chapter 10 : Folding Lists

foldr : f x (foldr f xs)

*) Folding is a concept that in usefulness and importance beyond lists, but lists are often how they are introduced.

↓

folds are a general concept are called catamorphisms. Means of deconstructing data. If the spine of a list is the structure of a list, then a fold is what can reduce that structure.

→ Bringing you into the fold : foldr : "fold right". This is the fold we are going to use the most with lists.

```
foldr :: (a → b → b) → b → [a] → b   (-- GHC 7.8 & olders)

foldr :: Foldable t
      ⇒ (a → b → b) → b → t a → b

foldr :: (a → b → b) → b → [] a → b
```

*) GHC 7.10 abstracted out the list-specific part of folding into a typeclass that lets us reuse the same folding functions for any datatype that can be folded, not just lists.

*) A parallel between map & foldr

```
foldr :: (a → b → b) → b → [a] → b
          (two args)                  (indicating recursive form of redn.)

map :: (a → b) → [a] → [b]
```

map (+1) 1 : 2 : 3 : []

(+1) 1 : (+1) 2 : (+1) 3 : []

foldr (+) 0 (1 : 2 : 3 : [])

1 + 2 + 3 + 0

*) where a map applies the given fn. to each member of a list, and returns a list, a fold replaces the cons constructors with the function & reduces the list.

→ Recursive Patterns

>> sum [1, 5, 10] → This has the effect of both mapping an operator over a list and also reducing the list.

16

```
>> sum :: [Integer] → Integer
   sum [] = 0
   sum (x:xs) = x + sum xs

>> length :: [a] → Integer
   length [] = 0
   length (x:xs) = 1 + length xs

>> product :: [Integer] → Integer
   product [] = 1
   product (x:xs) = x * product xs
```

In each case the base case is the identity for that function.

↓

Also each of them has a main def. with a recursive pattern that associates to the right.

```
>> concat :: [[a]] → [a]
   concat [ ] = [ ]
   concat (x : xs) = x ++ concat xs
```

↓
Head gets evaluated, set aside, then we move to the right & so on.

→ fold right

foldr : the "right fold" because the fold is right associative.

```
foldr :: (a → b → b) → b → [a] → b
foldr f z [ ] = z
foldr f z (x : xs) = f x (foldr f z xs)
```

second argument

first argument

the redn. function

evaluation happens on the basis of structure.

null case answer
↓
varies according to application

Example : foldr (+) 0 [1, 2, 3]

6

↓

2nd case : (+) 1 (foldr (+) 0 [2, 3])

5

foldr (+) 0 [2, 3]

↓

2nd case : (+) 2 (foldr (+) 0 [3])

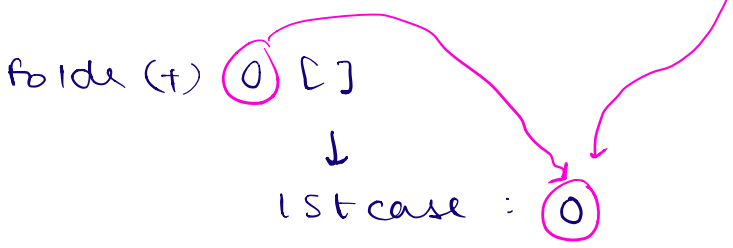
3 (after f applied)

foldr (+) 0 [3]

↓

2nd case : (+) 3 (foldr (+) 0 [ ])

0



* : this gets expanded only because (+) is strict in both its arguments.

x) One nonobvious aspect of folding is that it happens in two stages, traversal & folding. Traversal is the stage in which the fold recurses over the spine & folding refers to the evaluation or redn. of the folding fnc applied to the values.

↓

All folds recurse over the spine in the same direction, the folding direction is what is different.

↓

With foldr, as indicated before, one fold is an argument to the fn. we're folding with.

x) Given the two-stage process and nonstrict evaluation, if f doesn't evaluate its second argument (i.e. the second argument is not needed to get the answer), no more of the spine will be forced.

↓

One of the consequences of this is that the fold can avoid evaluate not only some of the values but some or all of the list's spine as well. For that reason

foldr can be used with ∞ lists as well. for example

```
myAny :: (a → Bool) → [a] → Bool
myAny f xs =
    foldr (\x b → f x || b) False xs
```

(xs → list; f → function; False → null case)

The first argument will be the first element of the list & the next will be the result of the rem. fold. Since || is non strict, hence we check if x satisfies the given predicate, and if it does, then we don't evaluate the remaining fold.

*) Another term we use for never-ending evaluation is known as bottom or undefined.

```
>> let xs = [1,2] ++ undefined
   length $ take 2 $ take 4 xs
```

It doesn't matter that take 4 could have hit bottom, nothing forced it to because of the take 2 between it & length.

*) Analysing the first argument now.

```
foldr :: (a → b → b) → b → [a] → b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

(z → represents the building base/ foundation; x)

↓

involves a pattern match that is strict by default i.e. the f only applies to x if there is an x value and not just an empty list.

↓

This means that foldr must force an initial cons cell in order to discriminate b/w the [] and the (x:xs) cases, so the first cons cell cannot be undefined, or in other words list cannot be undefined. [undefined] even works, but only undefined as a list argument doesn't work, because then it has no way of deconstructing. [undefined] can be deconstructed to undefined & [], and then if undefined argument is not required, then no error will be thrown around. But no matter, a spine list needs to be sent as argument anyhow.

↓

foldr (_ _ → 9001) 0 undefined → gives an error

foldr (_ _ → 9001) 0 [undefined] → doesn't give an error

↙

Here the cons cells contain bottom values but are not themselves bottom.

Traversing the rest of the spine doesn't happen unless the fn. asks for results of having folded the rest of the list.

*) const :: a → b → a

const x _ = x

>> foldr const 0 ([1,2] ++ undefined)

⊥

>> foldr const 0 ([undefined, 2])

error

→ foldl : folding process is left associative and proceeds in the opposite direction as that of foldr.

foldl :: (b → a → b) → b → [a] → b

foldl f acc [] = acc → represents accumulated value

foldl f acc (x:xs) = foldl f (f acc x) xs

*) example : foldl (+) 0 (1:2:3:[])

↓

foldl (+) ((+) 0 1) (2:3:[])

foldl (+) ((+) 0 1) (2:3:[])

f acc x

↓

foldl (+) (1) (2:3:[])

↓

foldl (+) ((+) 1 2) (3:[])

fold (+) 3 (3:[])

↓

foldl (+) ((+) 3 3) []

↓

↓

foldl (+) (6) [] = 6

*) Its evident that foldl begins its reduction process by adding the accumulator value to the head of the list, whereas foldr had added it to the final value of the list.

*) scans functions can be used to see how folds evaluate. Scans are similar to folds but return a list of all the immediate stages of the fold.

>> foldr (+) 0 [1..5]

15

>> scanr (+) (0) [1..5]

15, 10, 6, 3, 2, 1, (0) → starting stage

>> foldl (+) 0 [1..5]

15

>> scanl (+) (0) [1..5]

0, 1, 3, 6, 10, 15

starting stage

*) last (scanl f z xs) = foldl f z xs

*) head (scanr f acc xs) = foldr f acc xs

*) Associativity and folding: The fundamental way to think of evaluation in Haskell is through substitution.

↓

When we use a right fold on a list with the func. f and start value z, we're in a sense, replacing the

cons constructors with our folding fn. and the empty list constructor with our start value z.

↓

furthermore, lazy evaluation dictates what older things get evaluated in.

```
>> foldr f z [1, 2, 3]
⇒ 1 `f` (foldr f z [2, 3])
⇒ 1 `f` (2 `f` (foldr f z [3]))
⇒ 1 `f` (2 `f` (3 `f` (foldr f z [])))
                        ↓
                        z
⇒ 1 `f` (2 `f` (3 `f` z))
```

↓

gets evaluated first, since innermost paranthesis

↓

Right folds have to traverse the list outside-in, but the folding itself starts from the end of the list.

```
>> foldr (^) 2 [1..3]
```

```
foldr f z xs =
f x (foldl f z xs)
```

① ↓

```
(^) (1) (foldr (^) 2 [2, 3])
                ↓
(^) (2) (foldr (^) 2 [3])
                ↓
(^) (3) (foldr (^) 2 [])
                ↓
                2
```

512

9

(1 ^ (2 ^ (3 ^ 2)))

```
>> foldl (^) 2 [1..3]

      foldl (^) ((^) 2 1) [2,3]
                  ↓
          foldl (^) (2) [2,3]
                  ↓
(((1^2)^2)^3)   foldl (^) ((^) 2 2) [3]
                  ↓
              foldl (^) (4) [3]
                  ↓
              foldl (^) ((^) 4 3) []
              foldl (^) (64) []
```

```
(*) >> foldr (:) [] [1..3] → list to operate on
             |    → 2
          function

      ↓                         1 : 2 : 3 : []
   : 1 (foldr (:) [] [2,3])
            ↓                   2 : 3 : []
        : 2 (foldr (:) [] [3])
                 ↓              3 : []
              : 3 (foldr (:) [] [])
                          ↓
                          []
```

```
>> foldl (flip (:)) [] [1..3]
  => foldl (flip (:)) (flip (:) [] 1) [2,3]
  => foldl (flip (:)) ((:) 1 []) [2,3]
```

⇒ foldl (flip (:)) (1 : [ ]) [2, 3]

↓

⇒ foldl (flip (:)) (flip (:) (1 : [ ]) 2) [3]

⇒ foldl (flip (:)) ((:) 2 (1 : [ ])) [3]

⇒ foldl (flip (:)) (2 : 1 : [ ]) [3]

↓

⇒ foldl (flip (:)) (flip (:) (2 : 1 : [ ]) 3) [ ]

⇒ foldl (flip (:)) (3 : 2 : 1 : [ ]) [ ]

↓

3 : 2 : 1 : [ ] → [3, 2, 1]

←

*) foldr const 0 [1..5]

(const 1 _) ⟶ accumulates

1

*) foldr (flip const) 0 [1..5] → 0

↓

⇒ (flip const) 1 (foldr (flip const) 0 [2..5])

⇒ const (foldr (flip const) 0 [2..5]) (1)

↓

⇒ (flip const) (2) (foldr (flip const) 0 [3..5])

⇒ const (foldr (flip const) 0 [3..5]) (2)

↓

⇒ (flip const) (3) (foldr (flip const) 0 [4,5])

⇒ const (foldr (flip const) 0 [4,5]) (3)

↓

⇒ (flip const) (4) (foldr (flip const) 0 5)

⇒ const (foldr (flip const) 0 [5]) (4)

↓

⇒ (flip const) (5) (foldr (flip const) 0 [ ])

⇒ const (foldr (flip const) 0 [ ]) (5)

↓

0

*) foldl const 0 [1..5]

⇒ foldl const (const 0 1) [2..5]

⇒ foldl const (0) [2..5] → (0)

*) foldl (flip const) 0 [1..5]

⇒ foldl (flip const) (flip const 0 1) [2..5]

⇒ foldl (flip const) (1) [2..5]

⇒ foldl (flip const) (flip const 1 2) [3..5]

⇒ foldl (flip const) (2) [3..5] → (5)

*) foldl ((++), show) "" [1..5]

⇒ foldl ((++).show) ((++).show "" 1) [2..5]

→ Unconditional spine recursion : The successive recursion steps on the spine are not intermediated in foldl as they are done in foldl, i.e. recursion of the spine is unconditional.

↓

Having a fn. that doesn't force evaluation won't change anything.

|

for example: let xs = [1..5] ++ undefined

>> foldr const 0 xs → because here the first arg. got established & hence next arg. did not have to be evaluated.

1

>> foldr (flip const) 0 xs

error

↘ here the first argument is being evaluated through the recursive spine & will hit bottom, hence the error.

>> foldl const 0 xs

error

↘ here the evaluation of both arguments is unconditional & the fn. operates in ace, and the accumulator value keeps getting modified as we move deeper in the recursive tree.

>> foldl (flip const) 0 xs

error

↘ will give an error for the evaluation of the first argument itself as had happened for foldr

*) This evaluation is strict & unconditional over spine & not values. Hence selective evaluation of list values is allowed.

>> let xs = [1, 2, 3, 4, 5, undefined]

>> foldl (_ _ → 5) 0 xs

5

The arguements should force evaluation for the values to matter.

*) foldl is generally inappropriate with lists that are or could be ∞, but the combination of the forced spine evaluation with nonstrictness means that it is usually inappropriate even for long lists, as the forced evaluation of the spine affects performance negatively.

↓

Because foldl must evaluate its whole spine before it starts evaluating values in each cell, it piles up a lot of unevaluated values as it traverses the spine

↓

In situations where foldl is required, we can choose to use foldl'. It works the same except it is strict i.e. it forces evaluation of values inside cons cells. This has a less negative effect on performance over long lists.

→ How to write fold functions : factors while writing folds :

i) working out the starting value / identity for the function. (z or acc). This value happens to be our fall back, incase we encounter a empty list.

ii) next are our arguments:

a) one of the elements of the list

b) start value or the accumulated value (as the list is being processed)

*) A fn. to take 3 letters of each string in a list of strings and concatenate that result into a final string.

```
>> foldl (\b a -> take 3 a) "" pab

>> foldl f (f "" "Pizza") "" ab
          ↓
   foldl f "Piz" ab
          ↓
>> foldl f (f "Piz" "Apple") b.

   foldl f "App" b
>> foldl f (f "App" Banana) ""
   foldl f "Ban" "" -> "Ban"
```

```
foldl (\b a -> b ++ take 3 a) "" pab.
foldl (\a b -> take 3 a ++ b) "" pab
```

→ Exercise : Database processing

1. filterDbDate :: [DatabaseItem]
   → [UTCTime]

```
filterDbDate [] = []
filterDbDate (DbDate x) : xs) = x : filterDbDate xs
filterDbDate (_ : xs) = filterDbDate xs
```

3. mostRecent :: [DatabaseItem] → UTCTime

```
mostRecent x = minimum . filterDbDate $ x
```

4. sumDb :: [DatabaseItem] → Integer

```
sumDb x = sum . filterDbNumber $ x
```

→ folding and evaluation :

*) associativity differentiates foldl & foldr

*) The relationship between foldr & foldl

```
foldr f z xs = foldl (flip f) z (reverse xs)
```

↓

only for finite list

```
>> : 1 [2]
[1,2]
```

*) example :

```
>> let xs = [1..5]
>> foldr (:) [] xs
[1,2,3,4,5]
>> foldl (flip (:)) [] xs → reverse $ foldl (flip (:)) [] xs
[5,4,3,2,1]
>> foldl (flip (:)) [] (reverse xs)
[1,2,3,4,5]
```

*) foldl nearly useless and should almost always be replaced with foldl' for reasons explain later.

→ Scans : like folds accumulate values, and like maps, returns a list of values. The list of values corresponds to the results of the intermediate stages of evaluation.

```
foldr :: (a → b → b) → b → [a] → b
scanr :: (a → b → b) → b → [a] → [b]
```

→ main diff being the result is a list of values.

↓

```
foldl :: (b → a → b) → b → [a] → b
scanl :: (b → a → b) → b → [a] → [b]
```

*) scanl (+) 1 [1..3]

→ →

= [1, 1+1,
(1+1)+2,
((1+1)+2)+3]

→ unfolding (weak normal form)

= [1, 2, 4, 7] → normal form

*) scanl :: (a → b → a) → a → [b] → [a]

scanl f q ls =
q : (case ls of
[] → [],
x:xs → scanl f (f q x) xs)

→ older base library version

*) fibs = 1 : scanl (+) 1 fibs → gives an ∞ list

fibs N x = fibs !! x → extract the 'x' th term from the fibonacci list

fibsf x = take x fibs

fibss x = takeWhile (<100) fibs

1, 2, 6

1a) 3 tuples of all possible stop-vowel-stop combinations

stops = "pbtdkg"

vowels = "aeiou"

list comprehensions.

```
[ (x, y, z) | x ← stops, y ← vowels, z ← stops]
```

b) combinations beginning with 'p'

```
[ ('p', x, y) | x ← vowels, y ← stops]
```

2.
```
seekritfunc x = div (sum (map length (words x)))
                    (length (words x))
```

words function is applied to x to get a list a words, over which length function is being mapped, after which their sum is being taken, and that is being divided total no. of words.

↓

more on the likes of avg no. of letters per word

```
seekritfunc :: String → Int
```

3.
```
seekritfunc x = n / d
  where n = fromIntegral (sum $ map length $ words x)
        d = fromIntegral (length . words $ x)
```

1.
```
myOr :: [Bool] → Bool
myOr [] = False
myOr (x : xs) = x || myOr xs
```
→ Recursion

```
myOr x = foldr (||) False x
```
→ using foldr

```
foldr f z (x : xs)
```

2. myAny :: (a → Bool) → [a] → Bool

myAny f x = foldr (\x y → f x || y) False x

f x (foldr f z xs)

3. myElem :: Eq a ⇒ a → [a] → Bool

myElem a xs = foldr chk False xs

where chk = check a

(2nd part same as last chapter)

check :: a → a → Bool → Bool

check a x y = (x == a) || y

result from entire list

(x) first element

(y) result

4. myReverse :: [a] → [a]

myReverse x = foldr (\x y → y ++ [x]) [] x

5. myMap :: (a → b) → [a] → [b]

myMap f x = foldr (\x y → [f x] ++ y] [] x

6. myFilter :: (a → Bool) → [a] → [a]

myFilter f x = foldr (\x y → if f x then [x] ++ y

else y) [] x

7. squish :: [[a]] → [a]

squish x = foldr (\x y → x ++ y) [] x

8. squishMap :: (a → [b]) → [a] → [b]

maps a fn. over a list & concatenates the results.

squishMap f x = foldr (\x y → f x ++ y) [] x

9. same as last chapter, since the fn. defn is same.

10. myMaximumBy :: (a → a → Ordering) → [a] → a

myMaximumBy ord x = foldr1 (\x y → if ord x y

== GT

then x else y)

x

```
-- myMinimumBy :: (a -> a -> Ordering)
myMinimumBy ord x = foldl1 (\x y -> if ord x y == LT
                                    then x else y) x
```

takes the first element as the base, so ideal in these cases where the base case changes with type