrame :: Balls -> (Score, Balls)
scoreFrame (x1:    y1:y2:ys) | x1 == 10 =
  (x1+y1+y2, y1:y2:ys)  -- Strike
scoreFrame (x1:x2: y1:ys) | x1+x2 == 10 =
  (x1+x2+y1, y1:ys)     -- Spare
scoreFrame (x1:x2: ys) =
  (x1+x2,    ys)        -- Open frame
```

1. In bowling, we roll balls down a lane, trying to knock down pins. If we know how many pins we knock down with each ball, we can compute the final score.

4. We need to turn **scoreFrame** into a recursive function. If we apply **reduce** to **scoreFrame**, we get the recursive function we're looking for.

3. Given a function *f* and a list *ys*, apply *f* to *ys* to get *x* and *ys'*. Then build a list starting with *x*. To get the rest of the list, call ourselves recursively on *ys'*. Notice that we never bother to specify a **base case**! Because **Haskell** is a **lazy language**, we only compute as much of the list as we need.

2. To score an individual frame, we need to do two things:
   1. calculate the score for our frame
   2. figure out where the next frame starts

   Our scoring function returns both pieces of information

Eric Kidd

As **Eric** points out on the previous slide, when he explains the **reduce function**…

```
reduce :: ([a] -> (b,[a])) -> [a] -> [b]
reduce f ys = x:reduce f ys’
  where (x,ys') = f ys
```

“Notice that we never bother to specify a **base case**! Because **Haskell** is a **lazy language**, we only compute as much of the list as we need.”

The **reduce** function is **lazy**:

- Regardless of the number of **a**s that **reduce** is passed, which could even be **infinite**, the **number** of **a**s that it **consumes** is **governed** by the **number** of **b**s that it has to **produce**, which is the number **consumed** by the **caller** of **reduce**
- E.g. regardless of the number of **balls** that **reduce** is passed, which could even be **infinite**, the **number** of **balls** that it **consumes** is **governed** by the **number** of **frame scores** that it has to **produce**, which is the number **consumed** by **scoreGame**…

```
scoreGame :: Balls -> Score
scoreGame balls =
  sum (take 10 (reduce scoreFrame balls))
```

…and which is the **number** of **frame scores** that **scoreGame** **’takes’** from the **result** of the **reduce function**.

Eric's Haskell program was the first one in this series to use a two-phase approach:

- in the first phase it computes the score for individual frames
- in the second phase it computes the game's score by adding up the scores of the game's frames

```
scoreGame :: Balls -> Score
scoreGame balls =
  sum (take 10 (reduce scoreFrame balls))
```

See next slide for Eric Kidd's tests. He has included ones seen in Ron Jeffries' blog posts (highlighted in blue). The bottom two tests are the ones provided by Pit. The second of them verifies ten-ness.

Eric Kidd

**Jeffries** actually had a **very good point**, regarding the **lack of "ten-ness"** in the **earlier Haskell solutions**.

In particular, he described **two test cases** that **break** the **first Haskell implementation**:

```
(gutter 16++[0,0, 10,2,3], 15),
(gutter 16++[10, 2,3], 20),
```

If you're **processing frames recursively** (**with no frame counter**), you can't tell these **two cases apart**. In my version, I use **take 10** to ensure that the program processes exactly **10 frames**.

```
-- Lists of balls, and the desired scores.
testData :: [(Balls,Score)]
testData = [
  (  10:perfect 11, 300), -- Strike
  ( 9:1:perfect 11, 290), -- Spare
  ( 8:1:perfect 11, 279), -- Open frame
  ( 8:1: 9:1:perfect 10,  269),
  ( 7:2: 6:3:perfect 10,  258),
  (perfect 9++[10,10, 0], 290),
  (perfect 9++[10, 5, 5], 285),
  (perfect 9++[10, 0,10], 280),
  (perfect 9++[10, 0, 0], 270),
  (perfect 9++[ 9, 0],    267),
  (perfect 9++[ 9, 1, 5], 274),
  -- Two from http://www.xprogramming.com/xpmag/dbcHaskellBowling.htm
  ([10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10], 200),
  ([5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5], 200),
  -- Two from http://www.xprogramming.com/xpmag/dbcRecurringDrama.htm
  ([0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,2,3], 15),
  ([0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,  2,3], 20),
  (gutter 20, 0)]          -- Missed all
```

1..10
ten-ness

```
-- A list of 'n' perfect balls.
perfect :: Num a => Int -> [a]
perfect n = replicate n 10

-- A list of 'n' gutter balls.
gutter :: Num a => Int -> [a]
gutter n = replicate n 0

-- Construct a unit test asserting that
-- we calculate the expected score.
testFromData :: ([Int], Score) -> Test
testFromData (balls, score) =
  ("Scoring " ++ show balls) ~:
    score ~=? scoreGame balls

-- Build a list of tests and run it.
tests :: Test
tests = test (map testFromData testData)
```

Pit

On the next slide we **translate** **Eric**'s **program** into **Scala**. Because **his program** exploits the **laziness** of **Haskell** **lists**, in our **Scala** **translation** of the **program** we need to use **lazy lists**.

The subsequent slide shows the **program**'s **test suite**, which is the same **suite** used for the **Scala** **translation** of **Tom**'s **program**, but modified to use **lazy lists**.

```haskell
-- Pins knocked down by each ball.
type Balls = [Int]

-- Number of points scored.
type Score = Int

-- Given the number of pins knocked down
-- by each ball, score the game.
scoreGame :: Balls -> Score
scoreGame balls =
  sum (take 10 (reduce scoreFrame balls))

-- A slightly messier cousin of 'foldr'.
reduce :: ([a] -> (b,[a])) -> [a] -> [b]
reduce f ys = x:reduce f ys'
  where (x,ys') = f ys

-- Score one frame of a bowling game,
-- and calculate the starting point for
-- the next frame.
scoreFrame :: Balls -> (Score, Balls)
scoreFrame (x1:    y1:y2:ys) | x1 == 10 =
  (x1+y1+y2, y1:y2:ys)  -- Strike
scoreFrame (x1:x2: y1:ys) | x1+x2 == 10 =
  (x1+x2+y1, y1:ys)     -- Spare
scoreFrame (x1:x2: ys) =
  (x1+x2,    ys)        -- Open frame
```

```scala
// Pins knocked down by each ball
type Balls = LazyList[Int]

// Number of points scored.
type Score = Int

// Given the number of pins knocked down
// by each ball, score the game.
def scoreGame(balls: LazyList[Int]): Int =
  reduce(scoreFrame, balls)
    .take(10).sum

// A slightly messier cousin of 'foldright'.
def reduce[A,B](f:LazyList[A]=>(B,LazyList[A]),as:LazyList[A]):LazyList[B] =
  val (x, ys) = f(as)
  x#::reduce(f, ys)

// Score one frame of a bowling game,
// and calculate the starting point for
// the next frame.
def scoreFrame(balls: Balls): (Score, Balls) = balls match
  case x1#::y1#::y2#::ys if x1 == 10 =>
    (x1+y1+y2, y1#::y2#::ys) // Strike
  case x1#::x2#::y1#::ys if x1+x2 == 10 =>
    (x1+x2+y1, y1#::ys)      // Spare
  case x1#::x2#::ys =>
    (x1+x2,    ys)           // Open frame
```

Version 7 - **Haskell** – **domain-rich** - **Eric Kidd**

**Scala** translation of 'Version 7 - **Haskell** – **domain-rich** - **Eric Kidd** '

Robert C. Martin

Ron Jeffries

Pit

```scala
import org.scalatest.funsuite.AnyFunSuite
import scala.util.{Failure, Try}

class BowlingTest extends AnyFunSuite {

  test("gutter"):
    assert(0 == scoreGame(LazyList.fill(20)(0)))

  test("allOnes"):
    assert(20 == scoreGame(LazyList.fill(20)(1)))

  test("oneSpare"):
    assert(24 == scoreGame(5#::5#::7#::LazyList.fill(17)(0)))

  test("oneStrike"):
    assert(20 == scoreGame(10#::2#::3#::LazyList.fill(16)(0)))

  test("perfect game"):
    assert(300 == scoreGame(LazyList.fill(12)(10)))

  ////////////////////////////////////////

  test("alternating strike spare"):
    assert(200 == scoreGame(LazyList(10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10)))

  test("alternating spare strike"):
    assert(200 == scoreGame(LazyList(5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5)))

  test("pit strike final frame"):
    assert(15 == scoreGame(LazyList(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,2,3)))

  test("pit strike ninth frame"):
    assert(20 == scoreGame(LazyList(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 2,3)))

  ////////////////////////////////////////

  test("ill-formed sequence of rolls"): assert(
    Failure(IllegalArgumentException("ill-formed sequence of rolls")).toString
    == Try{ scoreGame(LazyList.empty) }.toString)
}
```

✅ 1..10 ten-ness

And now an **interesting plot twist**, courtesy of **Dan**, who in a comment on **Eric**'s blog post (in which he identified himself using only his first name), pointed out that the **need** for **Eric**'s **reduce function** can be **eliminated** by using **Haskell**'s `**unfoldr**` **function** (see right-hand side for pseudocode).

$$
\begin{aligned}
&unfoldr && :: (\alpha \to \textit{Maybe}\ (\beta, \alpha)) \to \alpha \to \textit{List}\ \beta \\
&unfoldr\ f\ u = \textbf{case}\ f\ u\ \textbf{of} \\
&\qquad \textit{Nothing} && \to \textit{Nil} \\
&\qquad \textit{Just}\ (x, v) && \to \textit{Cons}\ x\ (unfold'\ f\ v)
\end{aligned}
$$

unfoldr unfold

Dan wrote on Apr 28, 2007:

As it happens, reduce is in the standard library, although its type is somewhat disguised. The function is:

```
unfoldr :: (a -> Maybe (b, a)) -> a -> [b]
```

Of course, scoreFrame doesn't wrap its results in Maybe, and in fact, many of the standard functions that would be useful with unfoldr (splitAt, divMod…) don't either. However, we can write a function to augment these with a stopping predicate:

```
stopOn :: (a -> Bool) -> (a -> b) -> a -> Maybe b
stopOn p f a =
  if p a
    then Nothing
    else Just (f a)
```

Dan

Now, scoreGame becomes:

```
scoreGame balls =
  sum . take 10
    $ unfoldr (stopOn null scoreFrame) balls
```

Of course, defining stopOn is just about as much work as defining reduce, so I suppose it's a bit of a wash. It might be handy if stopOn were in the standard library for just such an occasion (although stopOn probably isn't the best name, but I haven't thought of a better one yet; my initial thought was 'unless', but that's taken already), but to my knowledge, it isn't.

(Apologies for the code formatting. I couldn't figure out which tag to use to get it to come out like yours (and the code tags were causing line wrapping).)

Haskell

**base-4.21.0.0: Core data structures and operations**

# Data.List

| | |
|---|---|
| **Copyright** | (c) The University of Glasgow 2001 |
| **License** | BSD-style (see the file libraries/base/LICENSE) |
| **Maintainer** | libraries@haskell.org |
| **Stability** | stable |
| **Portability** | portable |
| **Safe Haskell** | Safe |
| **Language** | Haskell2010 |

## Unfolding

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]                    # Source
```

> The `unfoldr` function is a `dual' to `foldr`: while `foldr` reduces a list to a summary value, `unfoldr` builds a list from a seed value. The function takes the element and returns `Nothing` if it is done producing the list or returns `Just (a,b)`, in which case, `a` is a prepended to the list and `b` is used as the next element in a recursive call. For example,

The function passed to **unfoldr** takes a **seed value** that it uses either to **generate Nothing** or to **generate** both a **new result item** and a **new seed value**.

# IterableFactory

scala.collection.IterableFactory

Scala 3 3.7.0

```
def unfold[A, S](init: S)(f: S => Option[(A, S)]): CC[A]
```

Produces a collection that uses a function `f` to produce elements of type `A` and update an internal state of type `S`.

**Type parameters**

| | |
|---|---|
| A | Type of the elements |
| S | Type of the internal state |

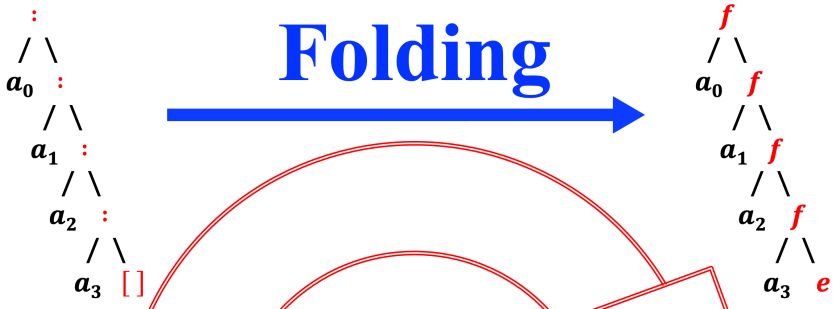
**Value parameters**

| | |
|---|---|
| f | Computes the next element (or returns `None` to signal the end of the collection) |
| init | State initial value |

**Attributes**

| | |
|---|---|
| Returns | a collection that produces elements using `f` until `f` returns `None` |
| Source | Factory.scala |

The function passed to **unfold** takes a **state value** that it uses either to **generate None** or to **generate** both a **new result item** and a **new state value**.




CHEAT-SHEET

#9

List Unfolding
$unfold$ as the Computational Dual of $fold$
and how $unfold$ relates to $iterate$

If you could do with an **introduction** to (or **refresher** on) the **unfold function**, maybe consider checking out one of these two decks.

# List Unfolding

## $unfold$ as the Computational Dual of $fold$ and how $unfold$ relates to $iterate$

Haskell

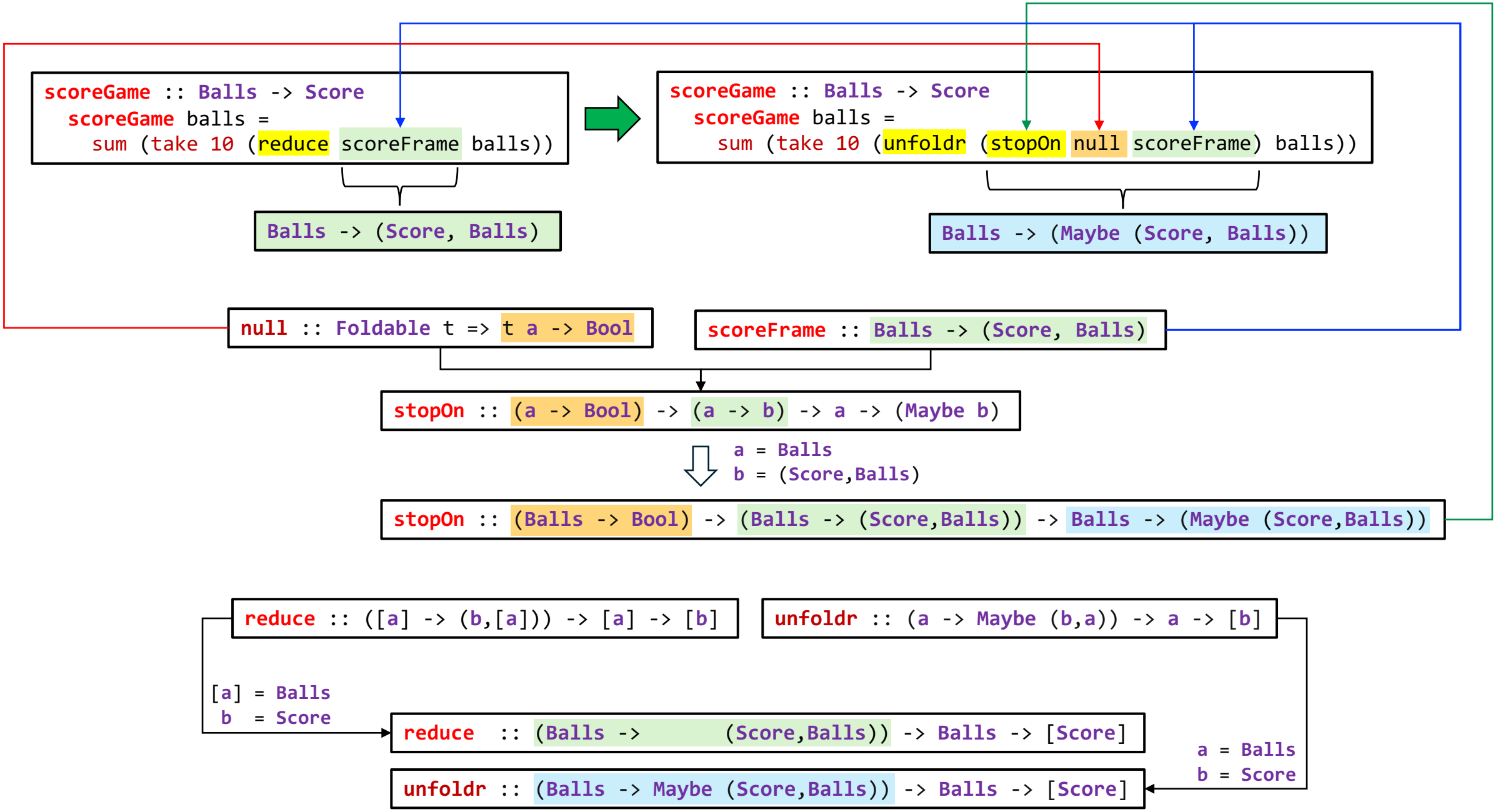
$fold$

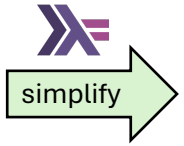
$unfold$

Scala

See next slide for a **diagram** that helps us see why in **Eric**'s **Haskell** program it is possible to **replace** the **reduce function** with the **unfoldr** and **stopOn functions**.

See the subsequent four slides for both **Eric**'s **program**, and its **Scala translation**, after making such a **replacement**.


scoreGame :: Balls -> Score
scoreGame balls =
sum (take 10 (reduce scoreFrame balls))
Balls -> (Score, Balls)
scoreGame :: Balls -> Score
scoreGame balls =
sum (take 10 (unfoldr (stopOn null scoreFrame) balls))
Balls -> (Maybe (Score, Balls))
null :: Foldable t => t a -> Bool
scoreFrame :: Balls -> (Score, Balls)
stopOn :: (a -> Bool) -> (a -> b) -> a -> (Maybe b)
a = Balls
b = (Score,Balls)
stopOn :: (Balls -> Bool) -> (Balls -> (Score,Balls)) -> Balls -> (Maybe (Score,Balls))
reduce :: ([a] -> (b,[a])) -> [a] -> [b]
unfoldr :: (a -> Maybe (b,a)) -> a -> [b]
[a] = Balls
b = Score
reduce :: (Balls -> (Score,Balls)) -> Balls -> [Score]
unfoldr :: (Balls -> Maybe (Score,Balls)) -> Balls -> [Score]
a = Balls
b = Score




| | | |
|---|---|---|
| `-- Pins knocked down by each ball.`<br>`type Balls = [Int]`<br><br>`-- Number of points scored.`<br>`type Score = Int`<br><br>`-- Given the number of pins knocked down`<br>`-- by each ball, score the game.`<br>`scoreGame :: Balls -> Score`<br>`scoreGame balls =` | = | `-- Pins knocked down by each ball.`<br>`type Balls = [Int]`<br><br>`-- Number of points scored.`<br>`type Score = Int`<br><br>`-- Given the number of pins knocked down`<br>`-- by each ball, score the game.`<br>`scoreGame :: Balls -> Score`<br>`scoreGame balls =` |
| `  sum (take 10 (reduce scoreFrame balls))` | <> | `  sum (take 10 (unfoldr (stopOn null scoreFrame) balls))` |
| | = | |
| `-- A slightly messier cousin of 'foldr'.`<br>`reduce :: ([a] -> (b,[a])) -> [a] -> [b]`<br>`reduce f ys = x:reduce f ys'`<br>`  where (x,ys') = f ys` | +- | |
| | = | |
| | <> | `stopOn :: (a -> Bool) -> (a -> b) -> a -> (Maybe b)`<br>`stopOn p f a = if p a then Nothing else Just (f a)` |
| `-- Score one frame of a bowling game,`<br>`-- and calculate the starting point for`<br>`-- the next frame.`<br>`scoreFrame :: Balls -> (Score, Balls)`<br>`scoreFrame (x1:    y1:y2:ys) \| x1 == 10 =`<br>`  (x1+y1+y2, y1:y2:ys)  -- Strike`<br>`scoreFrame (x1:x2: y1:ys) \| x1+x2 == 10 =`<br>`  (x1+x2+y1, y1:ys)     -- Spare`<br>`scoreFrame (x1:x2: ys) =`<br>`  (x1+x2,    ys)        -- Open frame` | = | `-- Score one frame of a bowling game,`<br>`-- and calculate the starting point for`<br>`-- the next frame.`<br>`scoreFrame :: Balls -> (Score, Balls)`<br>`scoreFrame (x1:    y1:y2:ys) \| x1 == 10 =`<br>`  (x1+y1+y2, y1:y2:ys)  -- Strike`<br>`scoreFrame (x1:x2: y1:ys) \| x1+x2 == 10 =`<br>`  (x1+x2+y1, y1:ys)     -- Spare`<br>`scoreFrame (x1:x2: ys) =`<br>`  (x1+x2,    ys)        -- Open frame` |

**Eric Kidd**'s original **Haskell** program

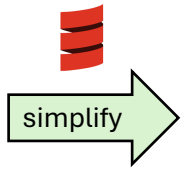
simplification suggested by **Dan**

```haskell
-- Pins knocked down by each ball.
type Balls = [Int]

-- Number of points scored.
type Score = Int

-- Given the number of pins knocked down
-- by each ball, score the game.
scoreGame :: Balls -> Score
scoreGame balls =
  sum (take 10 (unfoldr (stopOn null scoreFrame) balls)) †

stopOn :: (a -> Bool) -> (a -> b) -> a -> (Maybe b)
stopOn p f a = if p a then Nothing else Just (f a)

-- Score one frame of a bowling game,
-- and calculate the starting point for
-- the next frame.
scoreFrame :: Balls -> (Score, Balls)
scoreFrame (x1:    y1:y2:ys) | x1 == 10 =
  (x1+y1+y2, y1:y2:ys)  -- Strike
scoreFrame (x1:x2: y1:ys) | x1+x2 == 10 =
  (x1+x2+y1, y1:ys)     -- Spare
scoreFrame (x1:x2: ys) =
  (x1+x2,    ys)        -- Open frame
```

Dan

Version 7b – **Dan**'s variant of `Version 7 **Haskell** – **domain-rich** - **Eric Kidd**` - replaces **reduce** with **unfoldr**

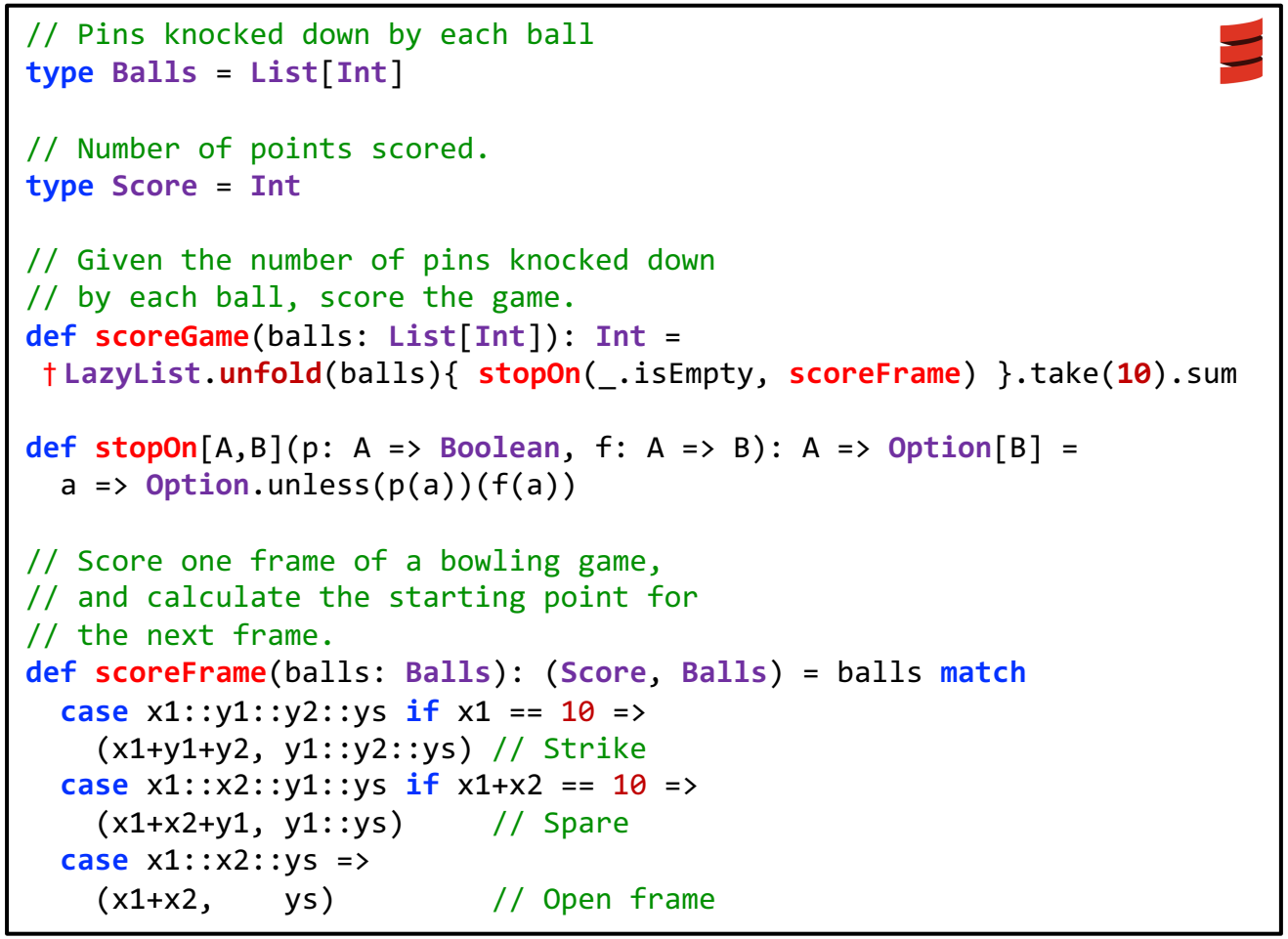
† **Dan** used operators . (function composition) and $ (function application) in the body of **scoreGame**, most likely to reduce the number of parentheses used in the body, but we have preferred to remove the operators in favour of extra parentheses, to make the body easier to understand for any readers less familiar with **Haskell**.




| | | |
|---|---|---|
| `// Pins knocked down by each ball` | = | `// Pins knocked down by each ball` |
| `type Balls = LazyList[Int]` | <> | `type Balls = List[Int]` |
| `// Number of points scored.`<br>`type Score = Int`<br><br>`// Given the number of pins knocked down`<br>`// by each ball, score the game.` | = | `// Number of points scored.`<br>`type Score = Int`<br><br>`// Given the number of pins knocked down`<br>`// by each ball, score the game.` |
| `def scoreGame(balls: LazyList[Int]): Int =`<br>`  reduce(scoreFrame, balls).take(10).sum` | <> | `def scoreGame(balls: List[Int]): Int =`<br>`  LazyList.unfold(balls){ stopOn(_.isEmpty, scoreFrame) }.take(10).sum` |
| | = | |
| `// A slightly messier cousin of 'foldr'.`<br>`def reduce[A,B](f: LazyList[A] => (B, LazyList[A]), as: LazyList[A]): LazyList[B] =`<br>`  val (x, ys) = f(as)`<br>`  x#::reduce(f,ys)` | <> | `def stopOn[A,B](p: A => Boolean, f: A => B): A => Option[B] =`<br>`  a => Option.unless(p(a))(f(a))` |
| `def scoreFrame(balls: Balls): (Score, Balls) = balls match` | = | `def scoreFrame(balls: Balls): (Score, Balls) = balls match` |
| `  case x1#::y1#::y2#::ys if x1 == 10 =>`<br>`  (x1+y1+y2, y1#::y2#::ys) // Strike`<br>`  case x1#::x2#::y1#::ys if x1+x2 == 10 =>`<br>`  (x1+x2+y1, y1#::ys) // Spare`<br>`  case x1#::x2#::ys =>` | <> | `  case x1::y1::y2::ys if x1 == 10 =>`<br>`  (x1+y1+y2, y1::y2::ys) // Strike`<br>`  case x1::x2::y1::ys if x1+x2 == 10 =>`<br>`  (x1+x2+y1, y1::ys) // Spare`<br>`  case x1::x2::ys =>` |
| `  (x1+x2, ys) // Open frame` | = | `  (x1+x2, ys) // Open frame` |

**Scala** translation of **Eric Kidd**'s original **Haskell** program

**Scala** equivalent of simplification suggested by **Dan**



```scala
// Pins knocked down by each ball
type Balls = List[Int]

// Number of points scored.
type Score = Int

// Given the number of pins knocked down
// by each ball, score the game.
def scoreGame(balls: List[Int]): Int =
 †LazyList.unfold(balls){ stopOn(_.isEmpty, scoreFrame) }.take(10).sum

def stopOn[A,B](p: A => Boolean, f: A => B): A => Option[B] =
  a => Option.unless(p(a))(f(a))

// Score one frame of a bowling game,
// and calculate the starting point for
// the next frame.
def scoreFrame(balls: Balls): (Score, Balls) = balls match
  case x1::y1::y2::ys if x1 == 10 =>
    (x1+y1+y2, y1::y2::ys) // Strike
  case x1::x2::y1::ys if x1+x2 == 10 =>
    (x1+x2+y1, y1::ys)     // Spare
  case x1::x2::ys =>
    (x1+x2,    ys)         // Open frame
```

**Scala** translation of Version 7b – **Dan**'s variant of `Version 7 **Haskell** – **domain-rich** - **Eric Kidd**` - replaces **reduce** with **unfoldr**

† While **Haskell**'s lists are **lazy**, **Scala**'s lists are not, so we have to use the **unfold function** of **LazyList** (rather than that of **List**). What happens if we don't do that is that **scoreFrame** eventually gets called with a **singleton list**, which it cannot handle because (as mentioned earlier on) **scoreFrame**'s pattern matching is **not exhaustive** (it doesn't handle **singleton** and **empty lists**). E.g. the following test fails with `scala.MatchError: List(10)`.

```scala
test("alternating strike spare") { assert(200 == scoreGame(List(10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10))) }
```

The **next program** we are going to look at also **exploits laziness** and adopts a **multi-phase approach**.

It is a **Clojure program** written by **Stuart Halloway** in 2009.

Thanks to its **functions**, their **naming**, and their **comments**, the **program** is very much a **domain-rich** one.

See **next slide** for where to find the program, the **subsequent slide** for the program's code (accompanied by some explanations), and the **slide after that** for the program's **tests suite** (to which I have added the usual tests).

https://github.com/stuarthalloway/clojure-bowling

stuarthalloway / clojure-bowling

Code Issues Pull requests Agents Actions

clojure-bowling Public

Clojure

README

This is an implementation of the Uncle Bob's bowling scorer [1] in Clojure.

Note that this requires a post-1.0 version of Clojure and contrib to use the new 'are' syntax.

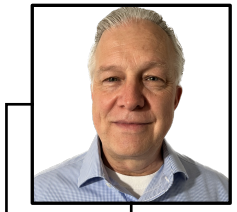
Required JARs a are in lib, and there are script to run a REPL or the tests in bin.

Stuart Halloway
http://blog.thinkrelevance.com

2009

**Stuart Halloway**
stuarthalloway

https://github.com/stuarthalloway/clojure-bowling



The **sequence** of **frames** returned by the **frames function** is **lazy**: only as many of its **elements** will be **computed** as will be **consumed** by **callers** of the **function**.

Just like the **recursive reduce function** in **Eric Kidd**'s program, this **recursive function** constructs and returns a **lazy sequence**, so it doesn't require a **base case**.

**three-phase approach**:

1. convert **rolls** into a **lazy sequence** of **frames** (collections of **rolls**)
2. compute **score** of each **frame**
3. compute **game score** by summing **frame scores**

```
(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

```
> (frames [10 2 3 4])
((10 2 3) (2 3) (4))        strike

> (frames [5 5 3 4])
((5 5 3) (3 4))             spare

> (frames [1 2 3 4])
((1 2) (3 4))               open frame
```

```
> (map score-frame `((10 2 3) (2 3) (4)))
(15 5 4)

> (map score-frame `((5 5 3) (3 4)))
(13 7)

> (map score-frame `((1 2) (3 4)))
(3 7)
```

```
> (score-game [10 2 3 4])
24

> (score-game [5 5 3 4])
20

> (score-game [1 2 3 4])
10
```

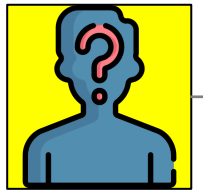
✅ 1..10 ten-ness

Version 8 - **Clojure** – **domain-rich** – **Stuart Halloway**

Stuart Halloway

Robert C. Martin

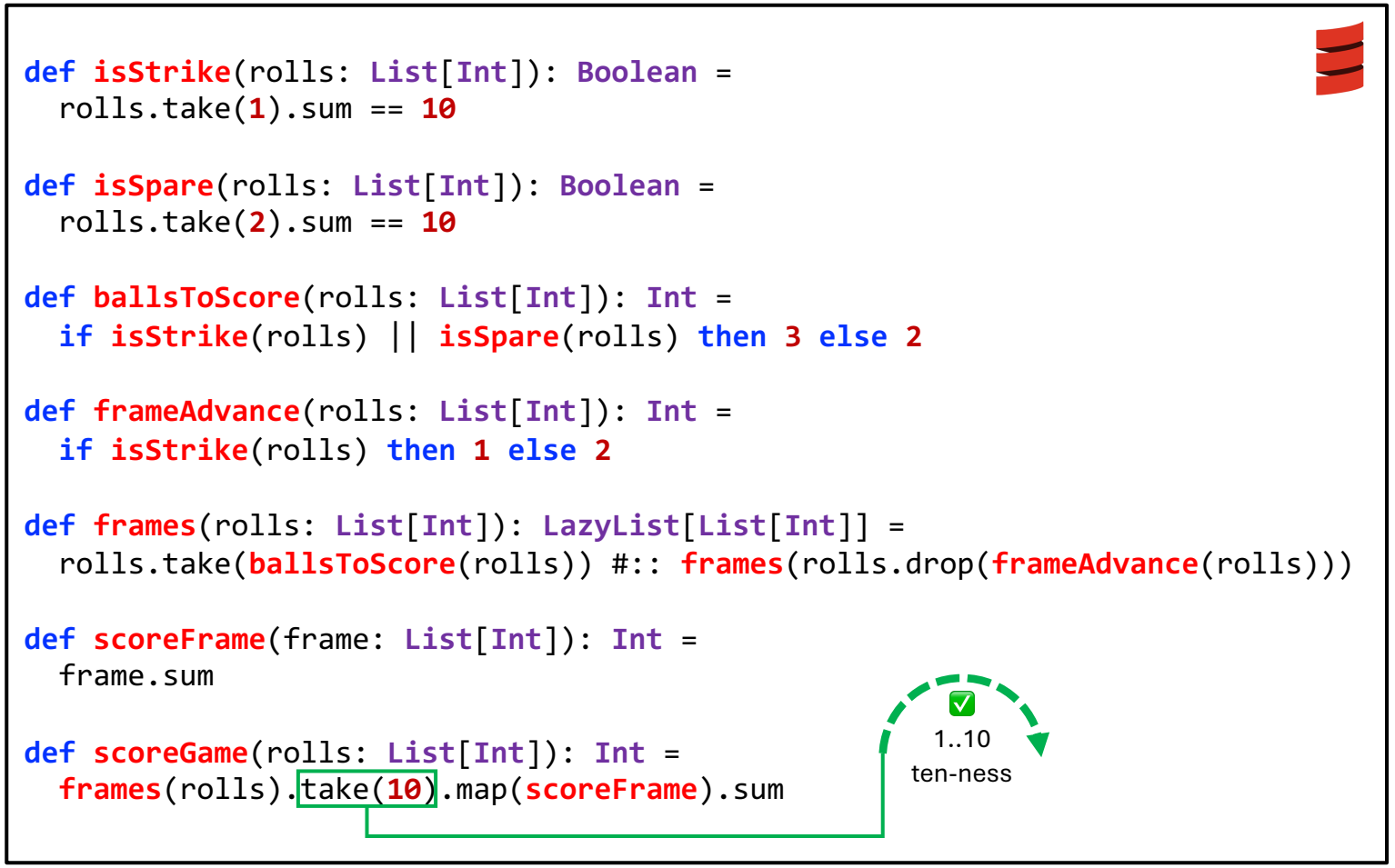
Ron Jeffries



Pit

```
(ns test.bowling-game
  (:use clojure.test)
  (:use bowling-game))

(deftest test-balls-to-score
  (are [description balls frames] (= balls (balls-to-score frames))
       "strike" 3 [10 10 10]
       "spare" 3 [5 5 10]
       "no mark" 2 [5 3 5]))

(deftest test-frame-advance
  (are [description advance frames] (= advance (frame-advance frames))
       "strike" 1 [10]
       "spare" 2 [5 5]
       "no mark" 2 [5 4]))

(deftest test-frames-for-various-games
  (are [description expected-frames game] (= expected-frames (take 10 (frames game)))
       "gutter game" (repeat 10 [0 0]) (repeat 0)
       "all ones" (repeat 10 [1 1]) (repeat 1)
       "all fives (spares)" (repeat 10 [5 5 5]) (repeat 5)
       "all tens (strikes)" (repeat 10 [10 10 10]) (repeat 10)
       "a partial game" [[1 2] [3 4]] [1 2 3 4]))

(deftest test-various-games
  (are [description expected-score game] (= expected-score (score-game game))
       "gutter game" 0 (repeat 0)
       "all ones" 20 (repeat 1)
       "one spare" 16 (concat [5 5 3] (repeat 0))
       "one strike" 24 (concat [10 3 4] (repeat 0))
       "perfect game" 300 (repeat 10)
       "alternating strike spare" 200 [10 5 5  10 5 5  10 5 5  10 5 5  10 5 5  10]
       "alternating spare strike" 200 [5 5  10 5 5  10 5 5  10 5 5  10 5 5  10 5 5]
       "pit strike final frame" 15 [0 0  0 0  0 0  0 0  0 0  0 0  0 0  0 0  0 0  10 2 3]
       "pit strike ninth frame" 20 [0 0  0 0  0 0  0 0  0 0  0 0  0 0  0 0  10  2 3]))
```

The next slide shows the **Scala translation** of **Stuart Halloway**'s **Clojure program**, and the subsequent one shows the **test suite** with which I **tested** the **translation**, which is practically the same one I used to test the **Scala translation** of **Tom**'s **Haskell program**.




Scala translation of `Version 8 - Clojure – domain-rich – Stuart Halloway`

Robert C. Martin

Ron Jeffries

Pit

```
import org.scalatest.funsuite.AnyFunSuite
import scala.util.{Failure, Try}

class BowlingTest extends AnyFunSuite {

  test("gutter"):
    assert(0 == scoreGame(List.fill(20)(0)))

  test("allOnes"):
    assert(20 == scoreGame(List.fill(20)(1)))

  test("oneSpare"):
    assert(24 == scoreGame(5::5::7::List.fill(17)(0)))

  test("oneStrike"):
    assert(20 == scoreGame(10::2::3::List.fill(16)(0)))

  test("perfect game"):
    assert(300 == scoreGame(List.fill(12)(10)))

  ////////////////////////////////////////

  test("alternating strike spare"):
    assert(200 == scoreGame(List(10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10)))

  test("alternating spare strike"):
    assert(200 == scoreGame(List(5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5)))

  test("pit strike final frame"):
    assert(15 == scoreGame(List(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10,2,3)))

  test("pit strike ninth frame"):
    assert(20 == scoreGame(List(0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 2,3)))

  ////////////////////////////////////////

  test("ill-formed sequence of rolls"): assert(
    Failure(IllegalArgumentException("ill-formed sequence of rolls")).toString
    == Try{ scoreGame(List.empty) }.toString)
}
```

The next slide shows the **Haskell** **translation** of **Stuart Halloway**'s **Clojure** **program**, and the subsequent one shows the **test suite** with which I **tested** the **translation**.

```haskell
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)))
```


**Haskell** translation of `Version 8 - **Clojure** – **domain-rich** – **Stuart Halloway**`

Here is **Tom Moertel**'s **updated test suite**, but with the following added:

- The tests seen in **Robert Martin**'s **Java** version of the program
- Two tests seen in **Ron Jeffries**' **Java** version of the program
- **Pit**'s two tests (added by **Ron Jeffries** to his corrected **Java** program), the second of which verifies **ten-ness**

```
import Test.HUnit

tests = test
    [ "gutters"       ~: score  (rep 20  0)          ~?=   0
    , "ones"          ~: score  (rep 20  1)          ~?=  20
    , "fives"         ~: score  (rep 22  5)          ~?= 150
    , "strikes"       ~: score  (rep 12 10)          ~?= 300
    , "1 + gutters"   ~: score  (1 : rep 19 0)       ~?=   1
    , "first spare"   ~: score  (5:5:5 : rep 17 0)   ~?=  20
    , "first strike"  ~: score  (10:5:5 : rep 17 0)  ~?=  30
    , "last spare"    ~: score (5:5:5 : rep 18 0)   ~?=  15
    , "last strike"   ~: rscore (5:5:10 : rep 18 0)  ~?=  20

    , "gutter game"  ~: score (rep 20  0)           ~?=   0
    , "all ones"     ~: score (rep 20  1)           ~?=   20
    , "one spare"    ~: score (5:5:7 : rep 17  0)   ~?=   24
    , "one strike"   ~: score (10:2:3 : rep 16  0)  ~?=   20
    , "perfect-game" ~: score (rep 12  10)          ~?=   300
    , "alternating strike spare" ~: score [10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10] ~?= 200
    , "alternating spare strike" ~: score [5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5, 10,5,5] ~?= 200
    , "strike final frame"       ~: score [0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 2,3] ~?= 15
    , "strike ninth frame"       ~: score [0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 10, 2,3] ~?= 20
    ]
  where
    rep    = replicate
    score  = scoreGame
    rscore = score . reverse  -- Scores a reversed list of rolls.
```

Robert Martin

Ron Jeffries

Pit

To conclude part two, the next slide shows the **Scala** and **Haskell** translations of **Stuart Halloway**'s **Clojure** program **side by side**.

```scala
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
```

**Scala** translation of `Version 8 - **Clojure** – **domain-rich** – **Stuart Halloway**`

```haskell
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)))
```

**Haskell** translation of `Version 8 - **Clojure** – **domain-rich** – **Stuart Halloway**`

That’s all for part two.

I hope you liked it.

See you in part three.