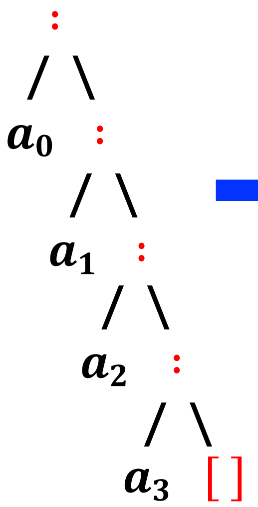
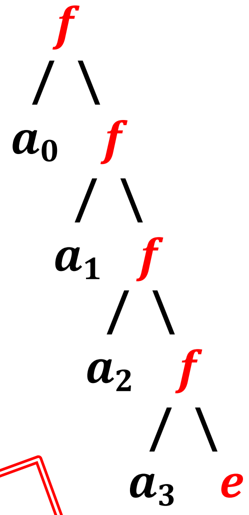


Folding



CHEAT-SHEET

#4

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

We want to write function *decimal*, which given the digits of an integer number

$$[d_0, d_1, \ldots, d_n]$$

computes the integer value of the number

$$\sum_{k=0}^{n} d_k * 10^{n-k}$$

```
haskell> decimal [1,2,3,4]
1234
```

```
scala> decimal(List(1,2,3,4))
val res0: Int = 1234
```

$$d_0 * 10^3 + d_1 * 10^2 + d_2 * 10^1 + d_3 * 10^0 = 1 * 1000 + 2 * 100 + 3 * 10 + 4 * 1 = 1234$$

Thanks to the universal property of fold, if we are able to define *decimal* so that its equations match those on the left hand side of the following equivalence, then we are also able to implement *decimal* using a **right fold**

The universal property of *fold*

$$\begin{aligned} g\ [\,] &= v \\ g\ (x : xs) &= f\ x\ (g\ xs) \end{aligned} \quad \Leftrightarrow \quad g = fold\ f\ v$$

$$\begin{aligned} g &:: [\alpha] \rightarrow \beta \\ v &:: \beta \\ f &:: \alpha \rightarrow \beta \rightarrow \beta \end{aligned}$$

i.e. given *foldr*

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

we can reimplement *decimal* like this:

$$decimal = foldr\ f\ v$$

Notice that f has two parameters: the head of the list, and the result of recursively calling g with the tail of the list

$$g\ (x : xs)\ = f\ x\ (g\ xs)$$

In order to define our $decimal$ function however, the two parameters of f are not sufficient. When $decimal$ is passed $[dk, \ldots, dn]$, f is passed digit d_k, so f needs n and k in order to compute 10^{n-k}, but $n - k$ is the number of elements in $[dk, \ldots, dn]$ minus one, so by nesting the definition of f inside that of $decimal$, we can avoid explicitly adding a third parameter to f :

```haskell
decimal :: [Int] -> Int
decimal [] = 0
decimal (d:ds) = f d (decimal ds) where
  e = length ds
  f :: Int -> Int -> Int
  f d ds = d * (10 ^ e) + ds
```

```scala
def decimal(digits: List[Int]): Int =
  val e = digits.length-1
  def f(d: Int, ds: Int): Int =
    d * Math.pow(10, e).toInt + ds
  digits match
    case Nil => 0
    case d +: ds => f(d, decimal(ds))
```

The unnecessary complexity of the $decimal$ functions on this slide is purely due to them being defined in terms of f. See next slide for simpler refactored versions in which f is inlined.

We nested f inside $decimal$, so that the equations of $decimal$ match (almost) those of g. They don't match perfectly, in that the f nested inside $decimal$ depends on $decimal$'s list parameter, whereas the f nested inside g does not depend on g's list parameter. Are we still able to redefine $decimal$ using $foldr$? If the match had been perfect, we would be able to define $decimal\ = foldr\ f\ 0$ (with $v = 0$), but because f needs to know the value of $n - k$, we can't just pass f to $foldr$, and use 0 as the initial accumulator. Instead, we need to use $(0, 0)$ as the accumulator (the second 0 being the initial value of $n - k$, when $k = n$), and pass to $foldr$ a helper function h that manages $n - k$ and that wraps f, so that the latter has access to $n - k$.

```haskell
h :: Int -> (Int,Int) -> (Int,Int)
h d (ds, e) = (f d ds, e + 1) where
  f :: Int -> Int -> Int
  f d ds = d * (10 ^ e) + ds

decimal :: [Int] -> Int
decimal ds = fst (foldr h (0,0) ds)
```

```scala
def h(d: Int, acc: (Int,Int)): (Int,Int) = acc match { case (ds, e) =>
  def f(d: Int, ds: Int): Int =
    d * Math.pow(10, e).toInt + ds
  (f(d, ds), e + 1)
}
def decimal(ds: List[Int]): Int =
  ds.foldRight((0,0))(h).head
```

Same *decimal* functions as on the previous slide, but refactored as follows:

1. inlined ***f*** in all four functions
2. inlined **e** in the first two functions
3. renamed ***h*** to ***f*** in the last two functions

```haskell
decimal :: [Int] -> Int
decimal [] = 0
decimal (d:ds) = d*(10^(length ds))+(decimal ds)
```

```scala
def decimal(digits: List[Int]): Int = digits match
  case Nil => 0
  case d +: ds => d * Math.pow(10, ds.length).toInt + decimal(ds)
```

```haskell
f :: Int -> (Int,Int) -> (Int,Int)
f d (ds, e) = (d * (10 ^ e) + ds, e + 1)

decimal :: [Int] -> Int
decimal ds = fst (foldr f (0,0) ds)
```

```scala
def f(d: Int, acc: (Int,Int)): (Int,Int) = acc match
  case (ds, e) => (d * Math.pow(10, e).toInt + ds, e + 1)

def decimal(ds: List[Int]): Int =
  ds.foldRight((0,0))(f).head
```

The definition of *decimal* using a **right fold** is inefficient because it computes $\sum_{k=0}^{n} d_k * 10^{n-k}$ by computing 10^{n-k} for each k.

Not every function on lists can be defined as an instance of $foldr$. ... Even for those that can, an alternative definition may be more efficient. To illustrate, suppose we want a function *decimal* that takes a list of digits and returns the corresponding decimal number; thus

$$decimal\ [x_0, x_1, \ldots, x_n] = \sum_{k=0}^{n} x_k 10^{(n-k)}$$

It is assumed that the most significant digit comes first in the list. One way to compute *decimal* efficiently is by a process of multiplying each digit by ten and adding in the following digit. For example

$$decimal\ [x_0, x_1, x_2] = 10 \times (10 \times (10 \times 0 + x_0) + x_1) + x_2$$

This decomposition of a sum of powers is known as ***Horner's* rule**.

Suppose we define $\oplus$ by $n \oplus x = 10 \times n + x$. Then we can rephrase the above equation as

$$decimal\ [x_0, x_1, x_2] = ((0 \oplus x_0) \oplus x_1) \oplus x_2$$

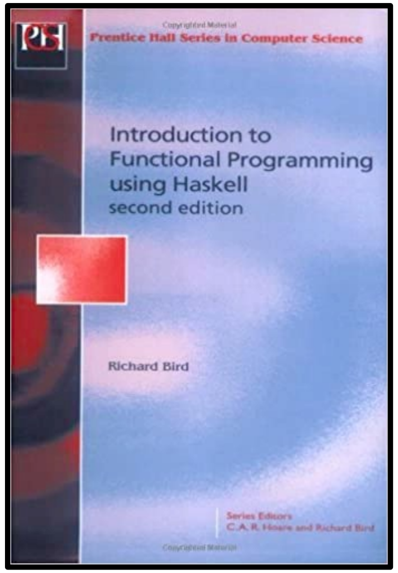
This is almost like an instance of $foldr$, except that the grouping is the other way round, and the starting value appears on the left, not on the right. In fact the computation is dual: instead of processing from right to left, the computation processes from left to right.

This example motivates the introduction of a second fold operator called $foldl$ (pronounced 'fold left'). Informally:

$$foldl\ (\oplus)\ e\ [x_0, x_1, \ldots, xn\ \_\ 1] = (\ldots((e \oplus x_0) \oplus x_1) \ldots) \oplus x_{n\ \_\ 1}$$

The parentheses group from the left, which is the reason for the name. The full definition of $foldl$ is

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



Richard Bird

If we look back at our initial recursive definition of *decimal*, we see that it splits its list parameter into a **head** and a **tail**.

```haskell
decimal :: [Int] -> Int
decimal [] = 0
decimal (d:ds) = d*(10^(length ds)) + (decimal ds)
```

```scala
def decimal(digits: List[Int]): Int = digits match
  case Nil => 0
  case d +: ds => d * Math.pow(10, ds.length).toInt + decimal(ds)
```

If we get *decimal* to split the list into **init** and **last**, we can make it more efficient by using **Horner's rule**:

```haskell
(⊕) :: Int -> Int -> Int
n ⊕ d = 10 * n + d
```

```scala
extension (n: Int)
  def ⊕(d Int): Int = 10 * n + d
```

```haskell
decimal :: [Int] -> Int
decimal [] = 0
decimal ds = (decimal (init ds)) ⊕ (last ds)
```

```scala
def decimal(digits: List[Int]): Int = digits match
  case Nil => 0
  case ds :+ d => decimal(ds) ⊕ d
```

We can then improve on that by going back to splitting the list into a **head** and a **tail**, and making *decimal* tail recursive:

```haskell
decimal :: [Int] -> Int -> Int
decimal [] acc = acc
decimal (d:ds) acc = decimal ds (acc ⊕d)
```

```scala
def decimal(ds: List[Int], acc: Int=0): Int = digits match
  case Nil => acc
  case d +: ds => decimal(ds, acc ⊕ d)
```

And finally, we can improve on that by defining *decimal* using a **left fold**:

```haskell
decimal :: [Int] -> Int
decimal = foldl (⊕) 0
```

```scala
def decimal(ds: List[Int]): Int =
  ds.foldLeft(0)(_⊕_)
```

Recap

In the case of the ***decimal*** function, defining it using a **left fold** is simple and mathematically more efficient

$$decimal\ [1,2,3,4] = 10 * (10 * (10 * (10 * 0 + d_0) + d_1) + d_2) + d_3 = 10 * (10 * (10 * (10 * 0 + 1) + 2) + 3) + 4 = 1234$$

```haskell
decimal :: [Int] -> Int
decimal = foldl (⊕) 0

(⊕) :: Int -> Int -> Int
n ⊕ d = 10 * n + d
```

```scala
def decimal(ds: List[Int]): Int =
  ds.foldLeft(0)(_⊕_)

extension (n: Int)
  def ⊕(d Int): Int = 10 * n + d
```

whereas defining it using a **right fold** is more complex and mathematically less efficient

$$decimal\ [1,2,3,4] = d_0 * 10^3 + (d_1 * 10^2 + (d_2 * 10^1 + (d_3 * 10^0 + 0))) = 1 * 1000 + (2 * 100 + (3 * 10 + (4 * 1 + 0))) = 1234$$

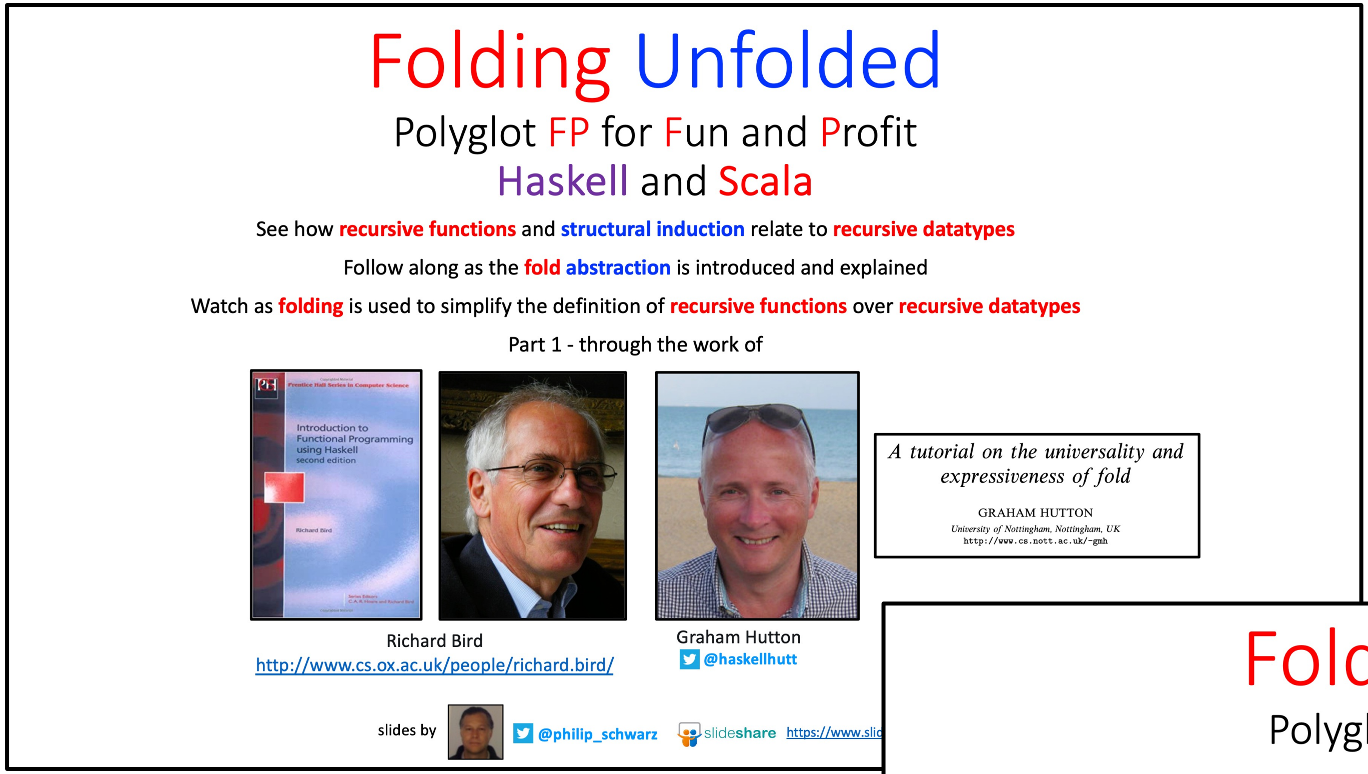
```haskell
decimal :: [Int] -> Int
decimal ds = fst (foldr f (0,0) ds)

f :: Int -> (Int,Int) -> (Int,Int)
f d (ds, e) = (d * (10 ^ e) + ds, e + 1)
```

```scala
def decimal(ds: List[Int]): Int =
  ds.foldRight((0,0))(f).head

def f(d: Int, acc: (Int,Int)): (Int,Int) = acc match
  case (ds, e) => (d * Math.pow(10, e).toInt + ds, e + 1)
```

inspired by



https://fpilluminated.com/

FP Illuminated