yes23

We acknowledge, celebrate and pay our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Read more:

- the Voice to Parliament
- the Yes23 campaign
- ANU's support for the Voice
- mis and disinformation

Higher order functions

Contents

Higher order functions

(THOMPSON: §11)

A **higher order function** is a function that

- takes another function as input, or
- returns a function as output

Mastering higher order functions will make you a more productive programmer

Example: the identity function (Thompson: §11)

Recall the parametric polymorphic function

```
id :: a -> a
id x = x
```

An instantiation is

```
id :: (Int -> Char) -> (Int -> Char)
```

This is a **higher order function** because it has a function as input

It is also a **higher order function** because it has a function as output

Associativity of ->

Recall that

```
(Int -> Char) -> (Int -> Char)
```

is the same as

```
(Int -> Char) -> Int -> Char
```

but it is **not** the same as

```
Int -> Char -> (Int -> Char)
Int -> Char -> Int -> Char
```

Associativity of ->

Any function with more than one input is a higher order function

```
-- Consider a function
f :: Int -> String -> Char

-- Giving it input 3 :: Int gives a function
f 3 :: String -> Char

-- Adding another input gives a Char
f 3 "Haskell" :: Char
```

Giving `f` only one input is called **partial application**

Partial application (Thompson: §11.3)

Consider the function

```
multiply :: Int -> (Int -> Int)
multiply x y = x * y
```

A partial application is

```
multiply 2 :: Int -> Int
```

Then we have

```
ghci> multiply 2 4
8
ghci> (multiply 2) 4
8
```

Partial application in general

The expressions

```
f :: t_1 -> t_2 -> ... -> t_n -> t
f e_1 e_2 ... e_k
```

are short for

```
f :: t_1 -> (t_2 -> ( ... -> (t_n -> t) ... ))
( ... ((f e_1) e_2) ... e_k)
```

In other words:

- `->` is **right-associative**
- function application is **left-associative**

Functions as input: applyTwice

```haskell
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

```
ghci> applyTwice sqrt 16
2.0
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " haha") "hey"
"hey haha haha"
ghci> applyTwice ("haha " ++) "hey"
"haha haha hey"
ghci> applyTwice (3:) [1]
[3,3,1]
```

Functions as input and output: Composition (Thompson: §11.1)

Recall from Week 1 that the **composition** of two functions `f :: a -> b` and `g :: b -> c` is a function

```
g . f :: a -> c
```

We can view **composition** itself as a higher order function, which takes two functions and returns one function

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(.) g f x = g (f x)
```

We usually write `(.) g f` infix as `g . f`

```
ghci> ((+1) . (*2)) 1
3
```

Example: `myOdd` via composition

We have built-in functions

```
even :: Int -> Bool
not :: Bool -> Bool
```

Assume we want to define

```
myOdd :: Int -> Bool
myOdd x = not (even x)
```

We can write this more succinctly as

```
myOdd' :: Int -> Bool
myOdd' = not . even
```

`map`

(Thompson: §10.2)

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

The result of `map f xs` is the list obtained by applying `f` to each element of `xs`

Example

```
ghci> map (+1) [1, 2, 3, 4]
[2,3,4,5]
ghci> map even [1, 2, 3, 4]
[False, True, False, True]
```

zipWith

(Thompson: §10.2)

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

The function `zipWith` generalises `zip` by zipping with the function given as the first argument, instead of a tupling function

Example

```
ghci> zipWith (+) [1, 2, 3, 4] [4, 3, 2]
[5,5,5]
ghci> zipwith (++) ["An", "example"] ["?", "!", "hello"]
["An?","example!"]
```

filter

(Thompson: §10.2)

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

The function `filter`, applied to a predicate and a list, returns the list of those elements that satisfy the predicate

Example

```
ghci> filter even [1, 2, 3, 4]
[2,4]
ghic> filter (<3) [1, 2, 3, 4]
[1,2]
ghci> filter (== "Hask") ["This","is","Hask","Hask"]
["Hask","Hask"]
```

Folds

(Thompson: §10.1)

- Sometimes called “reduce” functions
- A **fold** is traversing a list element by element, building an output as we go
- There are **right folds** and **left folds**, we focus on right folds first
- Folds are the most useful higher order function and are worth understanding, they will make you a more productive programmer!

Folds

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

Example

- 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
- et cetera

Folds

(Thompson: §10.3)

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


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

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


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

How are they **different**?

How are they **similar**?

Folds

(Thompson: §10.3)

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


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

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


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

The **names** are different, but Haskell does not really care about that

Folds

(Thompson: §10.3)

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


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

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


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

The **types** are different, which suggests that folds are polymorphic

Folds

(Thompson: §10.3)

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


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

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


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

Thebase cases are different

Folds

(Thompson: §10.3)

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


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

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


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

The **step cases** are slightly different: they are different recipes to combine the head with a recursive call on the tail

Folds

(Thompson: §10.3)

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


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

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


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

Everything else is **the same**!

Fold right

(Thompson: §10.3)

This suggests that we can define a polymorphic higher order function with input

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

that does the rest of the work for us

This saves us from writing time-consuming error-prone recursions

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

combining operation   input   output

base case

See also the Haskell documentation

Fold right in action

The function

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

has **instantiation**

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

We can define `mySum` and `myProd` via

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

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

Fold right in action

(Thompson: §10.3)

The function

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

has **instantiation**

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

We can define `mySum` and `myLen` via

```
myLen :: [a] -> Int
myLen = foldr (\x y -> y + 1) 0

-- or
myLen' :: [a] -> Int
myLen' = foldr (\_ y -> y + 1) 0
```

Fold right in action

The function

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

has **instantiation**

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

We can define `acro` via

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

Try it yourself!

(THOMPSON: §10.3)

The general function

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

Try to write the following functions using `foldr`

```
myAnd :: [Bool] -> Bool
myAnd list = case list of
  []    -> True
  x:xs -> x && (myAnd xs)

myConcat :: [[a]] -> [a]
myConcat list = case list of
  []    -> []
  x:xs -> x ++ (myConcat xs)
```

Try it yourself! (Thompson: §10.3)

The general function

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

The functions `myAnd` and `myConcat` via `foldr`

```
myAnd :: [Bool] -> Bool
myAnd = foldr (&&) True

myConcat :: [[a]] -> [a]
myConcat = (++) ""
```

Defining `foldr`

The general function

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

Another example

(Thompson: §10.3)

The general function

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

Try to write the following function using `foldr`

```
myReverse :: [a] -> [a]
myReverse = foldr snoc []
        where snoc x xs = xs ++ [x]

-- or with an anonymous function
myReverse' :: [a] -> [a]
myReverse' = foldr (\x xs -> xs ++ [x]) []
```

Next

Assignment 1

- Deadline: Friday 8 September, 11pm sharp
- Drop-ins during teaching break:
  - Wednesday 6 September at 12 noon
  - Friday 8 September at 6pm

Labs this week

- Recursion with lists, and more

Lectures

- Back in week 7 with HOF, trees, and more!