Fibonacci Function Gallery

Part 2

- Infinite Stream with Explicit Generation
- Infinite Stream with Implicit Definition
- Infinite Stream with Unfolding
- Infinite Stream with Iteration
- Infinite Stream with Scanning

Scheme

Scala

Haskell

Clojure

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

In part one of this series we looked at six implementations of the **Fibonacci** function. If you are interested in a quick recap of those functions, you can find it on the next slide, otherwise feel free to skip it.

Fibonacci Function Gallery

Part 1

- **Naïve Recursion**
- **Efficient Recursion** with **Tupling**
- **Tail Recursion** with **Accumulation**
- **Tail Recursion** with **Folding**
- **Stack-safe Recursion** with **Trampolining**

Scala

Haskell

Clojure

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

Implementations explored so far

1
```scala
def fib(i: Int): BigInt = i match
  case 0 => 0
  case 1 => 1
  case _ => fib(i - 1) + fib(i - 2)
```

2
```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) = i match
  case 0 => (0, 1)
  case _ => fibtwo(i - 1) match
    { case (a, b) => (b, a + b) }
```

3
```scala
def fib(i: Int): BigInt =
  tailFib(0, 1, i)

def tailFib(a: BigInt, b: BigInt, i: Int): BigInt =
  i match
    case 0 => a
    case _ => tailFib(b, a + b, i - 1)
```

4
```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) =
  (1 to i).foldLeft(BigInt(0), BigInt(1))
    { case ((a, b), _) => (b, a + b) }
```

5
```scala
def fib(i: Int): Eval[BigInt] = i match
  case 0 => Eval.now(0)
  case 1 => Eval.now(1)
  case _ =>
   for
     a <- Eval.defer(fib(i - 1))
     b <- fib(i - 2)
   yield a + b
```

6
```scala
def fib(i: Int): IO[BigInt] = i match
  case 0 => IO.pure(0)
  case 1 => IO.pure(1)
  case _ =>
   for
     a <- IO.defer(fib(i - 1))
     b <- fib(i - 2)
   yield a + b
```

1
version #1 (**naïve**)
- not **tail-recursive** (not **stack-safe**)
- **exponential** time complexity
- **linear** stack frame depth

2
version #2 (**tupling**-based)
- not **tail-recursive** (not **stack-safe**)
- **linear** time complexity
- **linear** stack frame depth

3
version #3 (**accumulator**-based)
- **tail-recursive** (**stack-safe**)
- **linear** time complexity

4
version #4 (**left fold**-based)
- **non-recursive** (**stack-safe**)
- **linear** time complexity

5
version #5 (**stack-safe naïve** - **Eval**-based)
- not **tail-recursive** but **stack-safe**
- **exponential** time complexity

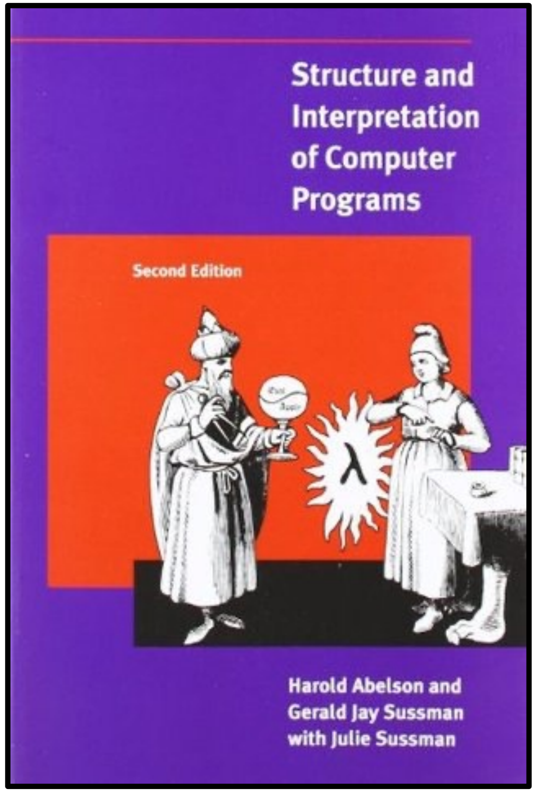
6
version #6 (**stack-safe naïve** - **IO**-based)
- not **tail-recursive** but **stack-safe**
- **exponential** time complexity

For the next two implementations of the **Fibonacci** function, we are going to turn to **Structure and Interpretation of Computer Programs** (**SICP**).

The following 10 slides are a lightning-fast, minimal refresher on (or intro to) the **building blocks** used in two implementations of the **Fibonacci sequence**.

The two implementations, written in the **Scheme** dialect of **Lisp**, don't use plain **sequences**, implemented with **lists**. Instead, they use **sequences** implemented with **streams**, i.e. **lazy** and possibly **infinite sequences**.

While a full introduction to **lists** and **streams** is outside the scope of this deck, let's learn (or review) just enough about them to be able to understand the two **Fibonacci sequence** implementations that we are interested in.



SICP

3.5 Streams

...

From an abstract point of view, a stream is simply a sequence.

However, we will find that the straightforward implementation of streams as lists (as in section 2.2.1) doesn't fully reveal the power of stream processing.

2.2.1 Representing Sequences

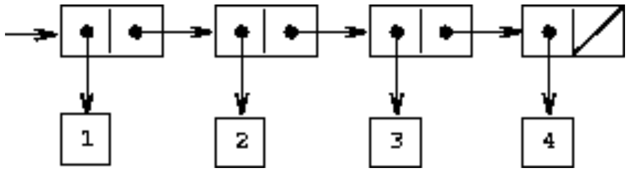


Figure 2.4: The sequence 1,2,3,4 represented as a chain of pairs.

One of the useful structures we can build with pairs is a sequence -- an ordered collection of data objects. There are, of course, many ways to represent sequences in terms of pairs. One particularly straightforward representation is illustrated in figure 2.4, where the sequence 1, 2, 3, 4 is represented as a chain of pairs. The car of each pair is the corresponding item in the chain, and the cdr of the pair is the next pair in the chain. The cdr of the final pair signals the end of the sequence by pointing to a distinguished value that is not a pair, represented in box-and-pointer diagrams as a diagonal line and in programs as the value of the variable nil. The entire sequence is constructed by nested cons operations:

```
(cons 1
      (cons 2
            (cons 3
                  (cons 4 nil))))
```

As an alternative, we introduce the technique of *delayed evaluation*, which enables us to represent very large (even infinite) sequences as streams. Stream processing lets us model systems that have state without ever using assignment or mutable data.

Structure and
Interpretation
of Computer Programs

Constructing a plain sequence (list)

λ

```
> (cons 1
        (cons 2
              (cons 3
                    (cons 4 nil))))

(1 2 3 4)

> (list 1 2 3 4)

(1 2 3 4)
```

```
> (cons 1
        (cons 2
              (cons 3
                    (cons 4 nil))))

(1 2 3 4)

> `(1 2 3 4)

(1 2 3 4)
```

```
> 1 :: (2 :: (3 :: (4 :: Nil)))

List(1, 2, 3, 4)

> List(1, 2, 3, 4)

List(1, 2, 3, 4)
```

```
> 1 : (2 : (3 : (4 : [])))

[1, 2, 3, 4]

> [1, 2, 3, 4]

[1, 2, 3, 4]
```

Selecting the head and tail of a plain sequence (list)

λ

```
> (define one-through-four (list 1 2 3 4))

> (car one-through-four)
1

> (cdr one-through-four)
(2 3 4)

> (car (cdr one-through-four))
2
```

```
> (def one-through-four `(1 2 3 4))

> (first one-through-four)
1

> (rest one-through-four)
(2 3 4)

> (first (rest one-through-four))
2
```

```
> val one_through_four = List(1, 2, 3, 4)

> one_through_four.head
1

> one_through_four.tail
List(2, 3, 4)

> one_through_four.tail.head
2
```

```
>  one_through_four = [1, 2, 3, 4]

> head one_through_four
1

> tail one_through_four
[2, 3, 4]

> head (tail one_through_four)
2
```

3.5.1 Streams Are Delayed Lists

As we saw in section 2.2.3, sequences can serve as standard interfaces for combining program modules. We formulated powerful abstractions for manipulating sequences, such as map, filter, and accumulate, that capture a wide variety of operations in a manner that is both succinct and elegant.

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))
```

$$
\begin{aligned}
&map &&:: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&map\ f\ [\,] &&= [\,] \\
&map\ f\ (x : xs) &&= f\ x : map\ f\ xs
\end{aligned}
$$

```
(define (filter predicate sequence)
        (cond ((null? sequence) nil)
              ((predicate (car sequence))
               (cons (car sequence)
                     (filter predicate (cdr sequence))))
              (else (filter predicate (cdr sequence)))))
```

$$
\begin{aligned}
&filter &&:: (\alpha \to Bool) \to [\alpha] \to [\alpha] \\
&filter\ p\ [\,] &&= [\,] \\
&filter\ p\ (x : xs) &&= \mathbf{if}\ p\ x \\
& && \quad\ \mathbf{then}\ x : filter\ p\ xs \\
& && \quad\ \mathbf{else}\ filter\ p\ xs
\end{aligned}
$$

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
```

$$
\begin{aligned}
&foldr &&:: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldr\ f\ e\ [\,] &&= e \\
&foldr\ f\ e\ (x : xs) &&= f\ x\ (foldr\ f\ e\ xs)
\end{aligned}
$$

Structure and Interpretation of Computer Programs

mapping, filtering, and folding a plain sequence (list)

λ

```scheme
(define (sum-even-fibs n)                                    †
  (fold-left +
             0
             (filter even?
                     (map fib
                          (iota (+ n 1) 0)))))
```

```scheme
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

```clojure
(defn sum-even-fibs [n]
  (reduce +
          0
          (filter even?
                  (map fib
                       (range 0 (inc n))))))
```

```clojure
(defn fib [n]
  (cond (= n 0) 0
        (= n 1) 1
        :else (+ (fib (- n 1))
                 (fib (- n 2)))))
```

```scala
def sum_even_fibs(n: Int): Int =
  List.range(0, n+1)
      .map(fib)
      .filter(isEven)
      .fold(0)(_+_)
```

```scala
def fib(n: Int): Int = n match
  case 0 => 0
  case 1 => 1
  case n => fib(n - 1) + fib(n - 2)
```

```scala
def isEven(n: Int): Boolean =
  n % 2 == 0
```

```haskell
sum_even_fibs :: Int -> Int
sum_even_fibs n =
  foldl (+)
        0
        (filter is_even
                (map fib
                     [0..n]))
```

```haskell
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

```haskell
is_even :: Int -> Bool
is_even n = (mod n 2) == 0
```

† FWIW, instead of using the hand-rolled **map**, **filter** and **accumulate** functions seen on the previous slide, we are using built-in functions **map**, **filter** and **fold-left**.

Unfortunately, if we represent sequences as lists, this elegance is bought at the price of severe inefficiency with respect to both the time and space required by our computations. When we represent manipulations on sequences as transformations of lists, our programs must construct and copy data structures (which may be huge) at every step of a process.

To see why this is true, let us compare two programs for computing the sum of all the prime numbers in an interval. The first program is written in standard iterative style:[53]

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

The second program performs the same computation using the sequence operations of section 2.2.3:

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime? (enumerate-interval a b))))
```

In carrying out the computation, the first program needs to store only the sum being accumulated. In contrast, the filter in the second program cannot do any testing until enumerate-interval has constructed a complete list of the numbers in the interval.

The filter generates another list, which in turn is passed to accumulate before being collapsed to form a sum.

Such large intermediate storage is not needed by the first program, which we can think of as enumerating the interval incrementally, adding each prime to the sum as it is generated.

[53] Assume that we have a predicate **prime?** (e.g., as in section 1.2.6) that tests for **primality**.

Structure and Interpretation of Computer Programs

The inefficiency in using lists becomes painfully apparent if we use the sequence paradigm to compute the second prime in the interval from 10,000 to 1,000,000 by evaluating the expression

```
(car (cdr (filter prime?
                  (enumerate-interval 10000 1000000))))
```

This expression does find the second prime, but the computational overhead is outrageous. We construct a list of almost a million integers, filter this list by testing each element for primality, and then ignore almost all of the result.

In a more traditional programming style, we would interleave the enumeration and the filtering, and stop when we reached the second prime.

Streams are a clever idea that allows one to use sequence manipulations without incurring the costs of manipulating sequences as lists.

With streams we can achieve the best of both worlds: We can formulate programs elegantly as sequence manipulations, while attaining the efficiency of incremental computation.

The basic idea is to arrange to construct a stream only partially, and to pass the partial construction to the program that consumes the stream.

If the consumer attempts to access a part of the stream that has not yet been constructed, the stream will automatically construct just enough more of itself to produce the required part, thus preserving the illusion that the entire stream exists.

In other words, although we will write programs as if we were processing complete sequences, we design our stream implementation to automatically and transparently interleave the construction of the stream with its use.

Structure and Interpretation of Computer Programs

On the surface, streams are just lists with different names for the procedures that manipulate them.

There is a constructor, cons-stream, and two selectors, stream-car and stream-cdr, which satisfy the constraints

```
(stream-car (cons-stream x y)) = x
(stream-cdr (cons-stream x y)) = y
```

There is a distinguishable object, the-empty-stream, which cannot be the result of any cons-stream operation, and which can be identified with the predicate stream-null?.[54]

Thus we can make and use streams, in just the same way as we can make and use lists, to represent aggregate data arranged in a sequence.

In particular, we can build stream analogs of the list operations from chapter 2, such as list-ref, map, and for-each:

```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))

(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc (stream-cdr s)))))
```

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
```

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))
```

…

[54] In the MIT implementation, the-empty-stream is the same as the empty list '(), and stream-null? is the same as null?.

Structure and Interpretation of Computer Programs

...

To make the stream implementation automatically and transparently interleave the construction of a stream with its use, we will arrange for the cdr of a stream to be evaluated when it is accessed by the stream-cdr procedure rather than when the stream is constructed by cons-stream.

...

As a data abstraction, streams are the same as lists. The difference is the time at which the elements are evaluated.

With ordinary lists, both the car and the cdr are evaluated at construction time. With streams, the cdr is evaluated at selection time.

Structure and Interpretation of Computer Programs

Next, we turn to infinite streams.

3.5.2 Infinite Streams

We have seen how to support the illusion of manipulating streams as complete entities even though, in actuality, we compute only as much of the stream as we need to access. We can exploit this technique to represent sequences efficiently as streams, even if the sequences are very long. What is more striking, we can use streams to represent sequences that are infinitely long. For instance, consider the following definition of the stream of positive integers:

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))

(define integers (integers-starting-from 1))
```

This makes sense because `integers` will be a pair whose car is 1 and whose cdr is a promise to produce the integers beginning with 2. This is an infinitely long stream, but in any given time we can examine only a finite portion of it. Thus, our programs will never know that the entire infinite stream is not there.

Using `integers` we can define other infinite streams, such as the stream of integers that are not divisible by 7:

```
(define (divisible? x y) (= (remainder x y) 0))

(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7)))
                 integers))
```

Then we can find integers not divisible by 7 simply by accessing elements of this stream:

```
(stream-ref no-sevens 100)
117
```

Structure and Interpretation of Computer Programs λ **#7 infinite stream implementation (explicit generation)**

Following that refresher on (or intro to) **sequences** implemented as **lists** or **streams**, let's turn to the use of **infinite streams** to compute the **Fibonacci sequence**.

Structure and Interpretation of Computer Programs

In analogy with `integers`, we can define the **infinite stream** of **Fibonacci numbers**:

```
(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))

(define fibs (fibgen 0 1))
```

`fibs` is a `pair` whose `car` is `0` and whose `cdr` is a **promise** to **evaluate** `(fibgen 1 1)`.

When we evaluate this delayed `(fibgen 1 1)`, it will produce a pair whose car is 1 and whose cdr is a promise to evaluate `(fibgen 1 2)`, and so on.

Let's try **fibs** out and then define **fib**.

```
> (stream-ref fibs 5)
5

> (stream-ref fibs 50)
12586269025

> (stream-ref fibs 100)
354224848179261915075
```

```
> (define (fib n)
    (stream-ref fibs n))

> (fib 50)
12586269025

> (fib 100)
354224848179261915075
```

#7 **infinite stream** implementation (**explicit generation**)

A **stream** being a **'delayed list'**, the **Scala** equivalent is a **lazy list**.

Here is the **Scala** version of the **Scheme infinite stream implementation**.

λ

```scheme
(define fibs
  (fibgen 0 1))

(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))
```

```scala
def fibs: LazyList[BigInt] =
  fibgen(0, 1)

def fibgen(a: BigInt, b: BigInt): LazyList[BigInt] =
  a #:: fibgen(b, a + b)
```

```scheme
(define (fib n)
  (stream-ref fibs n))
```

Since calling **fibs(n)** is just as convenient as calling **fib(n)**, let's not bother defining a **Scala** version of **fib**.

```scheme
> (stream-ref fibs 5)
5

> (stream-ref fibs 50)
12586269025

> (stream-ref fibs 100)
354224848179261915075
```

```scala
> fibs(5)
val res0: BigInt = 5

> fibs(50)
val res1: BigInt = 12586269025

> fibs(100)
val res2: BigInt = 354224848179261915075
```

The **fibgen** helper function in the **infinite stream implementation** is one that we also come across in **Programming in Scala** (see next slide).

```scala
def fibs: LazyList[BigInt] =
  fibgen(0, 1)

def fibgen(a: BigInt, b: BigInt): LazyList[BigInt] =
  a #:: fibgen(b, a + b)
```

#6 **infinite stream** implementation

```scala
def fibFrom(a: Int, b: Int): LazyList[Int] =
  a #:: fibFrom(b, a + b)
```

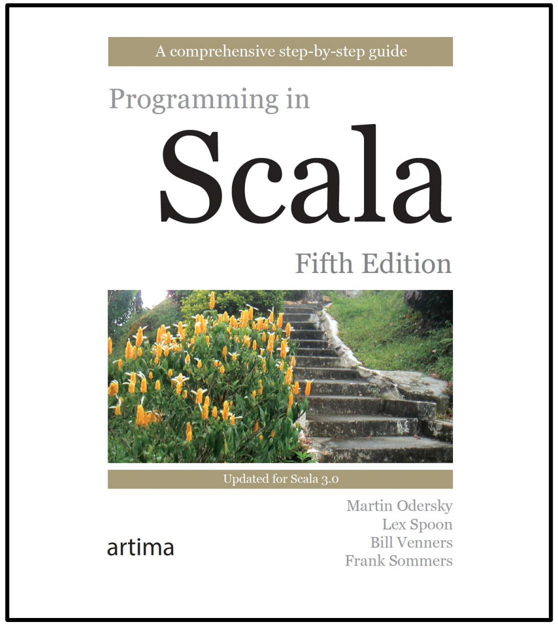
LazyList example in **Programming in Scala**

LazyLists

A lazy list is a list whose elements are computed lazily. Only those elements requested will be computed. A lazy list can, therefore, be infinitely long. Otherwise, lazy lists have the same performance characteristics as lists. Whereas lists are constructed with the :: operator, lazy lists are constructed with the similar-looking #:: . Here is a simple example of a lazy list containing the integers 1, 2, and 3:

```
scala> val str = 1 #:: 2 #:: 3 #:: LazyList.empty
val str: scala.collection.immutable.LazyList[Int] = LazyList(<not computed>)
```

The head of this lazy list is 1, and the tail of it has 2 and 3. None of the elements are printed here, though, because the list hasn't been computed yet! Lazy lists are specified to compute lazily, and the toString method of a lazy list is careful not to force any extra evaluation.



Below is a more complex example. It computes a lazy list that contains a Fibonacci sequence starting with the given two numbers. A Fibonacci sequence is one where each element is the sum of the previous two elements in the series:

```
scala> def fibFrom(a: Int, b: Int): LazyList[Int] =
         a #:: fibFrom(b, a + b)
def fibFrom: (a: Int, b: Int)LazyList[Int]
```

This function is deceptively simple. The first element of the sequence is clearly a, and the rest of the sequence is the Fibonacci sequence starting with b followed by a + b. The tricky part is computing this sequence without causing an infinite recursion. If the function used :: instead of #::, then every call to the function would result in another call, thus causing an infinite recursion. Since it uses #::, though, the right-hand side is not evaluated until it is requested.

Here are the first few elements of the Fibonacci sequence startinh witht two ones:

```
scala> val fibs = fibFrom(1, 1).take(7)
val fibs: scala.collection.immutable.LazyList[Int] = LazyList(<not computed>)
scala> fibs.toList
val res23: List[Int] = List(1, 1, 2, 3, 5, 8, 13)
```

What do **streams** look like in **Haskell**?

They are just **lists**, because **Haskell** uses an **evaluation strategy** called **lazy evaluation**, in which **expressions** are only **evaluated** as much as **required** by the **context** in which they are used.

In **Haskell** we can create an ordinary **list** that is **potentially infinite**: it is only **evaluated** as much as **required** by the **context**.

#7 infinite stream implementation (explicit generation)

Here is the Haskell equivalent of the Scala infinite stream implementation.

```scala
def fibs: LazyList[BigInt] =
  fibgen(0, 1)

def fibgen(a: BigInt, b: BigInt): LazyList[BigInt] =
  a #:: fibgen(b, a + b)
```

```haskell
fibs :: Num t => [t]
fibs = fibgen 0 1

fibgen :: Num t => t -> t -> [t]
fibgen a b = a : fibgen b (a + b)
```

```
> fibs(100)
val res0: BigInt = 354224848179261915075

> fibs.take(10).toList
val res1: List[BigInt] = List(0,1,1,2,3,5,8,13,21,34)
```

```
> fibs !! 100
354224848179261915075

> (take 10 fibs)
[0,1,1,2,3,5,8,13,21,34]
```

Othe next slide you can see a very similar Clojure implementation.

#7 infinite stream implementation (explicit generation)

Lazy Sequences

Lazy sequences are constructed using the macro lazy-seq: `(lazy-seq & body)`

A lazy-seq invokes its body only when needed, that is, when seq is called directly or indirectly. lazy-seq then caches the result for subsequent calls. You can use lazy-seq to define a lazy Fibonacci series as follows:

```
1: (defnlazy-seq-fibo
2:   ([]
3:     (concat [0 1] (lazy-seq-fibo 0N 1N)))
4:   ([a b]
5:     (let [n (+ a b)]
6:       (lazy-seq
7:         (cons n (lazy-seq-fibo b n))))))
```

On line 3, the zero-argument body returns the concatenation of the basis values [0 1] and then calls the two-argument body to calculate the rest of the values. On line 5, the two-argument body calculates the next value n in the series, and on line 7 it conses n onto the rest of the values.

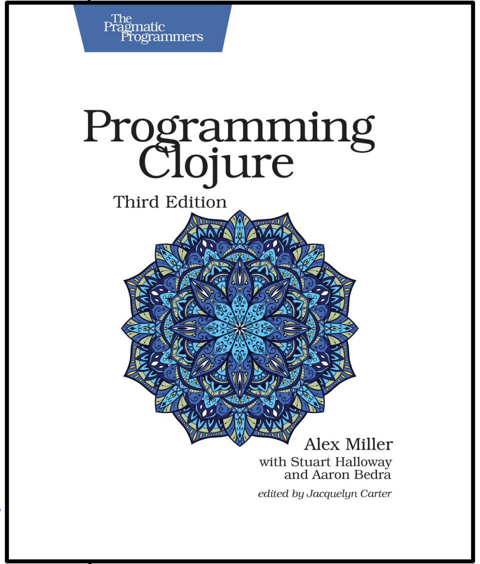
The key is line 6, which makes its body lazy. Without this, the recursive call to lazy-seq-fibo on line 7 would happen immediately, and lazy-seq-fibo would recurse until it blew the stack. This illustrates the general pattern: wrap the recursive part of a function body with lazy-seq to replace recursion with laziness.

lazy-seq-fibo works for small values:

```
(take 10 (lazy-seq-fibo))
 -> (0 1 1N 2N 3N 5N 8N 13N 21N 34N)
```

lazy-seq-fibo also works for large values. Use (rem ... 1000) to print only the last 3 digits of the one millionth Fibonacci number:

```
(rem (nth (lazy-seq-fibo) 1000000) 1000)
  -> 875N
```



Stuart Halloway
stuarthalloway

#7 **infinite stream** implementation (**explicit generation**)

Computing the millionth **Fibonacci number** using the **Scala** version runs out of **heap space**.

```
scala> def fibgen(a: BigInt, b: BigInt): LazyList[BigInt] =
     |   a #:: fibgen(b, a + b)
     |
def fibgen(a: BigInt, b: BigInt): LazyList[BigInt]

scala> def fibs: LazyList[BigInt] =
     |   fibgen(0, 1)
     |
def fibs: LazyList[BigInt]

scala> fibs(1_000_000)
java.lang.OutOfMemoryError: Java heap space
  at java.base/java.math.BigInteger.add(BigInteger.java:1475)
  at java.base/java.math.BigInteger.add(BigInteger.java:1381)
  at scala.math.BigInt.$plus(BigInt.scala:311)
  at rs$line$16$.fibgen$$anonfun$1(rs$line$16:2)
  at rs$line$16$$$Lambda/0x0000000131684800.apply(Unknown Source)
  at scala.collection.immutable.LazyList$Deferrer$.$anonfun$$hash$colon$colon$extension$2(LazyList.scala:1143)
  at scala.collection.immutable.LazyList$Deferrer$$$Lambda/0x0000000131659cd8.apply(Unknown Source)
  at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state$lzycompute(LazyList.scala:259)
  at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state(LazyList.scala:252)
  at scala.collection.immutable.LazyList.isEmpty(LazyList.scala:269)
  at scala.collection.immutable.LazyList$.$anonfun$dropImpl$1(LazyList.scala:1073)
  at scala.collection.immutable.LazyList$$$Lambda/0x0000000131659a20.apply(Unknown Source)
  at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state$lzycompute(LazyList.scala:259)
  at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state(LazyList.scala:252)
  at scala.collection.immutable.LazyList.isEmpty(LazyList.scala:269)
  at scala.collection.LinearSeqOps.apply(LinearSeq.scala:131)
  at scala.collection.LinearSeqOps.apply$(LinearSeq.scala:128)
  at scala.collection.immutable.LazyList.apply(LazyList.scala:240)
  ... 14 elided

scala>
```

#7 **infinite stream** implementation (**explicit generation**)

Computing the millionth **Fibonacci number** using the **Haskell** version does not exhaust **heap space**.

```
ghci> fibgen a b = a : fibgen b (a + b)

ghci> fibs = fibgen 0 1

ghci> (fibs !! 1000000) > 1

True

ghci>
```

#7 infinite stream implementation (explicit generation)

As a recap, here are the Scheme, Clojure, Scala, and Haskell versions compared.

```scheme
(define fibs
  (fibgen 0 1))

(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))
```

```clojure
(defn lazy-seq-fibo
  ([]
    (concat [0 1] (lazy-seq-fibo 0N 1N)))
  ([a b]
    (let [n (+ a b)]
      (lazy-seq
        (cons n (lazy-seq-fibo b n))))))
```

```scala
def fibs: LazyList[BigInt] =
  fibgen(0, 1)

def fibgen(a: BigInt, b: BigInt): LazyList[BigInt] =
  a #:: fibgen(b, a + b)
```

```haskell
fibs :: Num t => [t]
fibs = fibgen 0 1

fibgen :: Num t => t -> t -> [t]
fibgen a b = a : fibgen b (a + b)
```

In the next slide we are going to see:

1) Why I added **'explicit generation'** to the name of the implementation we have just seen
2) An even simpler implementation that defines an **infinite stream implicitly**

just seen

#7 **infinite stream** implementation (**explicit generation**)

#8 **infinite stream** implementation (**implicit definition**)

coming up next

Defining streams implicitly

The integers and fibs streams above were defined by specifying "generating" procedures that explicitly compute the stream elements one by one. An alternative way to specify streams is to take advantage of delayed evaluation to define streams implicitly.

For example, the following expression defines the stream ones to be an infinite stream of ones:

```
(define ones (cons-stream 1 ones))
```

This works much like the definition of a recursive procedure: ones is a pair whose car is 1 and whose cdr is a promise to evaluate ones. Evaluating the cdr gives us again a 1 and a promise to evaluate ones, and so on.

We can do more interesting things by manipulating streams with operations such as add-streams, which produces the elementwise sum of two given streams

```
(define (add-streams s1 s2)
  (stream-map + s1 s2))
```

The stream-map function used here is a generalisation of the one seen in slide 12, in that it allows the mapping of procedures that take multiple arguments.

Now we can define the integers as follows:

```
(define integers (cons-stream 1 (add-streams ones integers)))
```

This defines integers to be a stream whose first element is 1 and the rest of which is the sum of ones and integers.

Thus, the second element of integers is 1 plus the first element of integers, or 2; the third element of integers is 1 plus the second element of integers, or 3; and so on.

This definition works because, at any point, enough of the integers stream has been generated so that we can feed it back into the definition to produce the next integer.

Structure and Interpretation of Computer Programs

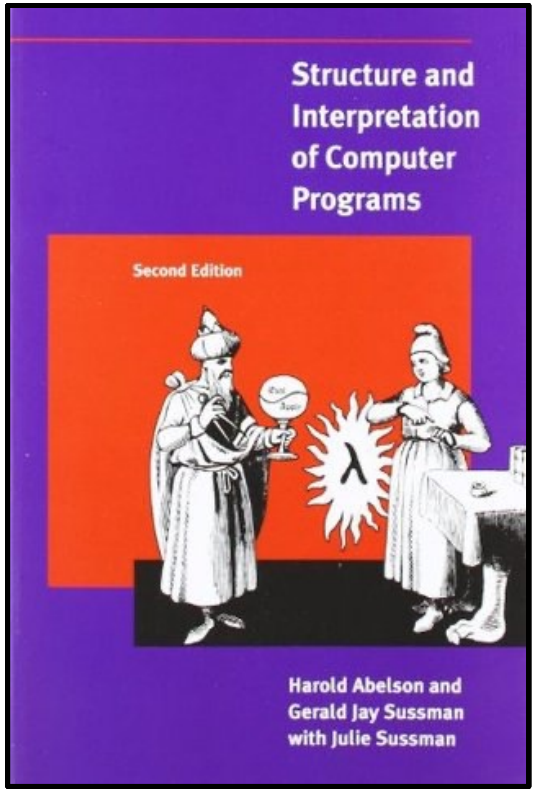
λ #8 **infinite stream** implementation (**implicit definition**)

We can define the Fibonacci numbers in the same style:

```
(define fibs
  (cons-stream 0
               (cons-stream 1
                            (add-streams (stream-cdr fibs)
                                         fibs))))
```

This definition says that fibs is a stream beginning with 0 and 1, such that the rest of the stream can be generated by adding fibs to itself shifted by one place:

```
  1 1 2 3 5 8  13 21 … = (stream-cdr fibs)
  0 1 1 2 3 5  8  13 … = fibs
0 1 1 2 3 5 8 13 21 34 … = fibs
```



SICP

Here is the Scala version of the Scheme infinite stream implementation with implicit definition.

#8 infinite stream implementation (implicit definition)

λ

```
(define fibs
  (cons-stream 0
               (cons-stream 1
                            (add-streams (stream-cdr fibs)
                                         fibs))))
```

```
    1 1 2 3 5 8  13 21 … = (stream-cdr fibs)
    0 1 1 2 3 5  8  13 … = fibs
  0 1 1 2 3 5 8 13 21 34 … = fibs
```

```
val fibs: LazyList[BigInt] =
  BigInt(0) #:: BigInt(1) #:: (fibs zip fibs.tail).map(_+_)
```

```
scala> fibs(10)
val res0: BigInt = 55

scala> fibs(100)
val res1: BigInt = 354224848179261915075

scala> fibs(1000)
val res2: BigInt = 43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079
255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

```
scala> fibs(100_000)

val res0: BigInt =
2597406934722172416615503402127591541488048538651769658472477070395253454351127368626555677283671674475463758722307443211163839947387509103096569738218830449305228763853133492135302679278956701051276578271635608073050532200243233114383986516137827238124777453778337299916214634050054669860390862750996639366409211890125271960172105060300350586894028558103675117658251368377438684936413457338834365158775425371912410500332195991330062204363035213756525421823998690848556374080179251761629391754963458558616300762819916081109836526352995440694284206571046044903805647136346033000520852277707554467947237090309790190148604328468198579610159510018506082649192345873133991501339199323631023018641725364771362664750801339824312317034314529641817900511879573167668349799016820118499077566864568450662873924856039140476051995500662888263458771894106803700918793650017330117100283104739474562560914449328213748555738640805798130282666402703542944121049199958031318768058991865134251759599115205631553377039969410355182752749199598022575079020377981030899229849963044962558140455170002502997643221934621653662108418767454282982613982344783665815880408190033073829395000821320093747154851310272208173054322648669496309879147143629255542526240439996153269798768075106468190687921182991679644091782718685617029181022126792674013626504997849688436809752547001310045741864064482994858725517447466956518791269169932445648176733222571493149677633458466238303338202397024368594782876418757885729107101337003000942293335972927791914092128049015459762627910570555248158884051779418192905216769576608748815567860128818354354292307397810154785701328438612728620176653953444993001980062953893698550072328665131718111358866135374726845854325489811371766051946169379168844253425947812631038895204795659438071530191125396484711263890071336285691015514534233294412843572209962867461194209516610023097407099655319005081586699114454426478828726428450172533204864831945789203998489382363674561822037509734856684743388724904933703163382657176072977889179891366732519062324711803728017392157239082276922807729245666275053833750069260772105936194212689203025674435653780083183063759333450235025697290651528532719436775601566603991640488256396769307929050295148869341379912517485666707471751493897903865333813953468483780861267375543838211084489765383684831825883633991731045585090566384620250146313118310874290772926221594302042915947403061018398168550669502619737615085717611994758757221298720531206079186498036159609233959410411863516885488391191851790615115627529361584900087215019222651178531508925102752804515123860379218469212153382928713692432152733271415747882959026015719548531644479454675028584023600023834479052034510803328201380388070898073483262012279526336067736698757833262548594490602191736886778624112056210983698501972901771578011204045864915393511578349954610063663574544850824188827906753135995051920622297601537652979730858816487311730823705982848940448740393205359293597645416556079547247786202996923295613897198946794221872736051233655952113310877875822887959758032045960847902450638519417431261637751045992110248687949634170686209290889306852523480569259983337751039010131661781230511457193270662916712544651215174680254819035835168897170757067786561880082203468363210181302623299602759940357999777404624495211453158837035790448329315000724617341735580556783215345434117002025856080916629419863740151456957227283692196322951118776253075340259478144820465746028848550006280693481139827601685558407954216205754355729151064153759293902288435612079264370556006236798654438246437394697247194599655579550583803482559783968277608473153025178895171863072276110363050936007426226171736305861329154402469543290461625869177463057850767493748799232918175016348406881346553437099758935360740517290941269765759329515681862474712763646883655175701835341727466260730651045119576286634992284867878059108511898565355543495876166401644758802863362970404628909706773625658430023531474946123391206863214663708784469921042754156941091224656857120471724113337848981676409692498163342117685715031167104006817530319211541561195804257065869312727621371069747222602965552461105371555453249975084327520019921430191050536299600704296329780510306665063878626815765877268374512897685079636637105938091122542883583919412115477375998130192165095214013330607098731373292651816922684506344395405672981203154639232498179378046910379342216949522910079302994923750729932506305094281390279308413447306141164335561476409310442591848136393054236937897652052645634764831827263337151211203062923388928648794920973784786188486826080464731953920084039830800880386904955741975621929392211082576639768136104449002472094834032679676883762139674407571388729286307982184931434387977808873795889684094614341592713175783651145782893558185990292353438888884658745213083813777944363611976283903689459576012031650227985790154534474735270697285145459986142290273729113146378204551622544753535677362279364854503571020864454120898423503890877022303984938021473480968743333622544915011741175157070456105089527400020638049796796040261781866448124854726963082347337724554339051984130876978127656591676422902294818176307571025579336500815228638363449131380899717850870708636322058690189383777660630060667577324272729292474212952650007066467227300099561241914091389846752249557907293984956087504566942177715511073466304566039441362358884436762152739285970722879373559667239246138274687032178584599482575147454064364609970593161205968415604732343966524572316503177928338605903883604176914287327357039868033426046700717173635730911229813069032861371225979370966057751729645282637574340757922821807443529086696068540217185978911663338638585897362091142484321786450394791954242081916260885710691104339948014730131008698488866430721216762473119618190737820766582968280796079482259549036328266578006994856825300536436674822534603705134503603152154296943991866236857638062351209884448741138600171117364763212602996140856192559970756682786677873237741944446227539990929104469771647615111867232723867920813336730618194484939660712334527185652025364362196419878275297881306008031314181706931446822118927578497828109436775154071010635055379800384221904550848223938699329692665922111274269813306230000734656284980936366930494468016285537126334126203784919194986000972008367278766507868863069334189952257683143908324848863403189401941610369798438333466086767094316436535384309121578155435128520777208580989020995864496024794919706872307656871092343807195098248144731578137800806393584187566550985013218828528401849814076907385073695353777118803885289353476009303385986916082893354211477229365619072762646037260272393209911878204070674122722581207667290400719242379303309721323641840939561029959712917998282900095391473824378027790511120309545825328887211461701334403859396540478061993332245473178034073409025121302172795957538631581488103929524754109438805550983826276331276067181261710220113561818007754002275167341441692164249731756213631285882819780057888324545345815224349372681334339977105125320814783450671398350383329013139459864818202723220433419309290119078328965692228783374973543015617228291156273294688148532819221007523736268276431526857354932230280181014496490090155292486383338885664893002250974343601200814365153625369199446709711126951966725780061891215440222487564601554632812091945824653557432047644212650790655208208337976071465127508320487165271577472325887275761128357592132553934446289433258105028633583669291828566894736223508250294964065798630809614341696830467595174355313224362664207197608459024263017473392225291248366316428006552870975051997504913009859468071013602336440164400179188610853230764991714372054467823597211760465153200163085336319351589645890681722372812310320271897917951272799656053694032111242846590994556380215461316106267521633805
```

56643943188812681994940055370686976218552318589211009634410129335357339184596681975398342846968228894600763520316889220020219313183697575569620611157743058263055358620156378
91246031220672933992617378379625150999935403648731423208873977968908908369996292995391977217796533421249291978383751460062054967341662833487341011097770535898066498136011395
57158432830871394058253527405608101150390794168807919721293314830307263867863141103844312821599493682434299818871976863760449634259752425688618868978980888315865076262604 85
64650043228968561492550639688114044004295038942458723822335431010786915173283336047792627277656860761777056168740502577437499837758301438561354272738385897741335269491654839
29721519554793578923866762502745370104660909382449626626935321303744538892479216161188889702077910448563199514826630802879549546453583866307344423753319712279158861707289652
09014984830543598320077132665340729066201677570640969018377120130682324533347796666052532549087360196148037824156607127165038358225728921570820936951099589013285949072430618
33257552012080900071750220229497428018234454137119162984499147222541965946822214682606449618392542496709031040075814888579716722463228870164384039084638567311643081695373267
90303114583680575021119639905615169154708510459700542098571797318015564741406172334145847111268547929892443001391468289103679179216978616582489007322033591376706527676521307
14398530276098847805621699465965546137917498565973922737941672649537780199209835542786617912312669937473077773056932443016683933301155451554265686493749212868704912175424596
78311329692484924667442619990339728256748734602011504422287804661243201830161082321839086547710423982285313165596856880052265714744288233175394565438819286244326625033453881
99590085105211383124491861802624432195540433985722841341254409411771722156867086291742124053110620522842986199273629406208834754853645128123279609097213953775360023076765694
20821994303464878334854449271353945022459133437466493770165560576338469706291872574542650587941463017663976045747431108155674709165270874812526715991379324052730461369396116
98925898083119063225107779285620719994594877006118010022961323045882945584409524966111583428049086438608807964405577636918577437540258968559272525145634043852178258905995539
54627451385454452916761042969267970893580056234501918571489030418495767400819359973218711957496357095967825171096264752068890806407651445893132870767454169607107931692704285
16809341331104635350624220981036321677191042078616218421376393819462569728678141363638962012397691046541895680619732314841422455007161721585132130203068417608721589270209887
91089380810459033972765473264169168454456276007595613671035845756490944306924525320850030910687831575615198475675691912847846546925586651115579134612724253360836351313421839
05177154511228464455136016013513228948543271504760839307556100908786096663870612278690274831819331606701484957163004705262222823840626681844878837454813199438038761383012885 9
88526420199228618820849958864088852135250145761539648264745102590253074317295689963649961570755185583716593536712544851508936290456773663003556245737477910098799249914696722
40414816012895309440154889426137831400878043114317418580718261851490511387448313584390672289494082582860216502889272283874264327861686903819605301558944594518087351972460082
21529343980828254126128257157209350985382800738560472910941184006084485235377833503306861977724501886364070344973366473100602018128792886991861824418453968994777259482169137
13364747045317297980924584436112961899759569624097184556402051143258959184472492094293030165148871307980210237906553652515478029805940752944051314580755153779486163587990115
81920198088796949671874482241568364635343261602426329347616344581638901638051238941845239734218414968892623984896486420934098166814947711551770095626690298501015135375998012
72501241971119871526593747484778935488777815192931171431167444773882941064615028751327709474504763922874890662989841540259350834035142035136168819248238998027706666916342133
42431205450735938866168769118818577611813577133248396520988208598239129860638682280475436240895652292141085985203733054462595326134023486468927506052689375514840329854208699
12210525970056285767077025676953009789700464089200098521069802954196998021380532957981594782899344432454915653278452238405512404452082264354206563133107029407223715527705042
63482073984454889589248861397657079145414427653584572951329719091947694411910966797474262675590953832039169673494261360032263077428684105040061351052194413778158095005714526
84600981035210924904002795805073643696102124113773971716486952549311480504012656835126882959841398322267637780450062650724173175739521979689075482519932925964980162706866565
80301788774056151671597319273204793762473755058550528396602945669925221736008740812120142090710419375985717214313380174251415824918247109050847159772494170493202541652393232
33258851588893337097136310892571531417761978326033750109026284066415801371359356529278088456305951770081443994114674291850360748852366654744869928083230516815711602911836374
14795849210086052898146954775081233889694315286102120273674704990393041703517134212692348670056662750622905863691188222890317051030540688209697087554532936943406398129769647
80318254516421783473477164710584232385945801830527562139101869976043058440686657123468696794560441557421000391797583489799358827518815246759308789281592434921975453876683056
84668420775409821781247053354523194797398953320175988640281058825557698004397120538312459428957377696001857497335249965013509368925958021863811725906506436882127156815751021
71290076599275037022828396396291597325117341858672102349731776596945428362551937155600914368032931196284254662840314244437064843239037490641081130079284895576724348120009030
98884572709077508736388732996425550504738125289759629348228789176199207251383093882882925104168376227582040819189336036538752841167857037209897188329869219278166296758445801
74911809119663048187434155067790863948831489241504300476704527971283482211522202837062857314244107823792513645086677566622804977211397140621664116324756784216612961477109018
82609467737768640617672148429389497667138012278894130902655351109611834701256519754080709538406091686393690667378662720942943426426040290215831734500372746258899262204987712
11784055633484924903260035085690993823927772974984135656148307882623633223683807098223460122742413790364734517359252157547571609342709351929017239549214264906911152715233381
09124042812102893738488167358953934508930697715522989199698903885883275409044300321986834003470271220020159699371690650330547577095398748580670024491045504890061727189168031
39452803616563394157133463722255047754746075605502410876438212168884891694037125890194849068537972224456200948381949153272450227621858916950740579498375982100660448199651936
01102615769471762025717020486849146168940684041408335875621183192108380056321445620189415059457800253187474719116048406779977654148306221790693308538751292989830095802775541
45435058768984944179136535891620098725222049055183554603706533183176716110738009786625247488691476077664470147193074476302411660335671765564874440577990531996271632972009109
44924921645603061882777294775076477744645258632891915910744452320082918209518021083700353881330983215894608680127954224752071924134648334963915094813097541433244209299930755
14810779190023461281223301617994299306188005334145506339321393396468616164169552202164479954172431711657444713641977332048993650747678441499295480730258564429423817876415064
92878361767978677158510784235702640213388018875601989234056868423215585628508645525258377010620532224244987990625263484010774322488172558602233302076399933854152015343847725
44291789513063705032044491779775237087195827797679968611362653229111862963116468515993466069346055754595606315583003369763400027668515129384363888609082837614115773200352756
51587459065670254394379311048385713132944906049265823631089495350900826731544972263966480886180415739778884728921746189741897217007700098624496537590127270152276345108749069
48012210684952063002519011655963580552429180205586904259685261047412834518466736938580027700252965356366721619883672428226933950325930390994583168665542234654857020875504617
52052185372156728267990341813552060299989536647010655790053212954133692447249221243632452304289518846177912233806967423398069488727058750338922839509513520912310925815900696

```
03951563677360671090505662996035718764232479207528361608055976977787564767672105212223271848214844466312614875842260926088757643317310232637688648225946912110323677375581
22133470556805958008310127481673962019583598023967414489867276845869819376783757167936723213081586191045995058970991064686919463448038574143829629547131372173669836184558
14450574867612432245151994336218291619146802609112179300186478805006135160314435007618921344160248809174105123229035717920549792797092450247994084269615881844261616378004
47594782122408732041244211691998055726491182436619218357147628914258057718717436880003241130087048193739622950171430900984769272374988759386399425305953316078916188108635
05982444578942799346514915952884869757488025823353571677864826828051140885429732788197765736966005727700162592404301688659946862983717270595809808730901820120931003430058
79655269478804980920548430546761103465474806729067439976361259243463771999584386281239198547020241488007688081884808789239159136946329311327684932977720164664172758725912
23547844808134333280500877588552646861195769621722393086937957571658218524162043419723839899327348034292623407223381551022091012629492497424232716988420232973032601617905
75673111235465890298298313111512360760677396899815381228699964201460985257979369124601634608876232128620563421590147918863219465963748348256429161627853294823931 3229440231
04327728876813955021334826638868745325928158785450389099156194963247885503509028939097371898800399902613201587267863787309567810962531100805448941885798356590206368069964
31650339120299443277267708693052407184165920700961392864019667257500870122181497331336958096003697517649513500402859262492033981110149532275336218445007443315624345324842
17986108346261345897591234839970751854223281677187215956827243245910829019886390369784542622566912542747056097567984857136623679023878478161201477982939080513150258174523
77352951016529693456278612224115078358775537334837276443983808200066721474003446632277691893696761287898348894209468810230842703645285450496675969731883604449670285319063
73969163579809288657199353977234954867871804164014152814894437850362910715178052858575839877111454742401564164771941163913549354667555935926088492005463846854030280809364
17250583653368093407225310820844723570226809826951426162451204040711501448747856199922814664565893938488028643822313849852328452360667045805113679663751039248163336173274
54727577563681097734453927582756059742516070546868965779453052160231593986578097480151541498709777807870535705800847237689242218975031275852714017311762127989874495840619
98439133656802977212087519349885044997139142851580323248230213406303125860726245416377652345055220510863182853596585207081733927095664450114040551065790550374177803933516
58360904543047721422281816832539613634982525215232257690920254216409657452618066051777901592902884240599998882753691957540116954696152270401280857579766154722192925655963
99182094889464265751228876633030213374636744921744935163710472573298083281272646818775935658421838359470279201366390768974173896225257578266399080979264701140758036785059
93818871845600946958332707751261812820153910417739509182441375619999378192403624695582359241714787027794484431087519018074141102903707060520851629757983617542510416422448
67577350756338018895379263183389855955956527857227926155524494739363665533904528656215464288343162282921123290451842212532888101415884061619939195042230059898349966569463
58018681671707481882321584864773438678091156466075517538555222442852404946803369229998930078390002069012151774069642857393019691050098827852305379763794025796895329511243
61667789105855572133817890899454539479159273749586002682378444868720372434888346168562900978505324970369333619424398028823643235538082080038757417109692897254998785662530
48867033095150518452126944989251596392079421452606508516052325614861938282489838000815085351564642761700832096483117944401971780149213345335903336672376719229722069970766
05548245224741692777463752213520171623172213763244569915402239549415822741893058991174693177377651873585003231801443288391637424379585469569122177409894861151556404660956
50945381155209218637115186845625432750478705300069984231401801694211091059254935961167194576309623288312712683285017603217716804002496576741869271132155732700499357099423
24416387089242427584407651215572676037924765341808984312676941110313165951429479377670698881249643421933287404390485538222160837088907598277390184204138197811025854537088
58670145062357851396010998747605253545010043935306207243970997644514679099338144899464460978095773195360493873495002686056455569322422969181563029392248760647087343116638
42054424896287602136502469918930401125131038350856219080602708666048735858490017042009239297891939381251167984217881152092591304355723216356608956035143838839390189531662
74355609970015699780289236362349895374653428746875

scala>
```

Earlier I said that in **Haskell**, **streams** are just ordinary **lists**, because we can create an ordinary **list** that is **potentially infinite**: it is only **evaluated** as much as **required** by the **context**.

Remember the first **Scheme** example of an **implicitly defined infinite stream**?

```
(define ones (cons-stream 1 ones))
```

λ

Here is how it looks in **Haskell**.

```
> ones = 1 : ones

> head ones
1

>  tail (head ones)
1

> ones !! 100
1

> take 10 ones
[1,1,1,1,1,1,1,1,1,1]
```

Remember the passage (in slide 3 of part 1) in which **Paul Hudak** asked what should be done to address the fact that the **time complexity** of the **naïve** implementation is **exponential**?

His answer, in the next four slides, is the **Haskell** version of the **infinite stream** implementation with **implicit definition**.

To understand **the cause of this inefficiency**, let's begin the calculation of, say, $fib\ 8$:

#1 naïve implementation

$$fib :: Integer \to Integer$$
$$fib\ 0 = 1$$
$$fib\ 1 = 1$$
$$fib\ n = fib\ (n-1) + fib\ (n-2)$$

$$fib\ 8$$

$$\Rightarrow fib\ 7 + fib\ 6$$

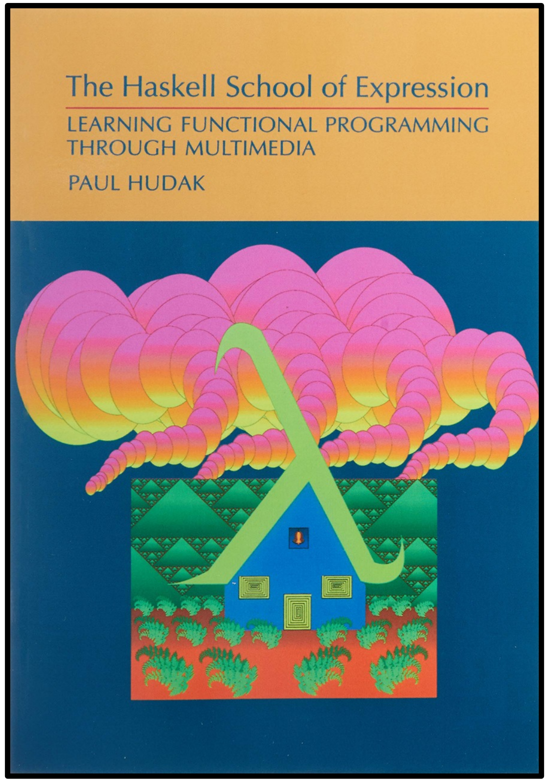
$$\Rightarrow (fib\ 6 + fib\ 5) + (fib\ 5 + fib\ 4)$$

$$\Rightarrow ((fib\ 5 + fib\ 4) + (fib\ 4 + fib\ 3)) + ((fib\ 4 + fib\ 3) + (fib\ 3 + fib\ 2))$$

$$\Rightarrow \begin{pmatrix} ((fib\ 4 + fib\ 3) + (fib\ 3 + fib\ 2)) \\ + \\ ((fib\ 3 + fib\ 2) + (fib\ 2 + fib\ 1)) \end{pmatrix} + \begin{pmatrix} ((fib\ 3 + fib\ 2) + (fib\ 2 + fib\ 1)) \\ + \\ ((fib\ 2 + fib\ 1) + (fib\ 1 + fib\ 0)) \end{pmatrix}$$

…

It is easy to see that **this calculation is blowing up exponentially**. That is, **to compute the nth Fibonacci number will require a number of steps proportional to 2^n**. **Sadly, many of the computations are being repeated**, but in general we cannot expect a **Haskell** implementation to realise this and take advantage of it. **So what do we do?**



Paul E. Hudak

#8 infinite stream implementation (implicit definition)

One solution is to construct the Fibonacci sequence directly as an infinite stream. The key to this construction is noticing that if we add pointwise the Fibonacci sequence to the tail of the Fibonacci sequence, we get the tail of the tail of the Fibonacci sequence.

```
1 1 2 3 5  8  13 21 … = Fibonacci sequence
1 2 3 5 8  13 21 34 … = tail of Fibonacci sequence
_______________________________________________
2 3 5 8 13 21 34 55 … = tail of tail of Fibonacci sequence
```

This leads naturally to the following definition of the Fibonacci sequence:

$$fibs :: [Integer]$$
$$fibs = 1 : 1 : zipWith\ (+)\ fibs\ (tail\ fibs)$$

Note the concise and naturally recursive nature of this definition. Evaluating $take\ 10\ fibs$ yields:

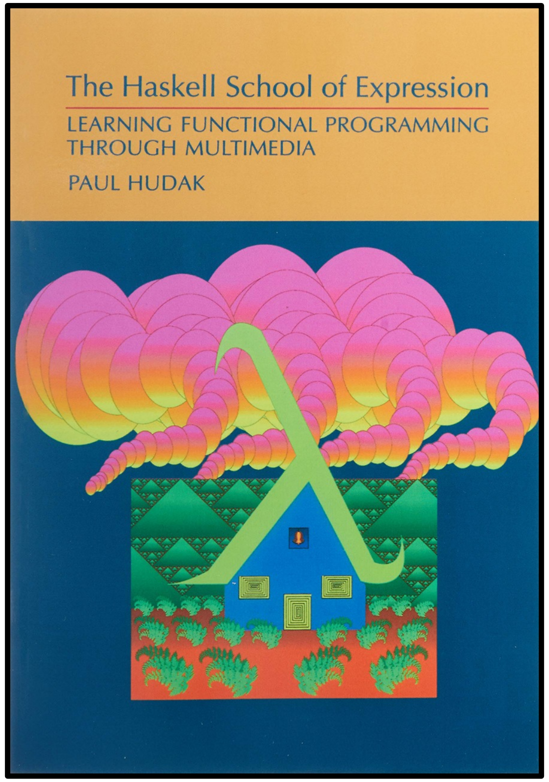
```
[1,1,2,3,5,8,13,21,34,55]
```

as expected.

DETAILS

Try running **take n fibs** with successively larger values of **n**; you will find that it runs very fast, even though the **Fibonacci numbers** start getting quite large. In fact, in Hugs the time is dominated by how long it takes to print the numbers; try running **fibs !! 1000** to see how quickly a single value can be computed and printed.

This program is very efficient. To see why, we can proceed by calculation.



Paul E. Hudak

The first step is easy:

$$fibs$$
$$\Rightarrow 1 : 1 : add\ fibs\ (tail\ fibs)$$

where, for succinctness, I have written $\boldsymbol{add}$ for $\boldsymbol{zipWith}\ (+)$.

However, if we now simply substitute the definition of $fibs$ for both of its occurrences, we find ourselves heading toward the same **exponential blow-up** that we saw earlier:

$$\Rightarrow 1 : 1 : add\ \big(1 : 1 : add\ fibs\ (tail\ fibs)\big)$$
$$(tail\ (1 : 1 : add\ fibs\ (tail\ fibs)))$$

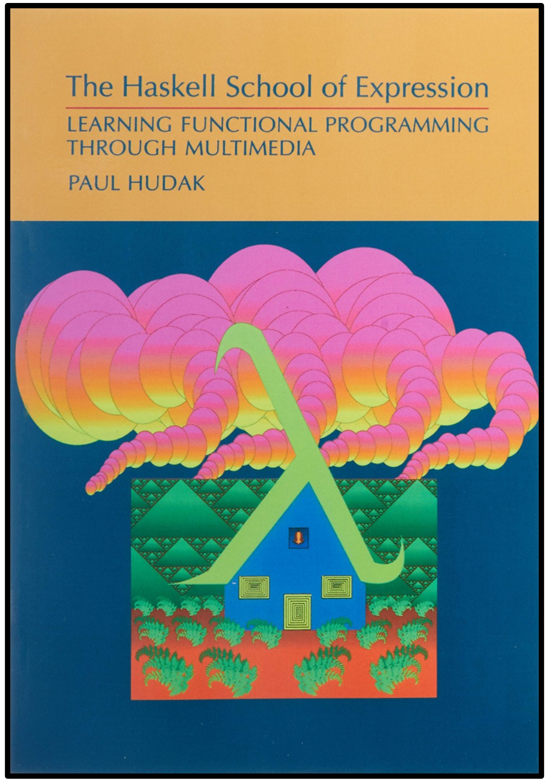
Fortunately, a Haskell implementation will be cleverer than this by sharing $fibs$ **as well as its tail. Starting from the beginning again, this sharing is noticeable immediately after the first step:**

$$fibs$$
$$\Rightarrow 1 : 1 : add\ fibs\ (tail\ fibs)$$

At which point, **we can express the sharing of the tail using a** ***where*** **clause:**

$$\Rightarrow 1 : tf$$
$$\textbf{where}\ tf = 1 : add\ fibs\ (tail\ fibs)$$

$$\Rightarrow 1 : tf$$
$$\textbf{where}\ tf = 1 : add\ fibs\ tf$$



Paul E. Hudak

We can also express the sharing of the tail of the tail in preparation for unfolding *add* (I will use $tf2$, $tf3$, and so on, for the names of the successive tails):

$$\begin{aligned}&\Rightarrow 1 : tf\\ &\quad \textbf{where } tf = 1 : tf2\\ &\quad\quad\quad \textbf{where } tf2 = add\ fibs\ tf\end{aligned}$$

Finally, we can unfold *add* to yield:

$$\begin{aligned}&\Rightarrow 1 : tf\\ &\quad \textbf{where } tf = 1 : tf2\\ &\quad\quad\quad \textbf{where } tf2 = 2 : add\ tf\ tf2\end{aligned}$$

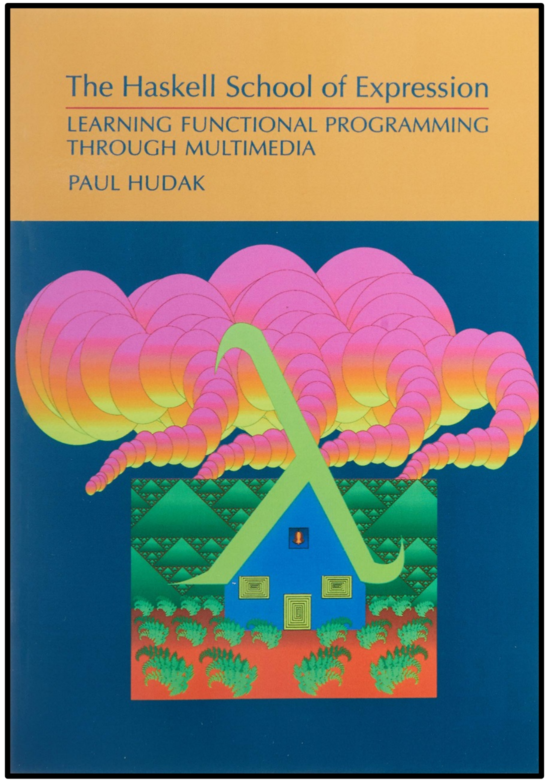
Repeating this process, we introduce even more sharing, and unfold *add* again:

$$\begin{aligned}&\Rightarrow 1 : tf\\ &\quad \textbf{where } tf = 1 : tf2\\ &\quad\quad\quad \textbf{where } tf2 = 2 : tf3\\ &\quad\quad\quad\quad\quad \textbf{where } tf3 = add\ tf\ tf2\end{aligned}$$

$$\begin{aligned}&\Rightarrow 1 : tf\\ &\quad \textbf{where } tf = 1 : tf2\\ &\quad\quad\quad \textbf{where } tf2 = 2 : tf3\\ &\quad\quad\quad\quad\quad \textbf{where } tf3 = 3 : add\ tf2\ tf3\end{aligned}$$

But now note that tf is only used in one place, and thus might as well be eliminated, yielding:

$$\begin{aligned}&\Rightarrow 1 : 1 : tf2\\ &\quad \textbf{where } tf2 = 2 : tf3\\ &\quad\quad\quad \textbf{where } tf3 = 3 : add\ tf2\ tf3\end{aligned}$$



Paul E. Hudak

At this point, we can begin to repeat the sequence of introducing a new *where* clause to capture sharing, unfolding *add*, and then eliminating the outermost *where* binding. This will yield successively longer versions of the result. Here is one more application of that sequence:

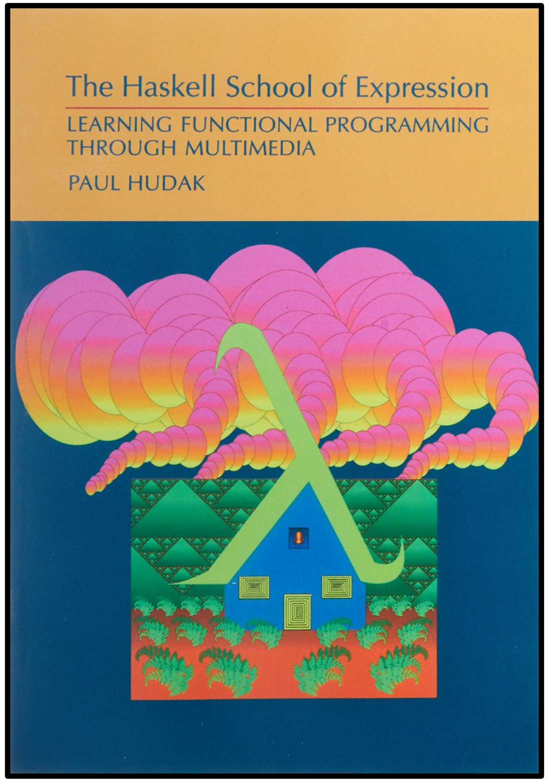
$$\Rightarrow 1 : 1 : tf2$$
$$\quad \textbf{where}\ tf2 = 2 :\ tf3$$
$$\qquad \textbf{where}\ tf3 = 3 : tf4$$
$$\qquad\quad \textbf{where}\ tf4 = add\ tf2\ tf3$$

$$\Rightarrow 1 : 1 : tf2$$
$$\quad \textbf{where}\ tf2 = 2 :\ tf3$$
$$\qquad \textbf{where}\ tf3 = 3 : tf4$$
$$\qquad\quad \textbf{where}\ tf4 = 5 : add\ tf3\ tf4$$

$$\Rightarrow 1 : 1 : 2 :\ tf3$$
$$\qquad \textbf{where}\ tf3 = 3 : tf4$$
$$\qquad\quad \textbf{where}\ tf4 = 5 : add\ tf3\ tf4$$

The reason why there are always at least two *where* clauses is that *fibs* recurses on itself as well as its tail. The elimination of the *where* clause corresponds to the garbage collection of unused memory by an implementation.

Although this process may seem a bit tedious, it is important only when wanting to reason about the efficiency (in time and space) of the calculation. If you are just interested in the resulting values, you can, of course, use the exponentially expanding calculation, with no fear that you might get a different answer.



Paul E. Hudak

```
ghci> fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

ghci> fibs !! 100000

2597406934722172416615503402127591541488048538651769658472477070395253454351127368626555677283671674475463758722307443211163839947387509103096569738218830449305228763853133
4921353026792789567010512765782716356080730505322002432331143839865161378272381247774537783372999162146340500546698603908627509966393664092118901252719601721050603003505868
9402855810367511765825136837743868493641345733883436515877542537191241050033219599133006220436303521375652542182399869084855637408017925176162939175496345855861630076281991
60811098365263529954406942842065710460449038056471363460330005208522777075554446794723709030979019014860432846819857961015951001850608264919234587313399150133919932363102301
8641725364771362664750801339824312317034314529641817900511879573167668349799016820118499077566864568450662873924856039140476051995500662888263458771894106803700918793650017
3301171002831047394745625609144493282137485557386408057981302826664027035429441210491999580313187680589918651342517595991152056315533770399694103551827527491995980225750790
2037798103089922984996304496255814045517000250299764322193462165366210841876745428298261398234478366581588040819003307382939500082132009374715485131027220817305432264866949
6309879147143629255542526240439996153269798768075106468190687921182991679644091782718685617029181022126792674013626504997849688436809752547001310045741864064482994858725517
4474669565187912691699324456481767332225714931496776334584662383033382023970243685947828764187578857291071013370030009422933359729277919140921280490154597626279105705524815
8884051779418192905216769576608748815567860128818354354292307397810154785701328438612728620176653953444993001980062953893698550072328665131718113588661353747268458543254898
1137176605194616937916884425342594781263103889520479565943807153019112539648471126389007133628569101551453423329441284357220996286746119420951661002309740709965531900508158
6699114454426478828726428450172533204864831945789203998489382363674561822037509734856684743388724904933703163382657176072977889179891366732519062324711803728017392157239082
2769228077292456662750538337500692607721059361942126892030256744356537800831830637593334502350256972906515285327194367756015666039916404882563967693079290502951488693413799
1251748566670747175149389790386533381395346848378086126737554383821108448976538368483182588363399173104558509056638462025014631311831087429077292622159430204291594740306101
8398168550669502619737615085717611994758757221298720531206079186498036159609233959410411863516885488391191851790615115627529361584900087215019222651178531508925102752804515
1238603792184692121533829287136924321527332714157478829590260157195485316444794546750285840236000238344790520345108033282013803880708980734832620122795263360677366987578332
6254859449060219173688677862411205621098369850197290177157801120404586491539351157834995461006366357454485082418882790675313599505192062229760153765297973085881648731173082
3705982848940448740393205359293597645416556079547247786202996923295613897198946794221872736051233655952113310877875822887959758032045960847902450638519417431261637751045992
1102486879496341706862092908893068525234805692599833377510390101316617812305114571932706629167125446512151746802548190358351688971707570677865618800822034683632101813026232
9960275994035799977740462449521145315883703579044832931500072461734173558055678321534543411700202585608091662941986374015145695722728369219632295111877625307534025947814482
0465746028848550006280693481139827601685558407954216205754355729151064153759293902288435612079264370556006236798654438246437394697247194599655579550583803482559783968277608
4731530251788951718630722761103630509360074262261717363058613291544024695432904616258691774630578507674937487992329181750163484068813465534370997589353607405172909412697657
5932951568186247471276364688365517570183534172746626073065104511957628663499228486787805910851189856535554349587616640164475880286336297040462890970677362565843002353147494
6123391206863214663708784469921042754156941091224656857120471724113337848981676409692498163342117685715031167104006817530319211541561195804257065869312727621371069747222602
9655524611053715554532499750843275200199214301910505362996007042963297805103066650638786268157658772683745128976850796366371059380911225428835839194121154773759981301921650
9521401333060709873137329265181692268450634439540567298120315463923249817937804691037934221694952291007930299492375072993250630509428139027930841344730614116433556147640931
0442591848136393054236937897652052645634764831827263337151211203062923388928648794920973784786188486826080464731953920084039830800880386904955741975621929392211082576639768
1361044490024720948340326796768837621396744075713887292863079821849314343879778088737958896840946143415927131757836511457828935581859902923534388888846587452130838137779443
6361197628390368945957601203165022798579015453447473527069728514545998614229027372911314637820455162254475353567736227936485450357102086445412089842350389087702230398493802
1473480968743333622544915011741175157070456105089527400020638049796796040261781866448124854726963082347337724554339051984130876978127656591676422902294818176307571025579336
5008152286383634493138089971785087070863632205869018938377766063006066757732427272929247421295265000706646722730009956124191409138984675224955790729398495608750456694217771
5511073466304566039441362358884436762152739285970722879373559667239246138274687032178584599482575147454064364609970593161205968415604732343966524572316503177928338605903883
6041769142873273570398680334260467007171736357309112298130690328613712259793709660577517296452826375743407579228218074435290866960685402171859789116633386385858973620911424
8432178645039479195424208191626088571069110433994801473013100869848866430721216762473119618190737820766582968280796079482259549036328266578006994856825300536436674822534603
7051345036031521542969439918662368576380623512098844487411386001711736476321260299614085619255997075668278667787323774194444622753999092910446977164761511186723272386792081
3336730618194484939660712334527185652025364362196419878275297881306008031314181706931446822118927578497828109436775154071010635055379800384221904550848223938699329692665922
1112742698133062300073465628498093636693049446801628553712633412620378491919498600097200836727876650786886306933418995225768314390832484886340318940194161036979843833346608
6767094316436535384309121578155435128520777208580989020995864496024794919706872307656871092343807195098248144731578137800806393584187566550985013218828528401849814076907385
0736953537771188038852893534760093033859869160828933542114772293656190727626460372602723932099118782040706741227225812076672904007192423793033097213236418409395610299597129
1799828290009539147382437802779051112030954582532888721146170133440385939654047806199333224547317803407340902512130217279595753863158148810392952475410943880555098382627633
1276067181261710220113561818007754002275167341441692164249731756213631285882819780057888324545345815224349372681334339977105125320814783450671398350383329013139459864818202
7232204334193092901190783289656922287833749735430156172282911562732946881485328192210075237362682764315268573549322302801810144964900901552924863833888566489300225097434360
1200814365153625369199446709711126951966725780061891215440222487564601554632812091945824653557432047644212650790655208208337976071465127508320487165271577472325887275761128
3575921325539344462894332581050286335836692918285668947362235082502949640657986308096143416968304675951743553132243626642071976084590242630174733922252912483663164280065528
7097505199750491300985946807101360233644016440017918861085323076499171437205446782359721176046515320016308533631935158964589068172237281231032027189791795127279965605369403
```

2111242846590994556380215461316106267521633805664394318881268199494005537068697621855231858921100963441012933535733918459668197539834284696822889460076352031688922002021931318
3697575569620611157743058263055358620156378912460312206729339926173783796251509999354036487314232088739779689089083699962929953919772177965334212492919783837514600620549673416
6283348734101109777053589806649813601139557158432830871394058253527405608101150390794168807919721293314830307263867863141103844312821599493682434299818871976863760449634259752
4256886188688978980888315865076262604856465004322896856149255063968811404400429503894245872382233543101078691517328333604779262727765686076177705616874050257743749983775830143
8561354272738385897741335269491654839297215195547935789238667625027453701046609093824496266269353213037445388924792161611888897020779104485631995148266308028795495464535838663
0734442375331971227915886170728965209014984830543598320077132665340729066201677570640969018377120130682324533347796666052532549087360196148037824156607127165038358225728921570
8209369510995890132859490724306183325755201208090007175022022949742801823445413711916298449914722254196594682221468260644961839254249670903104007581488857971672246322887016438
4039084638567311643081695373267903031145836805750211196399056151691547085104597005420985717973180155647414061723341458471112685479298924430013914682891036791792169786165824890
0732203359137670652767652130714398530276098847805621699465965546137917498565973922737941672649537780199209835542786617912312669937473077730569324430166839333011554515542656686
4937492128687049121754245967831132969248492466744261999033972825674873460201150442228780466124320183016108232183908654771042398228531316559685688005226571474428823317539456543
8819286244326625033453881995900851052113831244918618026244321955404339857228413412544094117717221568670862917421240531106205228429861992736294062088347548536451281232796090972
1395377536002307676569420821994303464878334854449271353945022459133437466493770165560576338469706291872574542650587941463017663976045747431108155674709165270874812526715991379
3240527304613693961169892589808311906322510777928562071999459487700611801002296132304588294558440952496611158342804908643860880796440557763691857743754025896855927252514563404
3852178258905995539546274513854544529167610429692679708935800562345019185714890304184957674008193599732187119574963570959678251710962647520688908064076514458931328707674541696
0710793169270428516809341331104635350624220981036321677191042078616218421376393819462569728678141363638962012397691046541895680619732314841422455007161721585132130203068417608
7215892702098879108938081045903397276547326416916845445627600759561367103584575649094430692452532085003091068783157561519847567569191284784654692558665111557913461272425336083
6351313421839051771545112284644551360160135132289485432715047608393075561009087860966638706122786902748318193316067014849571630047052622282384062668184487883745481319943803876
13830128859885264201992286188208499588640888521352501457615396482647451025902530743172956899636499615707551855583716593536712544851508936290456773663003556245737477910098799249
9146967224041481601289530944015488942613783140087804311431741858071826185149051138744831358439067228949408258286021650288927228387426432786168690381960530155894459451808735197
2460082215293439808282541261282571572093509853828007385604729109411840060844852353778335033068619777245018863640703449733664731006020181287928869918618244184539689947772594821
6913713364747045317297980924584436112961899759569624097184556402051143258959184472492094293030165148871307980210237906553652515478029805940752944051314580755153779486163587990
1158192019808879694967187448224156836463534326160242632934761634458163890163805123894184523973421841496889262398489648642093409816681494771155177009562669029850101513537599801
272501241971119871526593747484778935488778151929311714311674447738829410646150287513277094745047639228748906629898415402593508340351420351361688192482389980277066669163421334
2431205450735938866168769118818577611813577133248396520988208598239129860638682280475436240895652292141085985203733054462595326134023486468927506052689375514840329854208699122
1052597005628576707702567695300978970046408920009852106980295419699802138053295798159478289934443245491565327845223840551240445208226435420656313310702940722371552770504263482
0739844548895892488613976570791454144276535845729513297190919476944119109667974742626755909538320391696734942613600322630774286841050400613510521944137781580950057145268460098
1035210924904002795805073643696102124113773971716486952549311480504012656835126882959841398322267637780450062650724173175739521979689075482519932925964980162706866565803017887
7405615167159731927320479376247375505855052839660294566992522173600874081212014209071041937598571721431338017425141582491824710905084715977249417049320254165239323233258851588
8933370971363108925715314177619783260337501090262840664158013713593565292780884563059517700814439941146742918503607488523666547448699280832305168157116029118363741479584921008
6052898146954775081233889694315286102120273674704990393041703517134212692348670056662750622905863691188222890317051030540688209697087554532936943406398129769647803182545164217
8347347716471058423238594580183052756213910186997604305844068665712346869679456044155742100039179758348979935882751881524675930878928159243492197545387668305684668420775409821
7812470533545231947973989533201759886402810588255576980043971205383124594289573776960018574973352499650135093689259580218638117259065064368821271568157510217129007659927503702
2828396396291597325117341858672102349731776596945428362551937155600914368032931196284254662840314244437064843239037490641081130079284895576724348120009030988845727090775087363
8873299642555050473812528975962934822878917619920725138309388288292510416837622758204081918933603653875284116785703720989718832986921927816629675844580174911809119663048187434
1550677908639488314892415043004767045279712834822115222028370628573142441078237925136450866775666228049772113971406216641163247567842166129614771090188260946773776864061767214
8429389497667138012278894130902655351109611834701256519754080709538406091686393690667378662720942943426426040290215831734500372746258899262204987712117840556334849249032600350
8569099382392777297498413565614830788262363322368380709822346012274241379036473451735925215754757160934270935192901723954921426490691115271523338109124042812102893738488167358
9539345089306977155229891996989038858832754090443003219868340034702712200201596993716906503305475770953987485806700244910455048900617271891680313945280361656339415713346372225
5047754746075605502410876438212168884891694037125890194849068537972224456200948381949153272450227621858916950740579498375982100660448199651936011026157694717620257170204868491
4616894068404140833587562118319210838005632144562018941505945780025318747471911604840677997765414830622179069330853875129298983009580277554145435058768984944179136535891620098
7252220490551835546037065331831767161107380097866252474886914760776644701471930744763024116603356717655648744405779905319962716329720091094492492164560306188277729477507647774
4645258632891915910744425232008291820951802108370035388133098321589460868012795422475207192413464833496391509481309754143324420929993075148107791900234612812233016179942993061
8800533414550633932139339646861616416955220216447995417243171165744471364197733204899365074767844149929548073025856442942381787641506492878361767978677158510784235702640213388
0188756019892340568684232155856285086455252583770106205322242449879906252634840107743224881725586022333020763999338541520153438477254429178951306370503204449177977523708719582
7797679968611362653229111862963116468515993466069346055754595606315583003369763400027668515129384363888609082837614115773200352756515874590656702543943793110483857131329449060
4926582363108949535090082673154497226396648088618041573977888472892174618974189721700770009862449653759012727015227634510874906948012210684952063002519011655963580552429180205
5869042596852610474128345184667369385800277002529653563667216198836724282269339503259303909945831686655422346548570208755046175205218537215672826799034181355206029998953664701
065579005321295413369244724922124363245230428951884617791223380696742339806948872705875033892283950951352091231092581590069603515636773606710905056629960357187642324792075283
6160805597697778756476767210521222327184821484446631261487584226092608875764331731023263768864822594691211032367737558122133470556805958008310127481673962019583598023967414489

```
867276845869819376783757167936723213081586191045995058970991064686919463448038574143829629547131372173669836184558144505748676124322451519943362182916191468026091121793001864788050061351603144350076189213441602488091741051232290357179205497927970924502479940842696158818442616163780044759478212240873204124421169199805572649118243661921835714762891425805771871743688000324113008704819373962295017143090098476927237498875938639942530595331607891618810863505982444578942799346514915952884869757488025823353571677864826828051140885429732788197765736966005727700162592404301688659946862983717270595809808730901820120931003430058796552694788049809205484305467611034654748067290674399763612592434637719995843862812391985470202414880076880818848087892391591369463293113276849329777201646641727587259122354784480813433328050087758855264686119576962172239308693795757165821852416204341972383989932734803429262340722338155102209101262949249742423271698842023297303260161790575673111235465890298298313115123607606773968998153812286999642014609852579793691246016346088762321286205634215901479188632194659637483482564291616278532948239313229440231043277288768139550213348266388687453259281587854503890991561949632478855035090289390973718988003999026132015872678637873095678109625311008054489418857983565902063680699643165033912029944327726770869305240718416592070096139286401966725750087012218149733133695809600369751764951350040285926249203398111014953227533621844500744331562434532484217986108346261345897591234839970751854223281677187215956827243245910829019886390369784542622566912542747056097567984857136623679023878478161201477982939080513150258174523773529510165296934562786122241150783587755373348372764439838082000667214740034466322776918936967612878983488942094688102308427036452854504966759697318836044496702853190637396916357980928865719935397723495486787180416401415281489443785036291071517805285857583987711145474240156416477194116391354935466755593592608849200546384685403028080936417250583653368093407225310820844723570226809826951426162451204040711501448747856199922814664565893938488028643822313849852328452360667045805113679663751039248163336173274547275775636810977344539275827560597425160705468689657794530521602315939865780974801515414987097778078705357058008472376892422189750312758527140173117621279898744958406199843913365680297721208751934988504499713914285158032324823021340630312586072624541637765234505522051086318285359658520708173392709566445011404055106579055037417780393351658360904543047721422281816832539613634982525215232257690920254216409657452618066051777901592902884240599998882753691957540116954696152270401280857579766154722192925655963991820948894642657512288766330302133746367449217449351637104725732980832812726468187759356584218383594702792013663907689741738962252575782663990809792647011407580367850599381887184560094695833270775126181282015391041773950918244137561999937819240362469558235924171478702779448443108751901807414110290370706052085162975798361754251041642244867577350756338018895379263183389855955956527857227926155524494739363665533904528656215464288343162282921123290451842212532888101415884061619939195042230059898349966569463580186816717074818823215848647734386780911564660755175385552224428524049468033692299989300783900020690121517740696428573930196910500988278523053797637940257968953295112436166778910585557213381789089945453947915927374958600268237844486872037243488834616856290097850532497036933361942439802882364323553808208003875741710969289725499878566253048867033095150518452126944989251596392079421452606508516052325614861938282489838000815085351564642761700832096483117944401971780149213345335903336672376719229722069970766055482452247416927774637522135201716231722137632445699154022395494158227418930589911746931773776518735850032318014432883916374243795854695691221774098948611515564046609565094538115520921863711518684562543275047870530006998423140180169421109105925493596116719457630962328831271268328501760321771680400249657674186927113215573270049935709942324416387089242427584407651215572676037924765341808984312676941110313165951429479377670698881249643421933287404390485538222160837088907598277390184204138197811025854537088586701450623578513960109987476052535450100439353062072439709976445146790993381448994644609780957731953604938734950026860564555693224229691815630293922487606470873431166384205442489628760213650246991893040112513103835085621908060270866604873585849001704200923929789193938125116798421788115209259130435572321635660895603514383883939018953166274355609970015699780289236362349895374653428746875

ghci>
```

In the next two slides, **Stuart Halloway** shows us a **Clojure** version of the current **infinite stream** implementation, and in doing so, explains why we should be **careful** not to hold on to the **head** of a **lazy sequence**.

#8 infinite stream implementation (implicit definition)

Losing your head

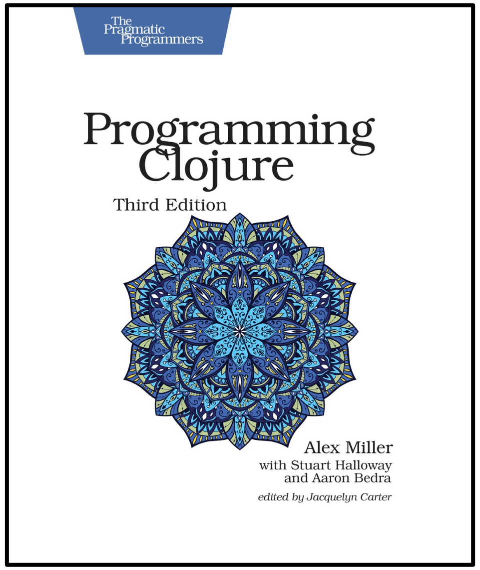
There's one last thing to consider when working with lazy sequences. Lazy sequences let you define a large (possibly infinite) sequence and then work with a small part of that sequence in memory at a given moment. This clever ploy will fail if you (or some API) unintentionally hold a reference to the part of the sequence you no longer care about.

The most common way this can happen is if you accidentally hold the head (first item) of a sequence. In the examples in previous sections, each variant of the Fibonacci numbers was defined as a function returning a sequence, not the sequence itself.

You could define the sequence directly as a top-level var:

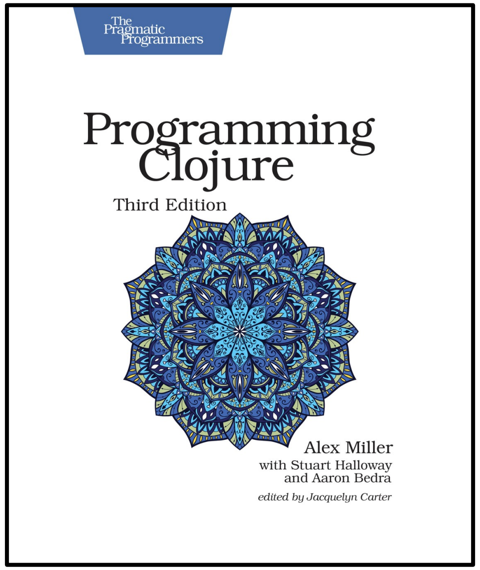
```
; holds the head (avoid!)
(defhead-fibo (lazy-cat [0N 1N] (map + head-fibo (rest head-fibo))))
```

This definition uses lazy-cat, which is like concat except that the arguments are evaluated only when needed. This is a very pretty definition, in that it defines the recursion by mapping a sum over (each element of the Fibonacci) and (each element of the *rest* of the Fibonaccis).



Stuart Halloway
stuarthalloway

#8 **infinite stream** implementation (**implicit definition**)



Stuart Halloway
stuarthalloway

head-fibo works great for small Fibonacci numbers:

```
  (take 10 head-fibo)
 -> [0N 1N 1N 2N 3N 5N 8N 13N 21N 34N]
```

but not so well for huge ones:

```
  (nth head-fibo 1000000)
  -> java.lang.OutOfMemoryError: GC overhead limit exceeded
```

The problem is that the top-level var head-fibo *holds the head* of the collection. This prevents the garbage collector from reclaiming elements of the sequence after you've moved past those elements. So, any part of the Fibonacci sequence that you actually use gets cached for the life of the value referenced by head-fibo, which is likely to be the life of the program.

Unless you want to cache a sequence as you traverse it, you must be careful not to keep a reference to the head of the sequence. As the head-fibo example demonstrates, you should normally expose lazy sequences as a function that *returns* the sequence, not as a var that *contains* the sequence. If a caller of your function wants an explicit cache, the caller can always create its own var. With lazy sequences, losing your head is often a good idea.

Computing the millionth **Fibonacci number** using the **Scala** version also runs out of **heap space**.

This is true of all the **Scala** implementations in this deck, so I am going to stop repeating it.

#8 **infinite stream** implementation (**implicit definition**)

```
scala> val fibs: LazyList[BigInt] =
     |   BigInt(0) #:: BigInt(1) #:: (fibs zip fibs.tail).map(_+_)
     |
val fibs: LazyList[BigInt] = LazyList(<not computed>)

scala> fibs(1_000_000)
java.lang.OutOfMemoryError: Java heap space
  at java.base/java.math.BigInteger.add(BigInteger.java:1425)
  at java.base/java.math.BigInteger.add(BigInteger.java:1331)
  at scala.math.BigInt.$plus(BigInt.scala:311)
  at rs$line$1$.$init$$$anonfun$1$$anonfun$1$$anonfun$1(rs$line$1:1)
  at rs$line$1$$$Lambda$1668/0x00000008015cc580.apply(Unknown Source)
  at scala.collection.immutable.LazyList.$anonfun$mapImpl$1(LazyList.scala:517)
  at scala.collection.immutable.LazyList$$Lambda$1671/0x00000008015cab18.apply(Unknown Source)
  at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state$lzycompute(LazyList.scala:259)
  at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state(LazyList.scala:252)
  at scala.collection.immutable.LazyList.isEmpty(LazyList.scala:269)
  at scala.collection.immutable.LazyList$.$anonfun$dropImpl$1(LazyList.scala:1073)
  at scala.collection.immutable.LazyList$$$Lambda$1663/0x000000080159f248.apply(Unknown Source)
  at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state$lzycompute(LazyList.scala:259)
  at scala.collection.immutable.LazyList.scala$collection$immutable$LazyList$$state(LazyList.scala:252)
  at scala.collection.immutable.LazyList.isEmpty(LazyList.scala:269)
  at scala.collection.LinearSeqOps.apply(LinearSeq.scala:131)
  at scala.collection.LinearSeqOps.apply$(LinearSeq.scala:128)
  at scala.collection.immutable.LazyList.apply(LazyList.scala:240)
  ... 14 elided

scala>
```

I didn't know that these days, computing the **Fibonacci sequence** is actually an example in the **Scala API documentation** for **LazyList** (see next slide).

LazyList

scala.collection.immutable.LazyList

Scala 3 3.6.2

This class implements an **immutable linked list**. We call it **"lazy"** because it computes its elements only when they are needed.

Elements are **memoized**; that is, the value of each element is computed at most once.

Elements are computed in-order and are never skipped. In other words, accessing the tail causes the head to be computed first.

How lazy is a `LazyList`? When you have a value of type `LazyList`, you don't know yet whether the list is empty or not. If you learn that it is non-empty, then you also know that the head has been computed. But the tail is itself a `LazyList`, whose emptiness-or-not might remain undetermined.

A `LazyList` may be **infinite**. For example, `LazyList`.from(0) contains all of the natural numbers 0, 1, 2, and so on. For **infinite sequences**, some methods (such as count, sum, max or min) will not terminate.

Here is an example:

```
import scala.math.BigInt
object Main extends App {
  val fibs: LazyList[BigInt] =
    BigInt(0) #:: BigInt(1) #:: fibs.zip(fibs.tail).map { n => n._1 + n._2 }
  fibs.take(5).foreach(println)
}
```

Note that the definition of **fibs** uses `val` not `def`. The `memoization` of the `LazyList` requires us to have somewhere to store the information and a `val` allows us to do that.

Further remarks about the **semantics** of `LazyList`:

- Though the `LazyList` changes as it is accessed, this does not contradict its **immutability**. Once the values are `memoized` they do not change. Values that have yet to be `memoized` still "exist", they simply haven't been computed yet.
- One must be cautious of **memoization**; it can eat up memory if you're not careful. That's because **memoization** of the **LazyList** creates a structure much like scala.collection.immutable.List. As long as something is holding on to the head, the head holds on to the tail, and so on recursively. If, on the other hand, there is nothing holding on to the head (e.g. if we used def to define the **LazyList**) then once it is no longer being used directly, it disappears.

#8 **infinite stream** implementation (**implicit definition**)

Computing the millionth **Fibonacci number** using the **Haskell** version does not exhaust **heap space**.

```
ghci> fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

ghci> (fibs !! (1000000)) > 1

True

ghci>
```

#8 infinite stream implementation (implicit definition)

As a recap, here are the Scheme, Clojure, Scala, and Haskell versions compared.

λ

```
(define fibs
  (cons-stream 0
               (cons-stream 1
                            (add-streams (stream-cdr fibs)
                                         fibs))))
```

```
(def fibs (lazy-cat [0N 1N] (map + fibs (rest fibs))))
```

```
val fibs: LazyList[BigInt] =
  BigInt(0) #:: BigInt(1) #:: (fibs zip fibs.tail).map(_+_)
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

#9 **infinite stream** implementation (**unfolding**)

Next, consider the **left-fold** based implementation from part 1. Note how **range** `(1 to i)` is only used to control the number of steps in the folding process. It is only the length of the **range** that matters, not its values. e.g. the **range** may just as well be `(-1 to -i)`.

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) =
  (1 to i).foldLeft(BigInt(0), BigInt(1))
    { case ((a, b), _) => (b, a + b) }
```

One way to eliminate the need for the **range** is to use the **unfold** function provided by **LazyList**.

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) =
  LazyList.unfold((BigInt(0), BigInt(1)))
    { case (a, b) => Some((a, b), (b, a + b)) }(i)
```

scala.collection.immutable

LazyList

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

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

A	Type of the elements
S	Type of the internal state
init	State initial value
f	Computes the next element (or returns `None` to signal the end of the collection)
returns	a lazy list that produces elements using `f` until `f` returns `None`

Definition Classes LazyList → IterableFactory

Next, we can **simplify** the **unfolding** implementation by using the **iterate** function provided by **LazyList**.

#10 **infinite stream** implementation (**iteration**)

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) =
  LazyList.unfold((BigInt(0), BigInt(1)))
    { case (a, b) => Some((a, b), (b, a + b)) }(i)
```

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) =
  LazyList.iterate((BigInt(0), BigInt(1)))
    { case (a, b) => (b, a + b) }(i)
```

scala.collection.immutable

LazyList

`def iterate[A](start: => A)(f: (A) => A): LazyList[A]`

An infinite LazyList that repeatedly applies a given function to a start value.

start	the start value of the LazyList
f	the function that's repeatedly applied
returns	the LazyList returning the infinite sequence of values `start, f(start), f(f(start)), ...`

Remember the first implementation of the **Fibonacci sequence** in this deck, which was based on an **implicit definition**?

```scala
val fibs: LazyList[BigInt] =
  BigInt(0) #:: BigInt(1) #:: (fibs zip fibs.tail).map(_+_)
```

#8 **infinite stream** implementation (**implicit definition**)

To conclude the deck, in the next four slides we look at another implementation that is also based on an **implicit definition**.

#11 **infinite stream** implementation (**scanning**)

Here we define the **Fibonacci sequence** simply as the result of consing **0** (the first **Fibonacci number**) onto the result of doing a **left scan** of the **sequence** itself with addition and an initial accumulator of 1.

```
fibs :: Num a => [a]
fibs = 0 : scanl (+) 1 fibs
```

Left scan function **`scanl`** is similar to **left fold** function **`foldl`**, but returns a list of successive reduced values from the left.

$$foldl \;::\; (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta$$
$$foldl\ f\ e\ [\,] = e$$
$$foldl\ f\ e\ (x : xs) = foldl\ f\ (f\ e\ x)\ xs$$

$$scanl \;::\; (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to [\beta]$$
$$scanl\ f\ e\ [\,] = [e]$$
$$scanl\ f\ e\ (x : xs) = e : scanl\ f\ (f\ e\ x)\ xs$$

$$foldl\ (\oplus)\ e\ [x_0, x_1, x_2]$$
$$\Downarrow$$
$$((e \oplus x_0) \oplus x_1) \oplus x_2$$

$$scanl\ (\oplus)\ e\ [x_0, x_1, x_2]$$
$$\Downarrow$$
$$[e, e \oplus x_0, (e \oplus x_0) \oplus x_1, ((e \oplus x_0) \oplus x_1) \oplus x_2]$$

Here is a less efficient, but easier to understand definition of a **left scan**. Given a function f, an initial accumulator e, and a **list** of items xs, $scanl$ first uses the $inits$ function to compute the **initial segments** of the given **list**, and then **maps** each resulting **segment** to the result of **folding** the **segment** using the given function f and the given initial accumulator e.

$$scanl \;::\; (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to [\beta]$$
$$scanl\ f\ e\ xs = map\ (foldl\ f\ e)(inits\ xs)$$

Here is an example of $inits$ being used to compute the **initial segments** of a small list.

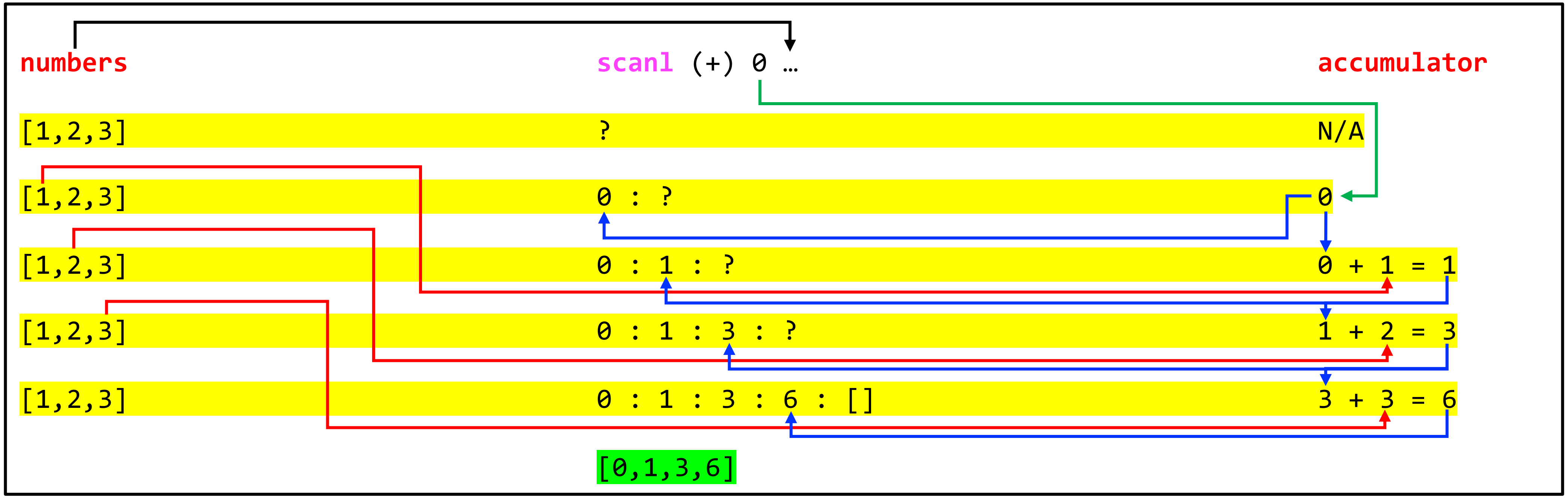
$$inits\ [x_0, x_1, x_2] = [[\,], [x_0], [x_0, x_1], [x_0, x_1, x_2]]$$

To understand how **fibs** works, on the next slide we go through the steps involved in the evaluation of

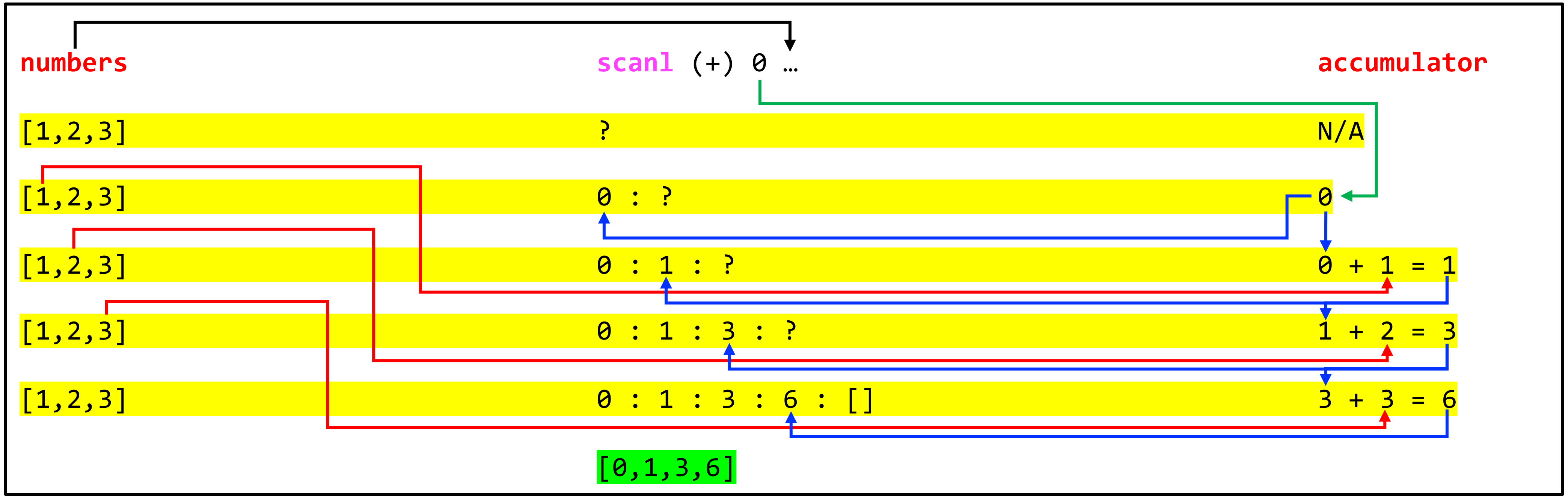
```
scanl (+) 0 [1,2,3]
```

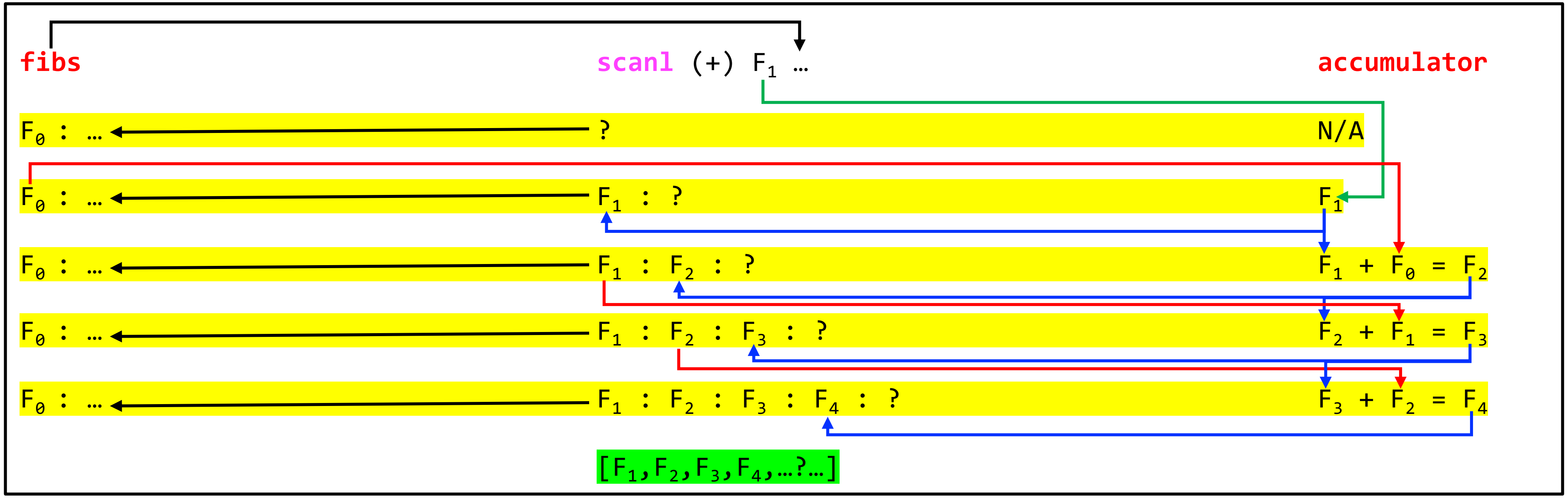
and on the subsequent slide we go through a similar exercise for the evaluation of

```
fibs = 0 : scanl (+) 1 fibs
```



numbers
scanl (+) 0 …
accumulator
[1,2,3]
?
N/A
[1,2,3]
0 : ?
0
[1,2,3]
0 : 1 : ?
0 + 1 = 1
[1,2,3]
0 : 1 : 3 : ?
1 + 2 = 3
[1,2,3]
0 : 1 : 3 : 6 : []
3 + 3 = 6
[0,1,3,6]





$F_0 = 0$
$F_1 = 1$
$F_2 = 1$
$F_3 = 2$
$F_4 = 3$
…

#11 infinite stream implementation (scanning)

Here is the Scala version of the Haskell implementation.

Scala function scan just delegates to function scanLeft.

```haskell
fibs :: Num a => [a]
fibs = 0 : scanl (+) 1 fibs
```

```scala
val fibs: LazyList[BigInt] =
  BigInt(0) #:: fibs.scan(BigInt(1))(_+_)
```

Scala 3 3.3.4

```scala
def scan[B >: B](z: B)(op: (B, B) => B): View[B]
```

Computes a prefix scan of the elements of the collection.

Note: The neutral element `z` may be applied more than once.

Type parameters

B	element type of the resulting collection

Value parameters

op	the associative operator for the scan
z	neutral element for the operator `op`

Attributes

Returns	a new iterable collection containing the prefix scan of the elements in this iterable collection
Inherited from:	IterableOps
Source	Iterable.scala

delegates to →

```scala
def scanLeft[B](z: B)(op: (B, A) => B): CC[B]
```

Produces a collection containing cumulative results of applying the operator going left to right, including the initial value.

Note: will not terminate for infinite-sized collections.

Note: might return different results for different runs, unless the underlying collection type is ordered.

Type parameters

B	the type of the elements in the resulting collection

Value parameters

op	the binary operator applied to the intermediate result and the element
z	the initial value

Attributes

Returns	collection with intermediate results
Source	IterableOnce.scala

To conclude this deck, the next slide contains the **Scala** version of the five implementations that we have seen in the deck.

```scala
def fibs: LazyList[BigInt] =
  fibgen(0, 1)

def fibgen(a: BigInt, b: BigInt): LazyList[BigInt] =
  a #:: fibgen(b, a + b)
```

#7 **infinite stream** implementation (**explicit generation**)

```scala
val fibs: LazyList[BigInt] =
  BigInt(0) #:: BigInt(1) #:: (fibs zip fibs.tail).map(_+_)
```

#8 **infinite stream** implementation (**implicit definition**)

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) =
  LazyList.unfold((BigInt(0), BigInt(1)))
    { case (a, b) => Some((a, b), (b, a + b)) }(i)
```

#9 **infinite stream** implementation (**unfolding**)

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) =
  LazyList.iterate((BigInt(0), BigInt(1)))
    { case (a, b) => (b, a + b) }(i)
```

#10 **infinite stream** implementation (**iteration**)

```scala
val fibs: LazyList[BigInt] =
  BigInt(0) #:: fibs.scan(BigInt(1))(_+_)
```

#11 **infinite stream** implementation (**scanning**)

The next slide is the same as the previous one, except that implementations #9 and #10 have been simplified so that they consist of a single **fibs** function with the same signature as the other implementations.

```scala
def fibs: LazyList[BigInt] =
  fibgen(0, 1)

def fibgen(a: BigInt, b: BigInt): LazyList[BigInt] =
  a #:: fibgen(b, a + b)
```

#7 **infinite stream** implementation (**explicit generation**)

```scala
val fibs: LazyList[BigInt] =
  BigInt(0) #:: BigInt(1) #:: (fibs zip fibs.tail).map(_+_)
```

#8 **infinite stream** implementation (**implicit definition**)

```scala
val fibs: LazyList[BigInt] =
  LazyList.unfold((BigInt(0), BigInt(1))){
    case (a, b) => Some((a, b), (b, a + b))
  }.map(_.first)
```

#9 **infinite stream** implementation (**unfolding**)

```scala
val fibs: LazyList[BigInt] =
  LazyList.iterate((BigInt(0), BigInt(1))){
    case (a, b) => (b, a + b)
  }.map(_.first)
```

#10 **infinite stream** implementation (**iteration**)

```scala
val fibs: LazyList[BigInt] =
  BigInt(0) #:: fibs.scan(BigInt(1))(_+_)
```

#11 **infinite stream** implementation (**scanning**)

That’s all for part 2.

I hope you found it useful.