Discussion Section: Folds

2021/11/05

Higher-Order Functions

- Takes one or more functions as an argument, or
- Returns a function
- Examples:

```
map                    foldr
filter                 foldl
```

Higher-Order Functions

```
map :: (a → b) → [a] → [b]
```

- *Map*s each element to a new value
- Polymorphic: maps a list of *a*s to a list of *b*s.
- Returns a list of the same `length`.

Quiz

```
map (\x → x `mod` 10) [1, 2, 100, 85]
```

A. [1,2,100,85]
B. [0,0,10,8]
C. [1,2,0,5]
D. [5,0,2,1]
E. Type Error

Quiz

```
map (\x → x `mod` 10) [1, 2, 100, 85]
```

A. [1,2,100,85]
B. [0,0,10,8]
C. [1,2,0,5]
D. [5,0,2,1]
E. Type Error

Higher-Order Functions

```
filter :: (a → Bool) → [a] → [a]
```

- *Filter*s the elements of the list.
- Doesn't change the elements, but
- May return a list of a different length

Quiz

```
filter (not . even) [1,2,3,4,5,6]
```

A. `[1,2,3,4,5,6]`
B. `[2,4,6]`
C. `[1,3,5]`
D. `[6,4,2]`
E. None of the above

Quiz

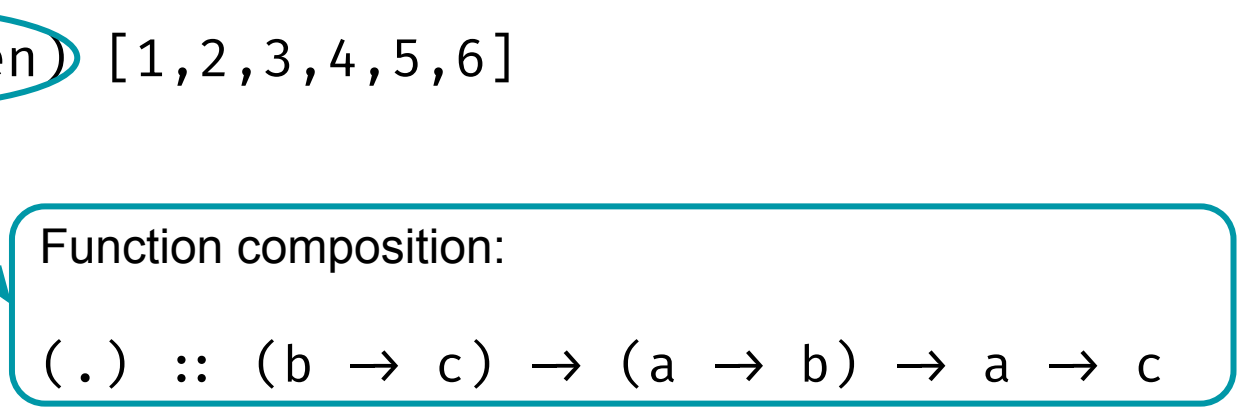
```
filter (not . even) [1,2,3,4,5,6]
```

A. `[1,2,3,4,5,6]`
B. `[2,4,6]`
C. `[1,3,5]`
D. `[6,4,2]`
E. None of the above

Quiz

```
filter (not . even) [1,2,3,4,5,6]
```

A. `[1,2,3,4,5,6]`
B. `[2,4,6]`
C. `[1,3,5]`
D. `[6,4,2]`
E. None of the above

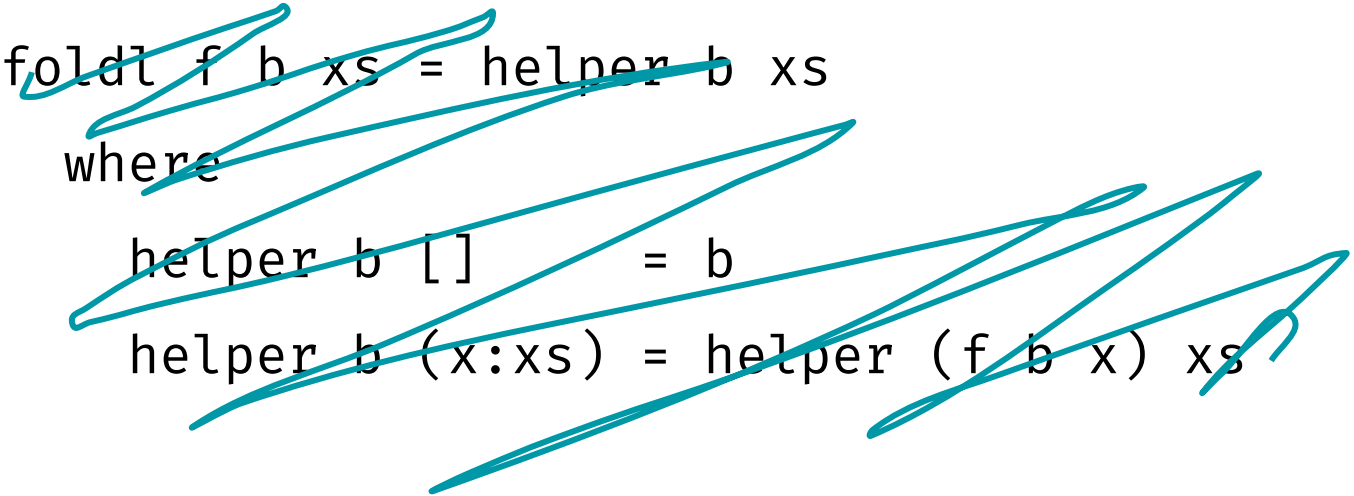


Higher-Order Functions

```
foldl :: (b → a → b) → b → [a] → b
foldl f b xs = helper b xs
  where
    helper b []     = b
    helper b (x:xs) = helper (f b x) xs
```

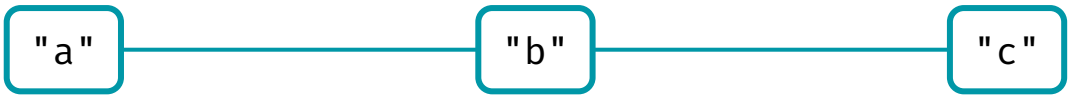
Higher-Order Functions

```
foldl :: (b → a → b) → b → [a] → b
foldl f b xs = helper b xs
  where
    helper b []     = b
    helper b (x:xs) = helper (f b x) xs
```



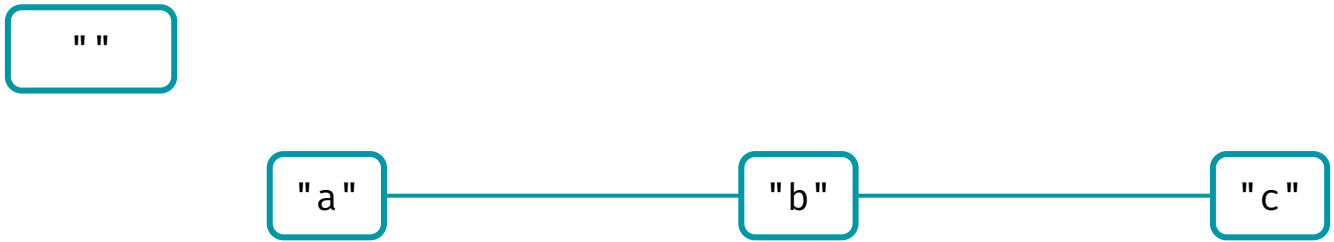
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldl (++) "" xs
```



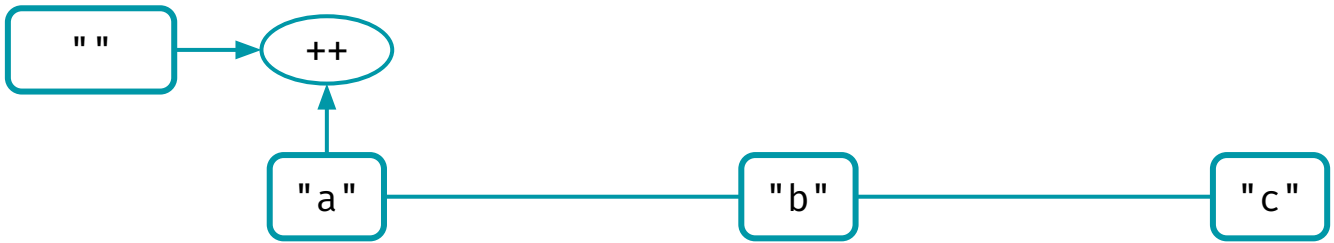
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldl (++) "" xs
```



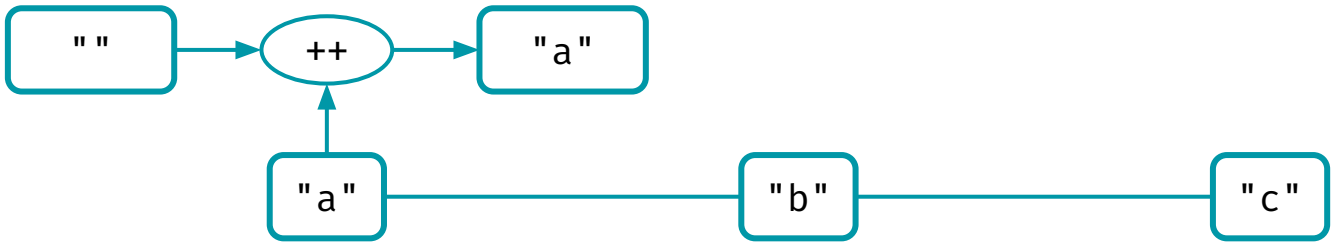
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldl (++) "" xs
```



Higher-Order Functions

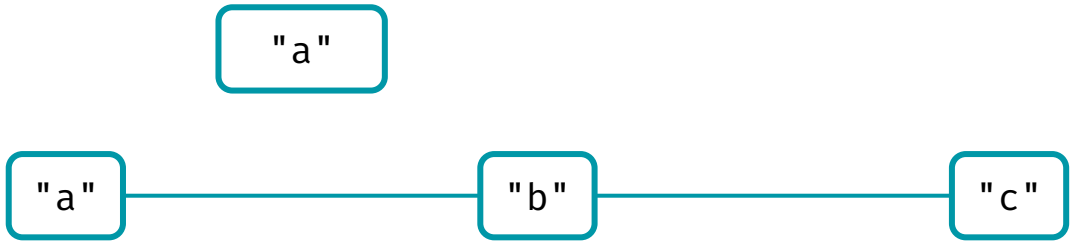
```
foldl :: (b → a → b) → b → [a] → b

cat :: [String] → String
cat xs = foldl (++) "" xs
```



Higher-Order Functions

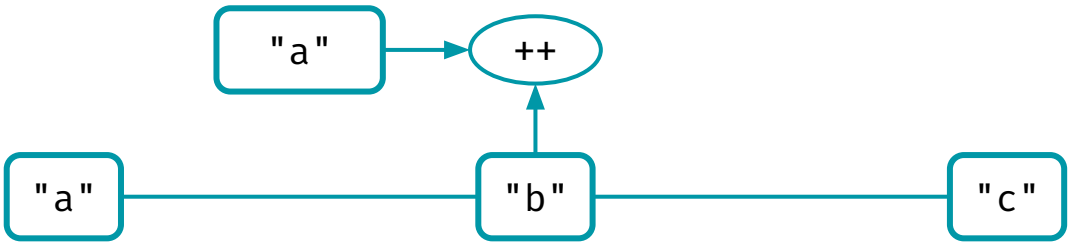
```
foldl :: (b → a → b) → b → [a] → b
```

```
cat :: [String] → String
cat xs = foldl (++) "" xs
```



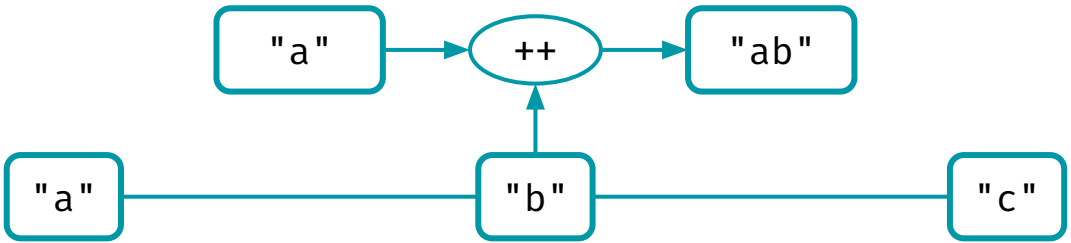
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldl (++) "" xs
```



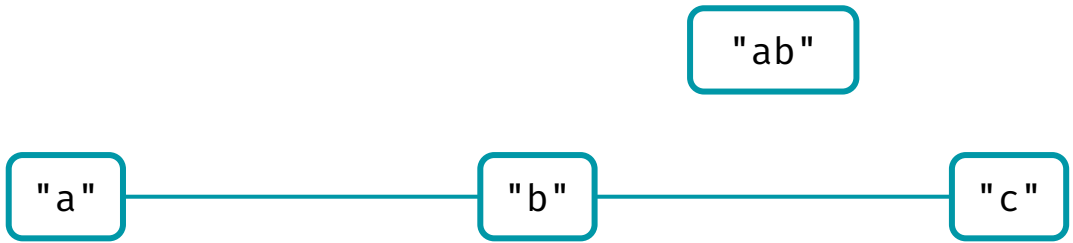
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldl (++) "" xs
```



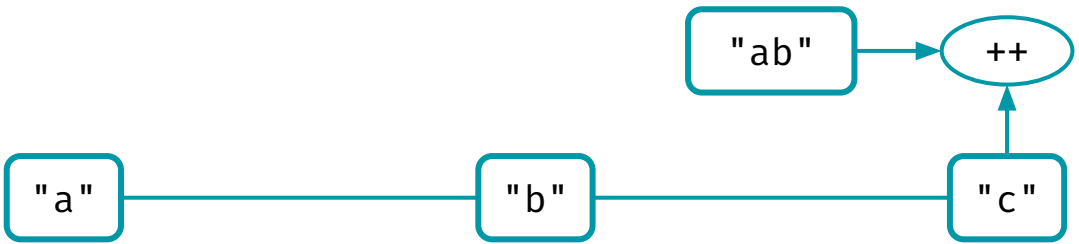
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldl (++) "" xs
```



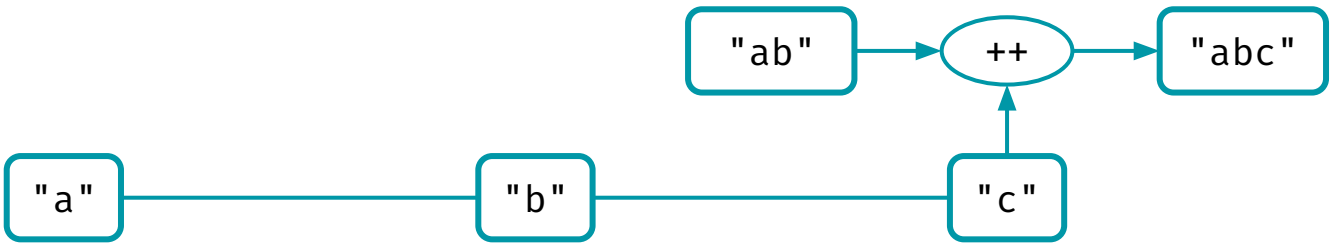
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldl (++) "" xs
```



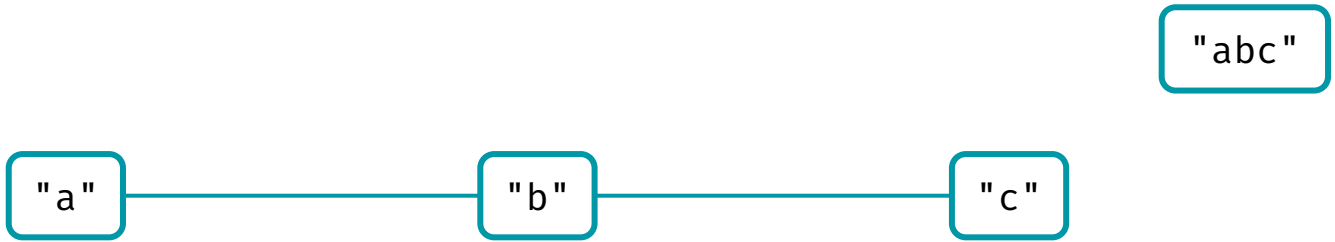
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldl (++) "" xs
```



Higher-Order Functions

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

cat :: [String] → String
cat xs = foldl (++) "" xs
```

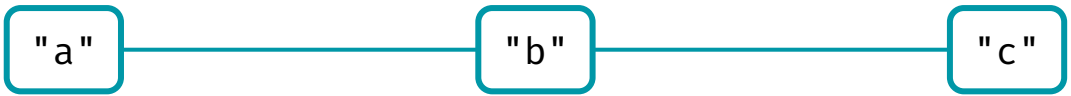


Higher-Order Functions

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

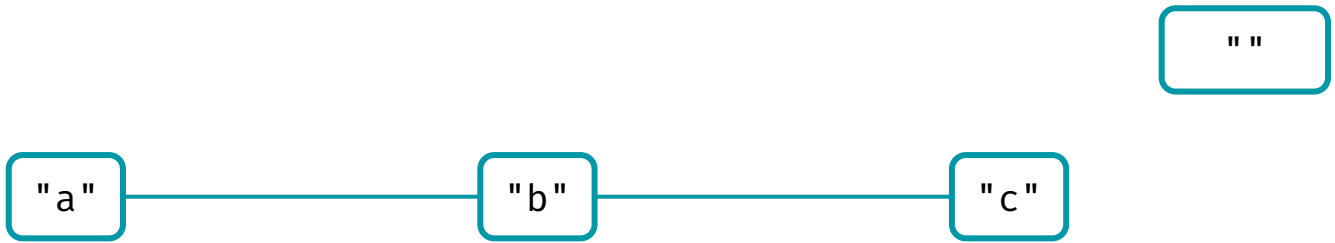
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldr (++) "" xs
```



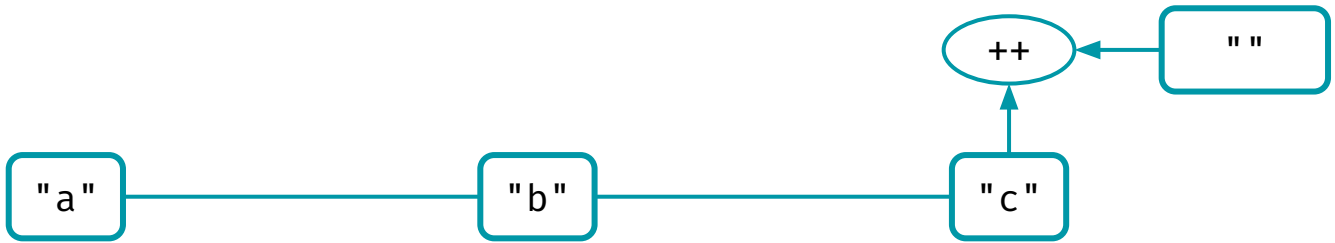
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldr (++) "" xs
```



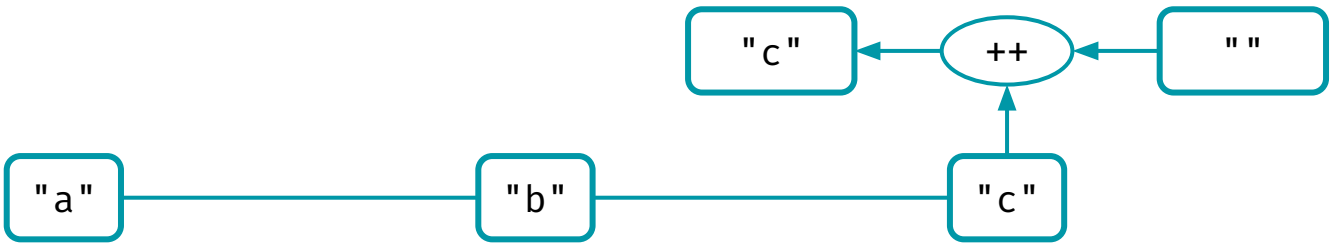
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldr (++) "" xs
```



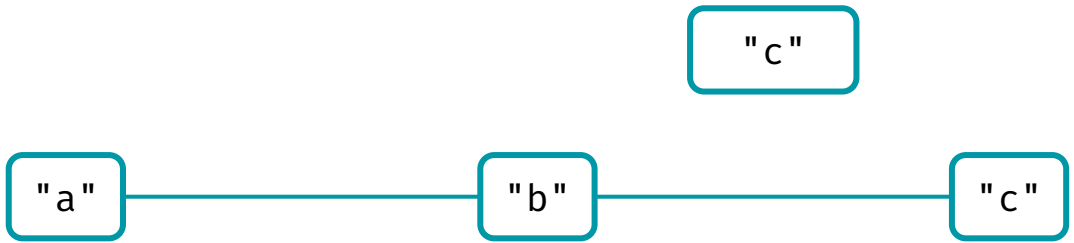
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldr (++) "" xs
```



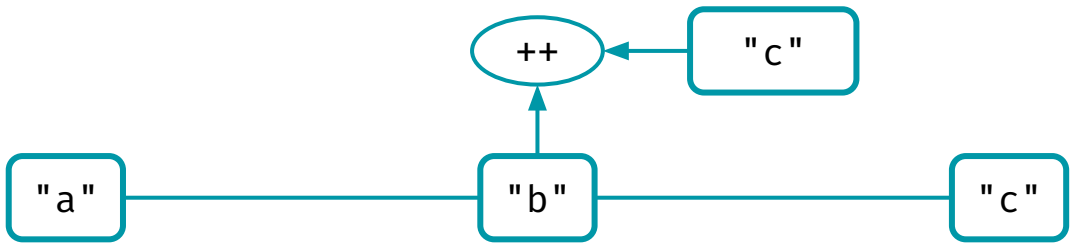
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldr (++) "" xs
```



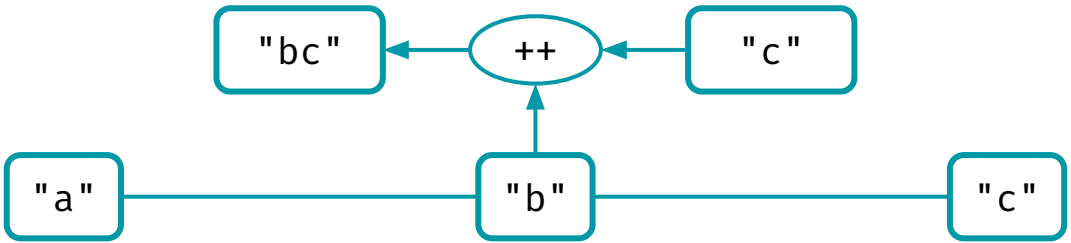
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldr (++) "" xs
```



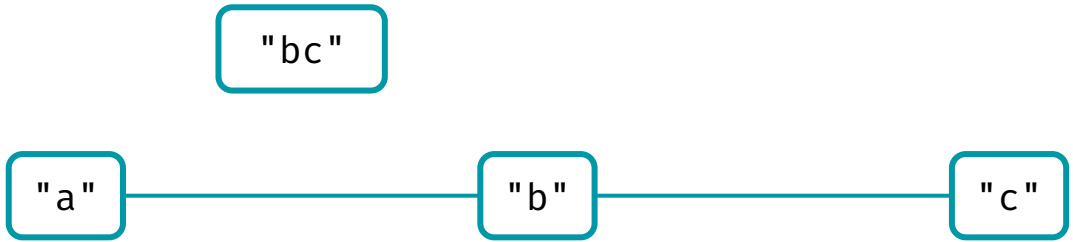
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldr (++) "" xs
```



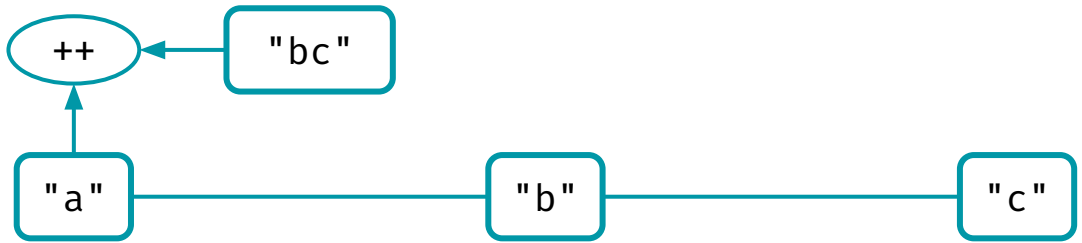
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldr (++) "" xs
```



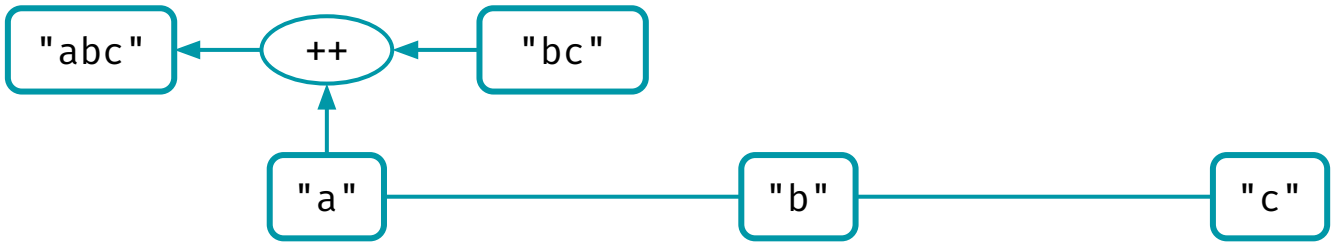
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldr (++) "" xs
```



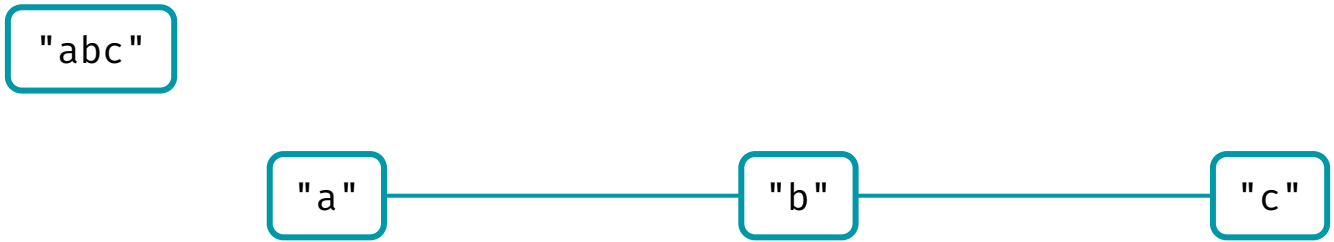
Higher-Order Functions

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

cat :: [String] → String
cat xs = foldr (++) "" xs
```



Higher-Order Functions

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

cat :: [String] → String
cat xs = foldr (++) "" xs
```



foldl vs foldr

```
-- Left:

foldl (++) "" ["a", "b", "c"]  ⟹ (("" ++ "a") ++ "b") ++ "c"


-- Right:

foldr (++) "" ["a", "b", "c"]  ⟹ "a" ++ ("b" ++ ("c" ++ ""))
```

Quiz: `foldl` vs `foldr`

```
foldr (-) 0 [1,2,3,4]
```

A. `[1,2,3,4]`
B. `-10`
C. `0`
D. `-2`
E. None of the above

Quiz: `foldl` vs `foldr`

```
foldr (-) 0 [1,2,3,4]
```

A. `[1,2,3,4]`
B. `-10`
C. `0`
D. `-2`
E. None of the above

Quiz: `foldl` vs `foldr`

```
foldr (-) 0 [1,2,3,4]
```

A. `[1,2,3,4]`
B. `-10`
C. `0`
D. `-2`
E. None of the above

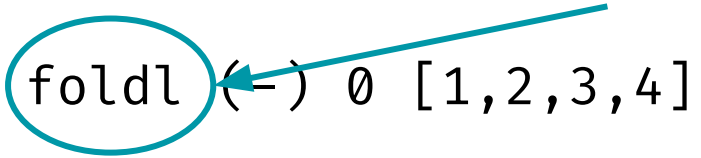
```
  1 - (2 - (3 - (4 - 0)))
= 1 - (2 - (3 - 4))
= 1 - (2 - (-1))
= 1 - 3
= -2
```

Quiz: `foldl` vs `foldr`

```
foldl (-) 0 [1,2,3,4]
```

A. `[1,2,3,4]`
B. `-10`
C. `0`
D. `-2`
E. None of the above

Quiz: `foldl` vs `foldr`



```
foldl (-) 0 [1,2,3,4]
```

A. `[1,2,3,4]`
B. `-10`
C. `0`
D. `-2`
E. None of the above

Quiz: `foldl` vs `foldr`

```
foldl (-) 0 [1,2,3,4]
```

A. `[1,2,3,4]`
B. `-10`
C. `0`
D. `-2`
E. None of the above

Quiz: `foldl` vs `foldr`

```
foldl (-) 0 [1,2,3,4]
```

A. `[1,2,3,4]`
B. `-10`
C. `0`
D. `-2`
E. None of the above

```
  ((0 - 1) - 2) - 3) - 4
= (-1 - 2) - 3) - 4
= (-3 - 3) - 4
= -6 - 4
= -10
```

Practice

```
reverse :: [a] -> [a]

reverse xs = foldl f base xs

  where

    f a x =

    base =
```

Practice

```
last :: [a] -> a

last [] = error “last: empty list”

last (x:xs) = foldl f base xs

  where

    f a x =

    base =
```

Practice

```
append :: [a] -> [a] -> [a]

append xs ys = foldr f base l

  where

    f x a =

    base =

    l =
```

Practice

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

map f xs = foldr fold_fun base xs
```

Practice

```
filter :: (a -> Bool) -> [a] -> [a]

filter p xs = foldr f base xs
```