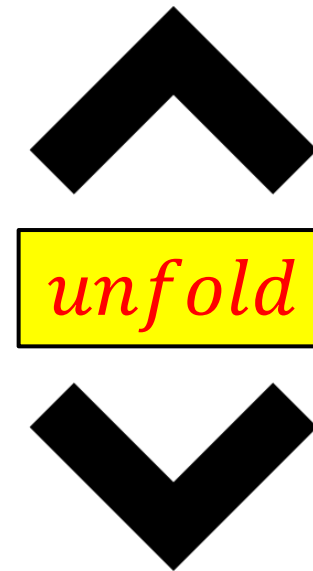
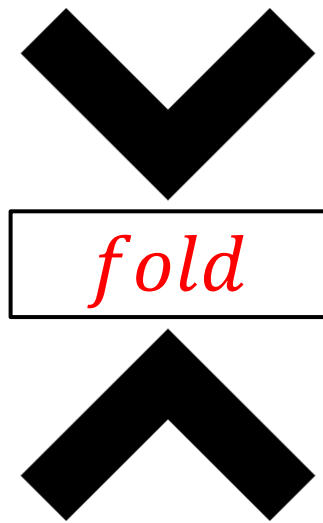


List Unfolding

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

 **Haskell**



 **Scala**

slides by



@philip_schwarz

FP Illuminated

<https://fpilluminated.org/>



  @philip_schwarz

In this deck we'll be looking at

- how **unfolding lists** is the **computational dual** of **folding lists**
- **different variants** of the function for **unfolding lists**
- how they relate to the **iterate** function



<https://fpilluminated.org/>



The next two slides are based on extracts from the third chapter of **The Fun of Programming**, which is called **Origami programming**, and is available at the following URL:
<https://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/origami.pdf>



Jeremy Gibbons

  @jer_gib



Oege de Moor

  @oegerikus

3.1 Introduction

...

In a precise technical sense, **folds** and **unfolds** are the **natural patterns of computation** over **recursive datatypes**; **unfolds generate data structures** and **folds consume** them.

Functional programmers are very familiar with the **foldr** function on **lists**, and its **directional dual foldl**; they are gradually coming to terms with the generalisation to **folds** on other **datatypes**.

The **computational duals, unfolds**, are still rather unfamiliar [45]; we hope to show here that they are no more complicated than, and just as useful as, **folds**, and to promote a style of programming based on these and similar recursion patterns (IFPH §3.3, §6.1.3, §6.4).

3.2 Origami with lists: sorting

...

Recall (IFPH §4.1.1) that **lists** may be defined explicitly via the following datatype declaration:

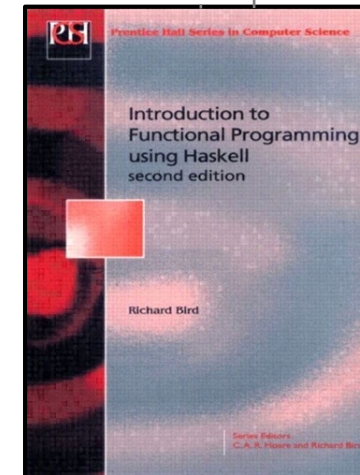
data *List* α = *Nil* | *Cons* α (*List* α)

As a concession to **Haskell**'s special syntax, we will define a function *wrap* for constructing **singleton lists**:

wrap $:: \alpha \rightarrow \text{List } \alpha$
wrap $x = \text{Cons } x \text{ Nil}$

We also define a function *nil* for detecting **empty lists**:

nil $:: \text{List } \alpha \rightarrow \text{Bool}$
nil Nil = *True*
nil (Cons x xs) = *False*



Folds for lists

The natural **fold** for **lists** may be defined as follows:

$$\begin{aligned} \text{foldL} &:: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \rightarrow \beta \\ \text{foldL } f \ e \ \text{Nil} &= e \\ \text{foldL } f \ e \ (\text{Cons } x \ xs) &= f \ x \ (\text{foldL } f \ e \ xs) \end{aligned}$$

This is equivalent to **Haskell's foldr** function; the '**L**' here is for '**list**', not for '**left**'.

...

Unfolds for lists

The dual of **folding** is **unfolding**. The **Haskell** standard **List** library defines the function **unfoldr** for **generating lists**:

$$\text{unfoldr} :: (\beta \rightarrow \text{Maybe } (\alpha, \beta)) \rightarrow \beta \rightarrow [\alpha]$$

Here, an instance of the type **Maybe** α may or may not have an instance of the type α :

$$\text{data Maybe } \alpha = \text{Just } \alpha \mid \text{Nothing}$$

We define an equivalent of **unfoldr** for our **list datatype**:

$$\begin{aligned} \text{unfoldL}' &:: (\beta \rightarrow \text{Maybe } (\alpha, \beta)) \rightarrow \beta \rightarrow \text{List } \alpha \\ \text{unfoldL}' \ f \ u &= \text{case } f \ u \ \text{of} \\ &\quad \text{Nothing} \rightarrow \text{Nil} \\ &\quad \text{Just } (x, v) \rightarrow \text{Cons } x \ (\text{unfoldL}' \ f \ v) \end{aligned}$$




Why is the name of the **unfold** function on the previous slide **primed**?

unfoldL'

In other words, why does the name end in ' ' ?

We'll come back to that question soon.

The next two slides show **Haskell** and **Scala documentation** for their equivalent of *unfoldL'*.

base-4.21.0.0: Core data structures and operations

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

Data.List

Unfolding

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

[# Source](#)

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

```
unfoldL' :: (β -> Maybe (α, β)) -> β -> List α
unfoldL' f u = case f u of
    Nothing -> Nil
    Just (x, v) -> Cons x (unfoldL' f v)
```



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



IterableFactory

scala.collection.IterableFactory



Scala 3 3.7.0



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

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

Type parameters

A	Type of the elements
S	Type of the internal state



Value parameters

f	Computes the next element (or returns <code>None</code> to signal the end of the collection)
init	State initial value

Attributes

Returns	a collection that produces elements using <code>f</code> until <code>f</code> returns <code>None</code>
Source	Factory.scala

unfoldL' :: ($\beta \rightarrow \text{Maybe}(\alpha, \beta)$) $\rightarrow \beta \rightarrow \text{List } \alpha$
unfoldL' f u = **case** f u **of**
 Nothing $\rightarrow \text{Nil}$
 Just (x, v) $\rightarrow \text{Cons } x (\text{unfoldL'} f v)$

 = unfoldr  = unfold

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





Let's look at some examples of **unfolding**.



Our **first example** of using *unfoldL* is a **range function** which, given an integer **lower bound** and an integer **upper bound**, **generates** the **sequence** of integers going from the **lower bound** to the **upper bound**, inclusive of both **bounds**.

In this **example**, the **items** in the **generated list** are the initial **seed value**, and the subsequent **seed values generated** by *unfoldL*.

```
range :: Int -> Int -> [Int]
range lwb upb = unfoldr gen lwb
  where gen n = if n > upb then Nothing
               else Just (n, n + 1)
```



```
> range 20 30
[20,21,22,23,24,25,26,27,28,29,30]
```

```
val range: Int => Int => List[Int] = lwb => upb =>
  List.unfold(lwb): n =>
    if n > upb then None
    else Some(n, n + 1)
```



```
scala> range(20)(30)
val res0: List[Int] = List(20,21,22,23,24,25,26,27,28,29,30)
```

Let's build on the above **example** to provide an **example** of the **dual processes** of **folding** and **unfolding** being **used together**. Let's compute the **factorial** of **N** by first **unfolding** and then **folding**.

```
multiply :: [Int] -> Int
multiply = foldr (*) 1
```

```
fact = multiply . range 1
```

```
> fact 10
3628800
```



```
val multiply: List[Int] => Int =
  _.foldRight(1)(_ * _)
```

```
val fact = multiply compose range(1)
```

```
scala> fact(10)
3628800
```





Our **second example** of using *unfoldL'* is a **sumOfSquares function** which, given an integer **N**, **generates** the **sum** of the **squares** of the integers in the **range** 1..**N**, and then **sums** the **squares**.

In this **example**, the **items** in the **generated list** are the **squares** of the initial **seed** (or state) **value** and of the subsequent **seed values** generated by *unfoldL'*.

```
squares :: Int -> [Int]
squares = unfoldr gen
  where gen 0 = Nothing
        gen n = Just (n * n, n - 1)
```

```
summation :: [Int] -> Int
summation = foldr (+) 0
```

```
sumOfSquares :: Int -> Int
sumOfSquares = summation . squares
```

```
> sumOfSquares 5
55
```



```
val squares: Int => List[Int] =
  List.unfold(_):
    case 0 => None
    case n => Some(n * n, n - 1)
```

```
val summation: List[Int] => Int =
  _.foldRight(0)(_+_)
```

```
val sumOfSquares: Int => Int =
  summation compose squares
```

```
scala> sumOfSquares(5)
55
```



Same **example**, but **refactored** to highlight the fact that we are **composing** a **fold** with an **unfold**.

```
sumOfSquares = foldr (+) 0 . unfoldr gen
  where gen 0 = Nothing
        gen n = Just(n * n, n - 1)
```

```
> sumOfSquares 5
55
```



```
val sumOfSquares: Int => Int =
  val gen: Int => Option[(Int, Int)] =
    case 0 => None
    case n => Some(n * n, n - 1)
  ((n: Int) => List.unfold(n)(gen)) andThen (_.foldRight(0)(_+_))
```

```
scala> sumOfSquares(5)
55
```





As an aside, **Scala** functions **range** and **squares** may alternatively be written as follows

```
val range: Int => Int => List[Int] = lwb => upb =>  
  List.unfold(lwb): n =>  
    if n > upb then None  
    else Some(n, n + 1)
```



```
val range: Int => Int => List[Int] = lwb => upb =>  
  List.unfold(lwb): n =>  
    Option.unless(n > upb)(n, n + 1)
```

```
val squares: Int => List[Int] =  
  List.unfold(_):  
    case 0 => None  
    case n => Some(n * n, n - 1)
```



```
val squares: Int => List[Int] =  
  List.unfold(_): n =>  
    Option.unless(n == 0)(n * n, n - 1)
```

In the example that we have just seen, the **function f** passed to the **unfold function** keeps returning a **Just (Some)** until a **condition** is met, at which point it returns a **Nothing (None)**.

But what stops **function f** from never returning a **Nothing (None)**? What happens if we **unfold** with such a **function**?

In **Haskell**, it can make sense for **function f** never to return **Nothing (None)**, because **Haskell lists** are **lazy**, so **f** can be used to generate an **infinite list**.

In the following **Haskell** example, we first use **foldr** to **generate** an **infinite list** of **Fibonacci numbers**, and then **take** the first 10 such numbers.

```
> take 10 (unfoldr (\(a,b) -> Just (a,(b,a+b))) (0,1))  
[0,1,1,2,3,5,8,13,21,34]
```

In **Scala** however, **lists** are **not lazy**, so it does not make sense to attempt to **generate** an **infinite list**, because doing so would take forever, if it weren't for the fact that the **list** keeps growing, and so eventually uses up all the available **heap space**.

```
> List.unfold((0,1)){ case (a,b) => Some((a,(b,a+b))) }.take(10)  
java.lang.OutOfMemoryError: Java heap space
```

The way we avoid the above problem is by **unfolding** a **lazy version** of a **list**:

```
> LazyList.unfold((0,1)){ case (a,b) => Some((a,(b,a+b))) }.take(10).toList  
val res0: List[Int] = List(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
```





Now that we have seen some examples of using the *unfoldL'* function, let's go back to the following question:

Why is the name of the function **primed**: why does the name end in ' ' ?

See the next two slides for the answer.

$unfoldL' :: (\beta \rightarrow \text{Maybe } (\alpha, \beta)) \rightarrow \beta \rightarrow \text{List } \alpha$
 $unfoldL' f u = \text{case } f u \text{ of}$
 $\text{Nothing} \rightarrow \text{Nil}$
 $\text{Just } (x, v) \rightarrow \text{Cons } x (unfoldL' f v)$

\gg unfoldr

 unfold

Sometimes it is convenient to provide the single argument of $unfoldL'$ as three components: a predicate indicating when that argument should return Nothing , and two functions yielding the two components of the pair when it does not.

The resulting function $unfoldL$ takes a predicate p indicating when the seed should unfold to the empty list, and for when this fails to hold, functions f giving the head of the list and g giving the seed from which to unfold the tail:

$unfoldL :: (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha$
 $unfoldL p f g b = \text{if } p b \text{ then Nil else Cons } (f b) (unfoldL p f g (g b))$

Conversely, one could define a function $foldL'$ taking a single argument of type $\text{Maybe } (\alpha, \beta) \rightarrow \beta$ in place of $foldL$'s two arguments:

$foldL' :: (\text{Maybe } (\alpha, \beta) \rightarrow \beta) \rightarrow \text{List } \alpha \rightarrow \beta$
 $foldL' f \text{ Nil} = f \text{ Nothing}$
 $foldL' f (\text{Cons } x xs) = f (\text{Just } (x, foldL' f xs))$

These primed versions make the duality between the fold and the unfold very clear, although they may sometimes be less convenient for programming with.

$foldL :: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \rightarrow \beta$
 $foldL f e \text{ Nil} = e$
 $foldL f e (\text{Cons } x xs) = f x (foldL f e xs)$

\gg foldr

 foldRight

cornerstones of computing
series editors: richard bird & tony hoare

the fun of
programming

jeremy gibbons
and oege de moor





The next slide aims to **reinforce** the notion of **duality** seen on the previous slide.

While they may sometimes be less convenient for programming with, *foldL'* and *unfoldL'*, the **primed** versions of *foldL* and *unfoldL*, make the **duality** between the **fold** and the **unfold** very clear



 foldr  foldRight

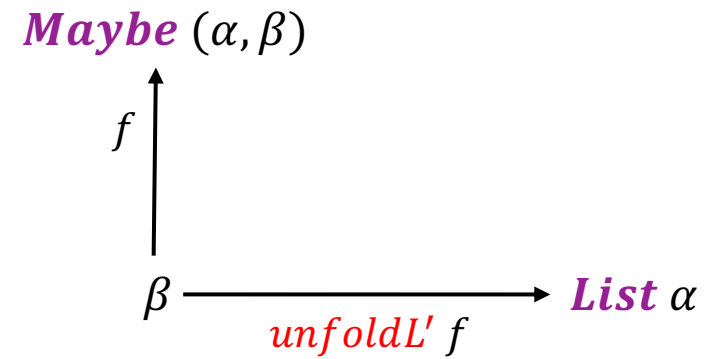
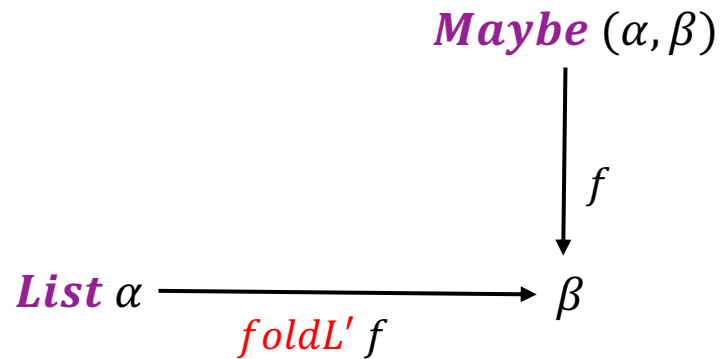
foldL :: $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \rightarrow \beta$
foldL *f* *e* *Nil* = *e*
foldL *f* *e* (*Cons* *x* *xs*) = *f* *x* (*foldL* *f* *e* *xs*)

foldL' :: $(\text{Maybe } (\alpha, \beta) \rightarrow \beta) \rightarrow \text{List } \alpha \rightarrow \beta$
foldL' *f* *Nil* = *f* *Nothing*
foldL' *f* (*Cons* *x* *xs*) = *f* (*Just* (*x*, *foldL'* *f* *xs*))

unfoldL :: $(\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha$
unfoldL *p* *f* *g* *b* = **if** *p* *b* **then** *Nil*
 else *Cons* (*f* *b*) (*unfoldL* *p* *f* *g* (*g* *b*))

unfoldL' :: $(\beta \rightarrow \text{Maybe } (\alpha, \beta)) \rightarrow \beta \rightarrow \text{List } \alpha$
unfoldL' *f* *u* = **case** *f* *u* **of**
 Nothing → *Nil*
 Just (*x*, *v*) → *Cons* *x* (*unfoldL'* *f* *v*)

 unfoldr  unfold





The next slide is based on slightly modified versions of two sentences from the paper **Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire**, which is available here <https://maartenfokkinga.github.io/utwente/mmf91m.pdf>

 foldr  foldRight

foldL :: $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \rightarrow \beta$



Functions that ‘**destruct**’ a **list** – they have been called **catamorphisms**, from the **Greek** proposition **κατά**, meaning ‘**downwards**’ as in ‘**catastrophe**’.

foldL' :: $(\text{Maybe } (\alpha, \beta) \rightarrow \beta) \rightarrow \text{List } \alpha \rightarrow \beta$

unfoldL :: $(\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha$

Functions that ‘**generate**’ a **list** of **items** of type α from a **seed** of type β – they have been called **anamorphisms**, from the **Greek** proposition **ἀνά**, meaning ‘**upwards**’ as in ‘**anabolism**’.

unfoldL' :: $(\beta \rightarrow \text{Maybe } (\alpha, \beta)) \rightarrow \beta \rightarrow \text{List } \alpha$

 unfoldr  unfold



unfoldL and *unfoldL'* are not the only **functions** that can be used to **unfold lists**.

There is also *iterate*.

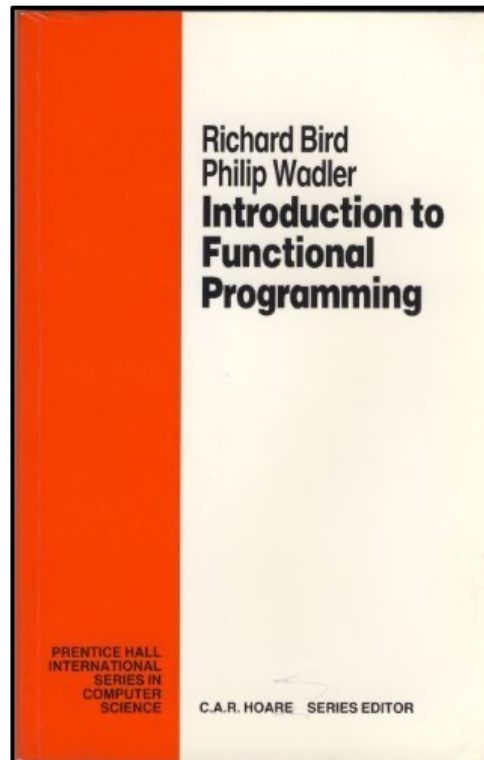
The next slide shows how **Introduction to Functional Programming** introduces the *iterate* function.

The subsequent slide shows how *iterate* is implemented, and the slide after that shows how the **Haskell** API documentation describes the function.



Richard Bird

<http://www.cs.ox.ac.uk/people/richard.bird/>



Philip Wadler

<https://github.com/wadler>

7.2 Iterate

Recall that in mathematics, the notation f^n denotes a **function composed** with **itself** n times; thus $f^0 = id$, $f^1 = f$, $f^2 = f \circ f$, $f^3 = f \circ f \circ f$ and so on, where id is the **identity function**. In our notation we will write f^n as *(power f n)*.

One way to define *power* is by the equations:

$$\text{power } f \ 0 = id$$

$$\text{power } f \ (n + 1) = f \circ \text{power } f \ n$$

Observe that f^n is similar in form to x^n , which denotes a number multiplied by itself n times, but one should be careful not to confuse the two. The function *iterate* is defined **informally** as follows:

$$\text{iterate } f \ x = [x, \ f \ x, \ f^2 x, \ f^3 x, \dots]$$

Thus *iterate* takes a **function** and a **starting value** and returns an **infinite list**.

For example:

$$\text{iterate } (+1) \ 1 = [1, \ 2, \ 3, \ 4, \ 5, \dots]$$

$$\text{iterate } (\times 2) \ 1 = [1, \ 2, \ 4, \ 8, \ 16, \ 32, \dots]$$

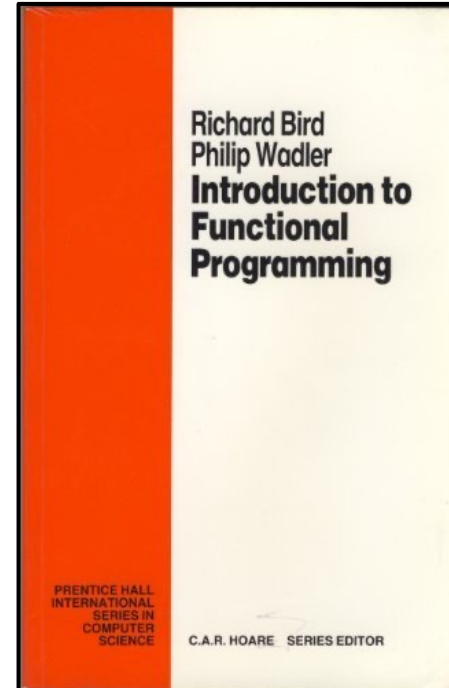
$$\text{iterate } (\text{div } 10) \ 2718 = [2718, \ 271, \ 27, \ 2, \ 0, \ 0, \dots]$$

We also have:

$$[m..] = \text{iterate } (+1) \ m$$

$$[m..n] = \text{takeWhile } (\leq n) \ (\text{iterate } (+1) \ m)$$

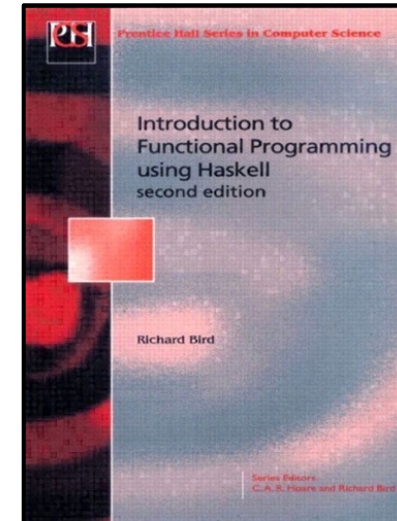
These **equations** provide one way of **defining** the **notations** $[m..]$ and $[m..n]$. In the second **equation**, *takeWhile* is used to **truncate** the **infinite list** to a **finite list**.





The definition of the *iterate* function on the previous slide was **informal**. Here is a **formal definition** (there are also alternative ones).

Produces an **infinite list** of **iterated applications** of a function to a value.

$$\textit{iterate} :: (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha]$$
$$\textit{iterate} f x = x : \textit{iterate} f (f x)$$


base-4.21.0.0: Core data structures and operations

List operations

Building lists

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

Infinite lists

```
iterate :: (a -> a) -> a -> [a]
```

[# Source](#)

`iterate` `f` `x` returns an infinite list of repeated applications of `f` to `x`:

```
iterate f x == [x, f x, f (f x), ...]
```

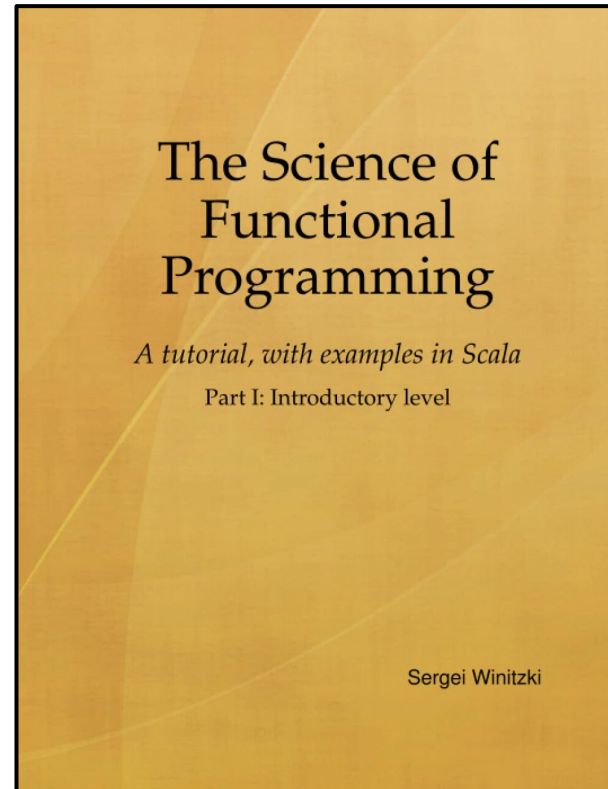
```
iterate :: ( $\alpha \rightarrow \alpha$ )  $\rightarrow \alpha \rightarrow [\alpha]$   
iterate f x = x : iterate f (f x)
```



It turns out that *iterate* is just a special case of *unfoldL* and *unfoldL'*.

The way we are going to **illustrate** this is by looking at a *digits* function which given a **positive integer number**, returns a **list** of the number's **digits**.

In the next three slides, we are going to see how, in **The Science of Functional Programming**, **Sergei Winitzki** introduces *iterate* and uses it to implement *digits* (which he calls **digitsOf**).



Sergei Winitzki

 [sergei-winitzki-11a6431](https://www.linkedin.com/in/sergei-winitzki-11a6431)

2.3 Generating a sequence from a single value

An **aggregation converts** (“folds”) a **sequence** into a **single value**; the **opposite operation** (“unfolding”) **builds** a **new sequence** from a **single value** and **other needed information**. An example is computing the **decimal digits** of a given integer:

```
def digitsOf(x: Int): Seq[Int] = ???
```

```
scala> digitsOf(2405)
res0: Seq[Int] = List(2, 4, 0, 5)
```

We cannot implement **digitsOf** using **map**, **zip**, or **foldLeft**, because these methods work only if we already have a **sequence**; but the function **digitsOf** needs to create a **new sequence**. We could create a **sequence** via the expression (1 to n) if the required length of the **sequence** were known in advance. However, the function **digitsOf** must produce a **sequence** whose length is determined by a condition that we cannot easily evaluate in advance.

A general “**unfolding**” operation needs to build a **sequence** whose length is not determined in advance. This kind of **sequence** is called a **stream**. The elements of a **stream** are computed only when necessary (unlike the elements of **List** or **Array**, which are all computed in advance). The **unfolding operation** will compute next elements **on demand**; this creates a **stream**. We can then apply **takeWhile** to the **stream**, in order to stop it when a certain condition holds. Finally, if required, the truncated **stream** may be converted to a **list** or another type of **sequence**. In this way, we can **generate** a **sequence** of initially unknown length according to any given requirements.

The **Scala library** has a general **stream-producing** function **Stream.iterate**². This function has two arguments, the **initial value** and a **function** that computes the **next value** from the **previous one**:

```
scala> Stream.iterate(2) { x => x + 10 }
res0: Stream[Int] = Stream(2, ?)
```

² In a future version of Scala 3, the **Stream** class will be replaced by **LazyList**.

The **stream** is ready to start computing the **next elements** of the **sequence** (so far, only the first element, 2, has been computed).

The Science of Functional Programming

A tutorial, with examples in Scala
Part I: Introductory level

Sergei Winitzki

In order to see the **next elements**, we need to **stop** the **stream** at a **finite size** and then convert the result to a **list**:

```
scala> Stream.iterate(2) { x => x + 10 }.take(6).toList
res1: List[Int] = List(2, 12, 22, 32, 42, 52)
```

If we try to evaluate **toList** on a **stream** without first **limiting** its **size** via **take** or **takeWhile**, the program will keep producing more elements until it **runs out of memory** and **crashes**. **Streams** are similar to **sequences**, and methods such as **map**, **filter**, and **flatMap** are also defined for **streams**. For instance, the method **drop** skips a given number of initial elements:

```
scala> Seq(10, 20, 30, 40, 50).drop(3)
res2: Seq[Int] = List(40, 50)

scala> Stream.iterate(2) { x => x + 10 }.drop(3)
res3: Stream[Int] = Stream(32, ?)
```

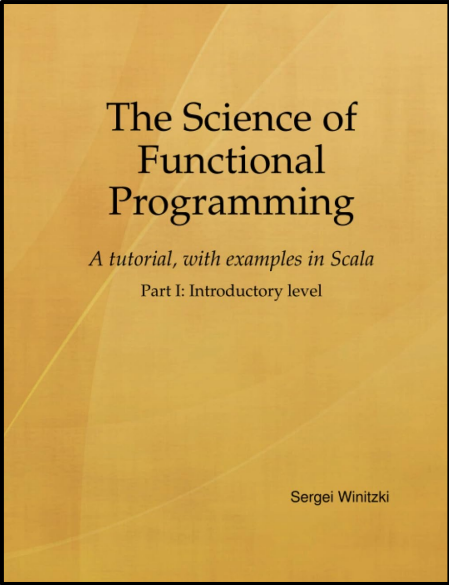
To figure out the code for **digitsOf**, we first write this function as a **mathematical formula**. To compute the digits for, say, **n** = **2405**, we need to divide **n** repeatedly by **10**, getting a **sequence** **n_k** of intermediate numbers (**n₀** = **2405**, **n₁** = **240**, ...) and the corresponding sequence of last digits, **n_k mod 10** (in this example: **5**, **0**, ...). The sequence **n_k** is defined using **mathematical induction**:

- **Base case:** **n₀** = **n**, where **n** is the given initial integer.
- **Inductive step:** **n_{k+1}** = $\left\lfloor \frac{n_k}{10} \right\rfloor$ for **k** = 1, 2, ...

Here $\left\lfloor \frac{n_k}{10} \right\rfloor$ is the mathematical notation for the integer division by **10**. Let us tabulate the evaluation of the **sequence** **n_k** for **n** = **2405**:

<i>k</i> =	0	1	2	3	4	5	6
<i>n_k</i> =	2405	240	24	2	0	0	0
<i>n_k</i> mod 10 =	5	0	4	2	0	0	0

The numbers **n_k** will remain all zeros after **k** = **4**. It is clear that the useful part of the **sequence** is before it becomes all zeros. In this example, the **sequence** **n_k** needs to be stopped at **k** = **4**. The **sequence** of digits then becomes [**5**, **0**, **4**, **2**], and we need to reverse it to obtain [**2**, **4**, **0**, **5**]. For reversing a **sequence**, the **Scala** library has the standard method **reverse**.



So, a complete implementation for **digitsOf** is:

```
def digitsOf(n: Int): Seq[Int] =  
  if (n == 0) Seq(0) else { // n == 0 is a special case.  
    Stream.iterate(n) { nk => nk / 10 }  
      .takeWhile { nk => nk != 0 }  
      .map { nk => nk % 10 }  
      .toList.reverse  
  }
```

We can shorten the code by using the syntax such as `(_ % 10)` instead of `{ nk => nk % 10 }`,

```
def digitsOf(n: Int): Seq[Int] =  
  if (n == 0) Seq(0) else { // n == 0 is a special case.  
    Stream.iterate(n) (_ / 10 )  
      .takeWhile ( _ != 0 )  
      .map ( _ % 10 )  
      .toList.reverse  
  }
```

The type signature of the method **Stream.iterate** can be written as

```
def iterate[A](init: A)(next: A => A): Stream[A]
```

and shows a close correspondence to a definition by **mathematical induction**. The **base case** is the first value, **init**, and the **inductive step** is a function, **next**, that computes the **next element** from the **previous** one. It is a general way of creating **sequences** whose length is not determined in advance.

The Science of Functional Programming

A tutorial, with examples in Scala

Part I: Introductory level

Sergei Winitzki



The next slide shows how the **Scala API documentation** describes the *iterate* function.



scala.collection.immutable.LazyList



Scala 3 3.7.0



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

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

Value parameters

f

the function that's repeatedly applied

start

the start value of the LazyList

Attributes

Returns

the LazyList returning the infinite sequence of values `start, f(start), f(f(start)), ...`

Source

[LazyList.scala](#)



Earlier we saw that when **Introduction to Functional Programming** introduces the *iterate* function, it provides some **simple examples** of its usage.

As shown on the next slide, the next thing the book does is introduce **two more interesting examples**, the first of which is the *digits* function.

Here are some more example of the use of **iterate**. First, the **digits** of a **positive integer** can be **extracted** by a **function digits** defined as follows:

$$\text{digits} = \text{reverse} \circ \text{map}(\text{mod } 10) \circ \text{takewhile } (/= 0) \circ \text{iterate}(\text{div } 10)$$

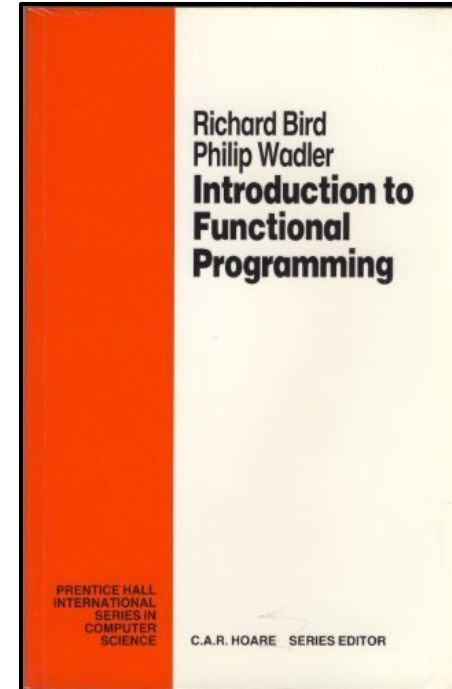
For example,

$$\begin{aligned} \text{digits } 2718 &= (\text{reverse} \circ \text{map}(\text{mod } 10) \circ \text{takewhile } (/= 0)) [2718, 271, 27, 2, 0, 0, \dots] \\ &= (\text{reverse} \circ \text{map}(\text{mod } 10)) [2718, 271, 27, 2] \\ &= \text{reverse } [8, 1, 7, 2] \\ &= [2, 7, 1, 8] \end{aligned}$$

Next, consider the function (*group n*) which breaks a list into segments of length *n*.

$$\text{group } n = \text{map}(\text{take } n) \circ \text{takewhile } (/= []) \circ \text{iterate}(\text{drop } n)$$

If the original list does not have a length that is evenly divisible by *n*, then the last segment will have length strictly less than *n*.





Here are the **Haskell** and **Scala** versions of the *digits* function next to each other



```
digits :: Int -> [Int]
digits = reverse . map (`mod` 10) . takeWhile (/= 0) . iterate (`div` 10)
```

```
> digits 2718
[2,7,1,8]
```



```
def digits(n: Int): List[Int] =
  LazyList.iterate(n)(_ / 10).takeWhile(_ != 0).map(_ % 10).toList.reverse
```

```
scala> digits(2718)
val res0: List[Int] = List(2,7,1,8)
```



Similarly for the *group* function.



```
group :: Eq a => Int -> [a] -> [[a]]
group n = map (take n) . takeWhile (/= []) . iterate (drop n)

> group 3 [1,2,3,4,5,6,7,8,9,10,11]
[[1,2,3],[4,5,6],[7,8,9],[10,11]]
```



```
def group[A](n: Int, as: List[A]): List[List[A]] =
  LazyList.iterate(as)(_ . drop(n)).takeWhile(_ . nonEmpty).map(_ . take(n)).toList

scala> group(3, List(1,2,3,4,5,6,7,8,9,10,11))
val res16: List[List[Int]] = List(List(1,2,3),List(4,5,6),List(7,8,9),List(10,11))
```

```
iterate :: ( $\alpha \rightarrow \alpha$ )  $\rightarrow \alpha \rightarrow [\alpha]$ 
iterate f x = x : iterate f (f x)
```




On the next slide, we see **Introduction to Functional Programming** introducing the **unfold** function, which is spoken of **speculatively**, and not mentioned again in the book.

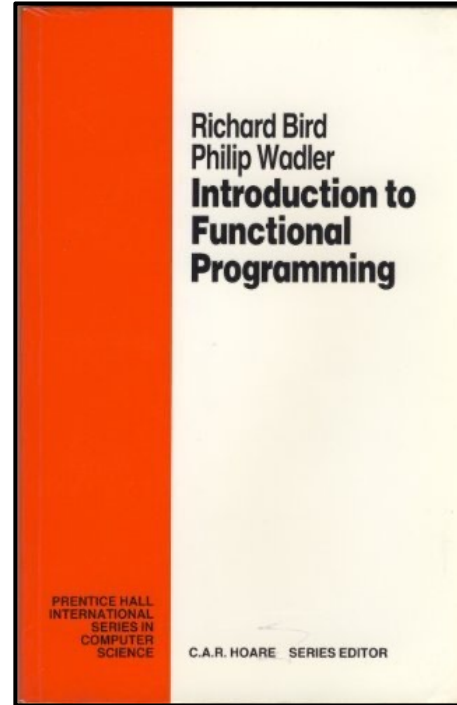
As the last two examples suggest, one often finds *map*, *takewhile* and *iterate* composed together in **sequence**. Suppose we **capture** this **pattern of computation** as a **generic function**, *unfold* say, defined as follows:

$$\text{unfold } h \ p \ t = \text{map } h \circ \text{takewhile } p \circ \text{iterate } t$$

The key feature about *unfold* is that it is a general **function for producing lists**. Moreover, the functions *h*, *t* and *p* correspond to simple operations on **lists**. We have

$$\begin{aligned} \text{hd } (\text{unfold } h \ p \ t \ x) &= h \ x \\ \text{tl } (\text{unfold } h \ p \ t \ x) &= \text{unfold } h \ p \ t \ (t \ x) \\ \text{nonnull } (\text{unfold } h \ p \ t \ x) &= p \ x \end{aligned}$$

Thus, the first argument *h* of *unfold* is a function that corresponds to *hd*, the third function, *t*, corresponds to *tl*, and the second function, *p*, to a predicate which tests whether the list is empty or not.



unfold :: $(\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha$
unfold *h p t* = *map h* ∘ *takewhile p* ∘ *iterate t*



Let's implement the **unfold** function. We'll use it in the next two slides.

unfold :: (b -> a) -> (b -> Bool) -> (b -> b) -> b -> [a]
unfold h p t = map h . takeWhile p . **iterate** t



```
def unfold[A,B](h: B => A, p: B => Boolean, t: B => B, b: B): LazyList[A] =  
  LazyList.iterate(b)(t).takeWhile(p).map(h)
```





Remember the *digits* function implemented a few slides ago using the *iterate* function?

```
digits :: Int -> [Int]
digits = reverse . map (`mod` 10) . takeWhile (/= 0) . iterate (`div` 10)
```



```
def digits(n: Int): List[Int] =
  LazyList.iterate(n)(_ / 10).takeWhile(_ != 0).map(_ % 10).toList.reverse
```



Let's reimplement it using *unfold*. The result is much simpler thanks to *unfold* capturing the pattern of computation involving *map*, *takeWhile* and *iterate*.



```
> digits = reverse . unfold (`mod` 10) (/= 0) (`div` 10)
> digits 2718
[2,7,1,8]
```



```
scala> def digits(n: Int): List[Int] =
      |   unfold[Int,Int](_ % 10, _ != 0, _ / 10, n).toList.reverse
      |
def digits(n: Int): List[Int]

scala> digits(2718)
val res0: List[Int] = List(2, 7, 1, 8)
```



unfold :: $(\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha$
unfold *h p t* = *map h* ◦ *takeWhile p* ◦ *iterate t*



Remember the *group* function implemented a few of slides ago using the *iterate* function?

```
group :: Eq a => Int -> [a] -> [[a]]
group n = map (take n) . takeWhile (/= []) . iterate (drop n)
```



```
def group[A](n: Int, as: List[A]): List[List[A]] =
  LazyList.iterate(as)(_.drop(n)).takeWhile(_.nonEmpty).map(_.take(n)).toList
```



Let's reimplement it using *unfold*. Again, the result is much simpler thanks to *unfold* capturing the pattern of computation involving *map*, *takeWhile* and *iterate*.



```
> group n = unfold (take n) (/= []) (drop n)
> group 3 [1,2,3,4,5,6,7,8,9,10,11]
[[1,2,3],[4,5,6],[7,8,9],[10,11]]
```



```
def group[A](n: Int, as: List[A]): List[List[A]] =
  unfold[List[A], List[A]](_.take(n), _.nonEmpty, _.drop(n), as).toList

scala> group(3, List(1,2,3,4,5,6,7,8,9,10,11))
val res0: List[List[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9), List(10, 11))
```



unfold :: $(\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha$
unfold *h p t* = *map h* \circ *takeWhile p* \circ *iterate t*



We have seen that *unfold* is defined in terms of *map*, *takewhile* and *iterate*.

What about defining *iterate* in terms of *unfold*?

It looks like this should work, right?

```
iterate :: ( $\alpha \rightarrow \alpha$ )  $\rightarrow$   $\alpha \rightarrow [\alpha]$   
iterate = unfold id (const True)
```

Yes, it does:

```
iterate' :: (a -> a) -> a -> [a]  
iterate' = unfold id (const True)
```

```
> take 10 (iterate' (+1) 1)  
[1,2,3,4,5,6,7,8,9,10]
```



```
unfold :: ( $\beta \rightarrow \alpha$ )  $\rightarrow$  ( $\beta \rightarrow \mathbf{Bool}$ )  $\rightarrow$  ( $\beta \rightarrow \beta$ )  $\rightarrow$   $\beta \rightarrow \mathbf{List} \alpha$   
unfold h p t = map h  $\circ$  takewhile p  $\circ$  iterate t
```



It looks like apart from some **renaming** and **reordering** of **parameters**, and the **inversion** of the **condition** tested by **predicate function** p , the $unfold$ function is equivalent to the $unfoldL$ function.



$$\begin{aligned} &unfold :: (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \\ &unfold\ h\ p\ t = \text{map } h \circ \text{takeWhile } p \circ \text{iterate } t \end{aligned}$$


$$\begin{aligned} &unfoldL :: (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \\ &unfoldL\ p\ f\ g\ b = \text{if } p\ b \text{ then Nil} \\ &\quad \text{else Cons } (f\ b) (unfoldL\ p\ f\ g\ (g\ b)) \end{aligned}$$


$$\begin{aligned} &unfoldL' :: (\beta \rightarrow \text{Maybe } (\alpha, \beta)) \rightarrow \beta \rightarrow \text{List } \alpha \\ &unfoldL'\ f\ u = \text{case } f\ u \text{ of} \\ &\quad \text{Nothing} \rightarrow \text{Nil} \\ &\quad \text{Just } (x, v) \rightarrow \text{Cons } x (unfoldL'\ f\ v) \end{aligned}$$


 unfoldr  unfold

And we already know that $unfoldL$ is equivalent to $unfoldL'$. So on the next slide, let's reimplement $digits$ and $group$ using $unfoldL'$.



```
> digits = reverse . unfoldr gen
  where gen 0 = Nothing
        gen d = Just (d `mod` 10, d `div` 10)

> digits 2718
[2,7,1,8]
```



```
scala> def digits(n: Int): List[Int] =
  |   LazyList.unfold(n):
  |     case 0 => None
  |     case x => Some(x % 10, x / 10)
  |   .toList.reverse
def digits(n: Int): List[Int]

scala> digits(2718)
val res0: List[Int] = List(2, 7, 1, 8)
```



```
> group n = unfoldr gen
  where gen [] = Nothing
        gen xs = Just (take n xs, drop n xs)

> group 3 [1,2,3,4,5,6,7,8,9,10,11]
[[1,2,3],[4,5,6],[7,8,9],[10,11]]
```



```
scala> def group[A](n: Int, as: List[A]): List[List[A]] =
  |   LazyList.unfold(as):
  |     case Nil => None
  |     case xs => Some(xs.take(n), xs.drop(n))
  |   .toList
def group[A](n: Int, as: List[A]): List[List[A]]

scala> group(3, List(1,2,3,4,5,6,7,8,9,10,11))
val res0: List[List[Int]]=List(List(1,2,3),List(4,5,6),List(7,8,9),List(10,11))
```



unfoldr



unfold

unfoldL' :: $(\beta \rightarrow \text{Maybe}(\alpha, \beta)) \rightarrow \beta \rightarrow \text{List } \alpha$
unfoldL' f u = **case** f u **of**
 Nothing → Nil
 Just (x, v) → *Cons* x (*unfoldL'* f v)



As an aside, **Scala** functions **digits** and **group**, may alternatively be written as follows

```
def digits(n: Int): List[Int] =  
  LazyList.unfold(n):  
    case 0 => None  
    case x => Some(x % 10, x / 10)  
  .toList.reverse
```



```
def digits(n: Int): List[Int] =  
  LazyList.unfold(n): x =>  
    Option.unless(x==0)(x % 10, x / 10)  
  .toList.reverse
```

```
def group[A](n: Int, as: List[A]): List[List[A]] =  
  LazyList.unfold(as):  
    case Nil => None  
    case xs => Some(xs.take(n), xs.drop(n))  
  .toList
```



```
def group[A](n: Int, as: List[A]): List[List[A]] =  
  LazyList.unfold(as): xs =>  
    Option.unless(xs.isEmpty)(xs.take(n), xs.drop(n))  
  .toList
```




Earlier we looked at how to define *iterate* in terms of *unfold*.


Implementing *iterate* in terms of *unfoldL* is very similar, because *unfold* and *unfoldL* are very similar:

$$\begin{aligned} \text{iterate} &:: (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha] \\ \text{iterate} &= \text{unfoldL} (\text{const False}) \text{id} \end{aligned}$$

What about defining *iterate* in terms of *unfoldL'*?

The answer below can be found on the next slide, in the documentation of Haskell's equivalent of *unfoldL'*

$$\begin{aligned} \text{iterate} &:: (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha] \\ \text{iterate } f &= \text{unfoldL}' (\lambda x. \text{Just } (x, f x)) \end{aligned}$$
$$\begin{aligned} \text{unfoldL} &:: (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \\ \text{unfoldL } p \ f \ g \ b &= \text{if } p \ b \\ &\quad \text{then Nil} \\ &\quad \text{else Cons } (f \ b) (\text{unfoldL } p \ f \ g \ (g \ b)) \end{aligned}$$


 unfoldr  unfold

$$\begin{aligned} \text{unfoldL}' &:: (\beta \rightarrow \text{Maybe } (\alpha, \beta)) \rightarrow \beta \rightarrow \text{List } \alpha \\ \text{unfoldL}' \ f \ u &= \text{case } f \ u \ \text{of} \\ &\quad \text{Nothing} \rightarrow \text{Nil} \\ &\quad \text{Just } (x, v) \rightarrow \text{Cons } x \ (\text{unfoldL}' \ f \ v) \end{aligned}$$

base-4.21.0.0: Core data structures and operations

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

Data.List

Unfolding

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

[# Source](#)

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

```
iterate f == unfoldr (\x -> Just (x, f x))
```

```
iterate :: (α → α) → α → [α]
iterate f x = x : iterate f (f x)
```

```
unfoldL' :: (β → Maybe (α, β)) → β → List α
unfoldL' f u = case f u of
    Nothing → Nil
    Just (x, v) → Cons x (unfoldL' f v)
```

 `unfoldr`  `unfold`



Earlier we saw that *unfold* is defined in terms of *map*, *takewhile* and *iterate*.

$$\begin{aligned} \text{unfold} &:: (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \\ \text{unfold } h \, p \, t &= \text{map } h \circ \text{takewhile } p \circ \text{iterate } t \end{aligned}$$

What about *unfoldL*? It looks like this should work, right?

$$\text{unfoldL } p \, f \, g = \text{map } f \circ \text{takeWhile } (\text{not} \circ p) \circ \text{iterate } g$$

Yes, it does. Just like with *unfold* implementing *digits* and *group* using *unfoldL* is much simpler thanks to *unfoldL* capturing the pattern of computation involving *map*, *takewhile* and *iterate*.

```
> unfoldL p f g = map f . takeWhile (not . p) . iterate g
> digits = reverse . unfoldL (==0) (`mod` 10) (`div` 10)
> digits 2718
[2,7,1,8]
```

```
> group n = unfoldL (==[]) (take n) (drop n)
> group 3 [1,2,3,4,5,6,7,8,9,10,11]
[[1,2,3],[4,5,6],[7,8,9],[10,11]]
```


$$\begin{aligned} \text{unfoldL} &:: (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \\ \text{unfoldL } p \, f \, g \, b &= \text{if } p \, b \text{ then Nil} \\ &\quad \text{else Cons } (f \, b) (\text{unfoldL } p \, f \, g \, (g \, b)) \end{aligned}$$



The next slide recaps the different ways of writing a function to **unfold** a list by using *iterate*, *unfold unfoldL*, *unfoldL'*.

The slide after that recaps how *unfoldL* and *unfoldL'* relate to *iterate*.



Here are the **four different implementations** of *digits* that we have considered (we have not actually bothered with the third **Scala** one, due to it being very similar to the second one).

In the **two succinct middle ones**, we are relying on our **own implementation** of *unfold* and *unfoldL*, since as far as I can tell, they are not provided by **Haskell** and **Scala**.

iterate `digits = reverse . map (`mod` 10) . takeWhile (/= 0) . iterate (`div` 10)`

unfold `digits = reverse . unfold (`mod` 10) (/= 0) (`div` 10)`

unfoldL `digits = reverse . unfoldL (== 0) (`mod` 10) (`div` 10)`

unfoldL' `digits = reverse . unfoldr gen
 where gen 0 = Nothing
 gen d = Just (d `mod` 10, d `div` 10)`

iterate `def digits(n: Int): List[Int] = LazyList.iterate(n)(_ / 10).takeWhile(_ != 0).map(_ % 10).toList.reverse`

unfold `def digits(n: Int): List[Int] = unfold[Int,Int](_ % 10, _ != 0, _ / 10, n).toList.reverse`

unfoldL `def digits(n: Int): List[Int] = unfoldL[Int,Int](_ == 0, _ % 10, _ / 10, n).toList.reverse`

unfoldL' `def digits(n: Int): List[Int] = LazyList.unfold(n): x =>
 Option.unless(x==0)(x % 10, x / 10)
 .toList.reverse`

iterate :: $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha]$
iterate $f\ x = x : \text{iterate}\ f\ (f\ x)$



unfold :: $(\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List}\ \alpha$
unfold $h\ p\ t = \text{map}\ h \circ \text{takeWhile}\ p \circ \text{iterate}\ t$



iterate $f = \text{unfoldL}\ (\text{const}\ \text{False})\ \text{id}\ f$

iterate $f = \text{unfoldL}'\ (\lambda x. \text{Just}\ (x, f\ x))$

unfoldL $p\ f\ g = \text{map}\ f \circ \text{takeWhile}\ (\text{not} \circ p) \circ \text{iterate}\ g$

unfoldL :: $(\beta \rightarrow \text{Bool}) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List}\ \alpha$
unfoldL $p\ f\ g\ b = \text{if}\ p\ b$
 then *Nil*
 else *Cons* $(f\ b)\ (\text{unfoldL}\ p\ f\ g\ (g\ b))$



 *unfoldr*  *unfold*

unfoldL' :: $(\beta \rightarrow \text{Maybe}\ (\alpha, \beta)) \rightarrow \beta \rightarrow \text{List}\ \alpha$
unfoldL' $f\ u = \text{case}\ f\ u\ \text{of}$
 Nothing $\rightarrow \text{Nil}$
 Just $(x, v) \rightarrow \text{Cons}\ x\ (\text{unfoldL}'\ f\ v)$



The next two slides are the final ones
and contain some afterthoughts.



The *unfoldL* function also makes an appearance in one of the exercises in [Graham Hutton's](#) book, *Programming in Haskell*.

His example of **unfolding** a number into **binary digits** only differs from our example of **unfolding** a number in to **decimal digits** in that he doesn't need to **reverse** the result of the **unfolding** because he doesn't require the **binary digits** to be ordered from **most significant** to **least significant**.

A **higher-order** function **unfold** that **encapsulates** a **simple pattern** of **recursion** for **producing** a **list** can be defined as follows:

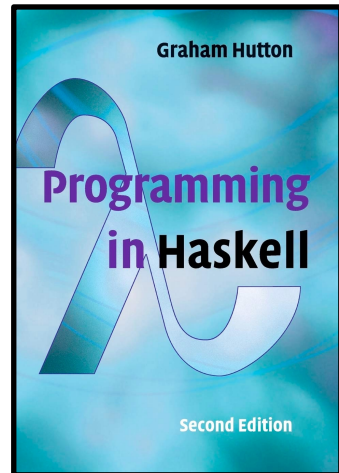
```
unfold p h t x | p x = []
                | otherwise = h x : unfold p h t (t x)
```

That is, the function **unfold p h t** **produces** the **empty list** if the **predicate p** is true of the argument value, and otherwise **produces** a **non-empty list** by applying the function **h** to this value to give the **head**, and the function **t** to **generate** another argument that is **recursively processed** in the same way to **produce** the **tail** of the **list**.

For example, the function **int2bin** can be **rewritten more compactly** using **unfold** as follows:

```
int2bin = unfold (== 0) ('mod' 2) ('div' 2)
```

Redefine the functions **chop8**, **map f** and **iterate f** using **unfold**.



Graham Hutton

X @haskellhutt

```
unfoldL      :: (β → Bool) → (β → α) → (β → β) → β → List α
unfoldL p f g b = if p b
                  then Nil
                  else Cons (f b) (unfoldL p f g (g b))
```

```
> unfoldL p f g = map f . takeWhile (not . p) . iterate g
> digits = reverse . unfoldL (==0) (`mod` 10) (`div` 10)
> digits 2718
[2,7,1,8]
```




While in part 1 of the **The Science of Functional Programming**, **Sergei Winitzki** doesn't mention the **unfold** function provided by **Scala**, this may be because the function was introduced in **Scala 2.13** (in June 2019), i.e. about 1.5 years after the first code commit for the book (in Oct 2017). **Sergei** does discuss the **unfold** function in the book's exercises (see below). Also, in upcoming part 2 of the book I see there is a section called **12.2.8 Using recursion schemes. II. Unfolding operations**.

Example 2.5.1.7 Implement a function **unfold** with the type signature

```
def unfold[A](init: A)(next: A => Option[A]): Stream[A]
```

The function should create a **stream** of values of type **A** with the initial value **init**. Next elements are computed from previous ones via the function **next** until it returns **None**.

...

Exercise 2.5.2.13 Implement a function **unfold2** with the type signature

```
def unfold2[A,B](init: A)(next: A => Option[(A,B)]): Stream[B]
```

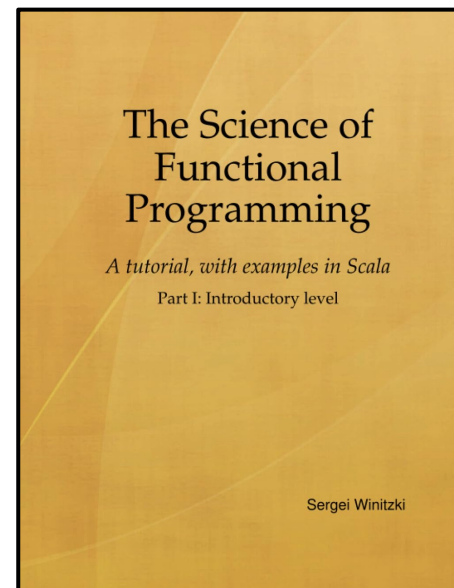
The function should create a **stream** of values of type **B** by repeatedly applying the given function **next** until it returns **None**. At each iteration, **next** should be applied to the value of type **A** returned by the previous call to **next**. An example test:

```
scala> unfold2(0) { x => if (x > 5) None else Some((x + 2, s"had $x")) }
```

```
res0: Stream[String] = Stream(had 0, ?)
```

```
scala> res0.toList
```

```
res1: List[String] = List(had 0, had 2, had 4)
```





That's all for this deck.
I hope you found it useful.

 [X@philip_schwarz](#)