Higher Order Functions, Continued

Folds

Often we wish to traverse a list once, element by element, building up an output as we go.

- Add each number in a list to get the sum of all the elements;
- Multiply each number in a list to get the product of all the elements;
- Increment a number by 1 for each element to get the list’s length;
- Turn a list of strings into an acronym;
- Map a function over a list element by element;
- etc…

```haskell
mySum :: [Int] -> Int
mySum list = case list of
  [] -> 0
  x:xs -> x + mySum xs
```

```haskell
myProd :: [Int] -> Int
myProd list = case list of
  [] -> 1
  x:xs -> x * myProd xs
```

```haskell
myLen :: [a] -> Int
myLen list = case list of
  [] -> 0
  x:xs -> 1 + myLen xs
```

```haskell
acro :: [String] -> String
acro list = case list of
  [] -> ""
  x:xs -> head x : acro xs
```

How are these definitions **different**? How are they the **same**?

```haskell
mySum :: [Int] -> Int
mySum list = case list of
  [] -> 0
  x:xs -> x + mySum xs
```

```haskell
myProd :: [Int] -> Int
myProd list = case list of
  [] -> 1
  x:xs -> x * myProd xs
```

```haskell
myLen :: [a] -> Int
myLen list = case list of
  [] -> 0
  x:xs -> 1 + myLen xs
```

```haskell
acro :: [String] -> String
acro list = case list of
  [] -> ""
  x:xs -> head x : acro xs
```

Everything else is the same!

Fold Right

This suggests that we could define a polymorphic higher order function that takes as input

- a base case;
- a recipe for combining the head with a recursive call on the tail

and then does all of the rest of the work for us!

No more time-consuming error-prone recursions to write

  - Except for all the ones that don’t follow the format of the previous slide

foldr

foldr :: (a -> b -> b) -> b -> [a] -> b

base Prelude, base Data.List

foldr, applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

```
> foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)
```

foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Combining operation

Base case of type b

Input list of inputs of type a

Result of type b

`foldr` in action

```
foldr :: (a -> b -> b) -> b -> [a] -> b

foldr :: (Int -> Int -> Int) -> Int -> [Int] -> Int

mySum = foldr (+) 0

myProd = foldr (*) 1
```

`foldr` in action

```
foldr :: (a -> b -> b) -> b -> [a] -> b

foldr :: (a -> Int -> Int) -> Int -> [a] -> Int

myLen = foldr (\x y -> y + 1) 0
```

Or, to avoid a warning:

```
myLen = foldr (\_ y -> y + 1) 0
```

`foldr` in action

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr :: (String -> String -> String) -> String -> [String] -> String
```

```
acro = foldr (\x y -> head x : y) ""
```

How might you write a ‘safe’ version of `acro` that ignores empty strings instead of crashing on them?

`foldr` in action

`foldr` is just another function, and can be used as a helper anywhere:

```
myMaximum :: [Int] -> Int
myMaximum list = case list of
  [] -> error "No maximum of an empty list"
  x:xs -> foldr max x xs
```

Take some time to understand all the types here!

Defining `foldr`

```
myFoldr :: (a -> b -> b) -> b -> [a] -> b
myFoldr combine base list = case list of
  [] -> base
  x:xs -> combine x (myFoldr combine base xs)
```

Why is it fold **right**?

```
foldr (+) 0 [1,2,3]
= 1 + foldr (+) 0 [2,3]
= 1 + (2 + foldr (+) 0 [3])
= 1 + (2 + (3 + foldr (+) 0 []))
= 1 + (2 + (3 + 0))
```

So the combining operation associates to the **right**.

- i.e. start with the base case, combine it with the rightmost element of list, then continue until we reach the leftmost element of the list.

`foldr` and the structure of lists

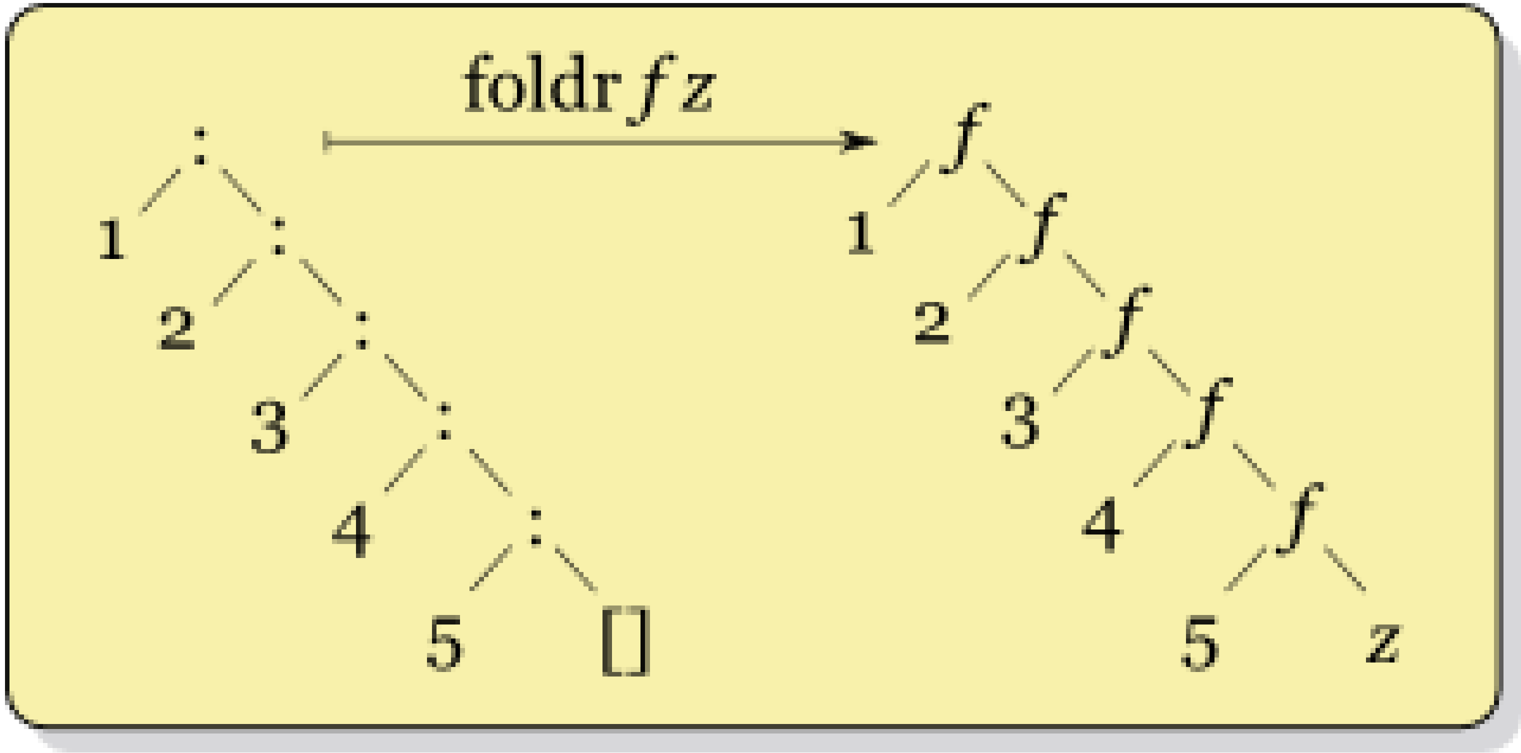
A list `[1,2,3]`

is really `1 : (2 : (3 : []))`

`foldr` replaces `[]` with a base case and `:` with a combining function

e.g. `1 + (2 + (3 + 0))`

So folding right is very natural because lists themselves associate right


foldr f z
:
1
2
3
4
5
[]
f
z

Left folds

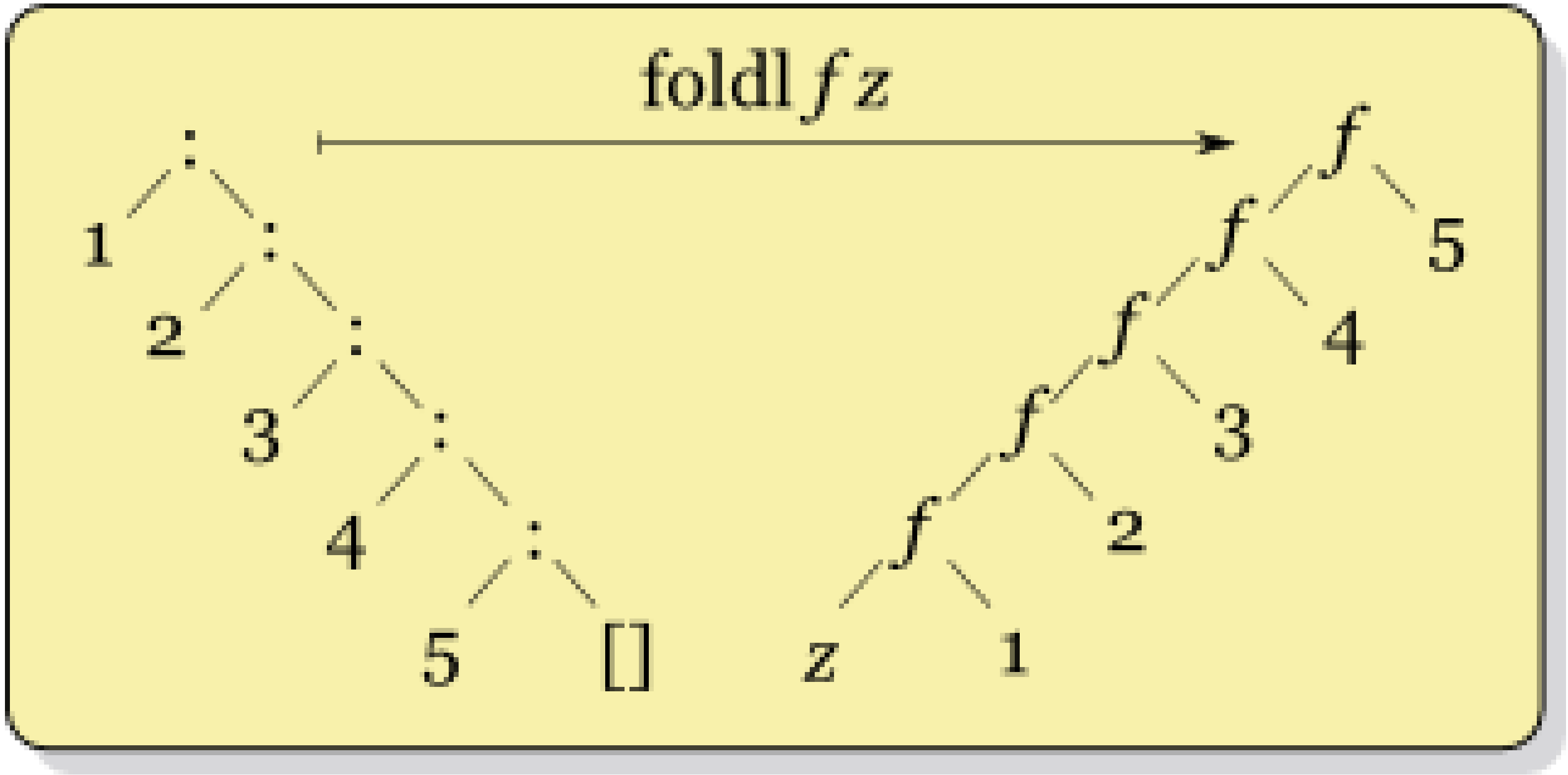
```
mySuml :: [Int] -> Int
mySuml = mySumAcc 0
  where
    mySumAcc acc list = case list of
      [] -> acc
      x:xs -> mySumAcc (acc + x) xs
```

In the above, `mySumAcc` has type `Int -> [Int] -> Int`

Left folds

```
mySuml [1,2,3]
= mySumAcc 0 [1,2,3]
= mySumAcc (0 + 1) [2,3]
= mySumAcc ((0 + 1) + 2) [3]
= mySumAcc (((0 + 1) + 2) + 3) []
= (((0 + 1) + 2) + 3)
```

No difference for +, but not all combining operations are associative!


foldl f z
:
1
2
3
4
5
[]
f
z

foldl

foldl :: (b -> a -> b) -> b -> [a] -> b

base Prelude, base Data.List

foldl, applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

```
> foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
```

The list must be finite.

It folds the list up from the left side

Defining `foldl`

```
myFoldl :: (b -> a -> b) -> b -> [a] -> b
myFoldl combine acc list = case list of
  [] -> acc
  x:xs -> myFoldl combine (combine acc x) xs
```

Compare to the final line of the **right** fold:

```
  x:xs -> combine x (myFoldr combine base xs)
```

`foldl`, folding left

```
foldl (+) 0 [1,2,3]
= foldl (+) (0 + 1) [2,3]
= foldl (+) ((0 + 1) + 2) [3]
= foldl (+) (((0 + 1) + 2) + 3) []
= (((0 + 1) + 2) + 3)
```

So the combining operation associates to the **left**.

- i.e. start with the accumulator, combine it with the leftmost element of list, then continue until we reach the empty list and return the accumulator.

`foldr` versus `foldl`

An example where they give different answers:

```
> foldr (-) 0 [1,2,3]
2

> foldl (-) 0 [1,2,3]
-6
```

Because $((0 - 1) - 2) - 3 \neq 1 - (2 - (3 - 0))$

`foldr` versus `foldl`

More generally, if we think of lists as built up from the empty list by using cons repeatedly, then ***lists are constructed from the right***.

Therefore `foldr` tends to follow the way lists are constructed.

e.g. `foldr (:) [] :: [a] -> [a]` is the identity!

`foldl` goes in the reverse direction from the list's construction

- What happens if you use `(:)` with `foldl`?