Programming In Haskell Chapter 11

CS 1JC3

Higher Order Functions

Functions are Data in Haskell, and can be treated just like data of any other type. This is known as first class functions and allows for the following useful properties

- Functions can be combined using operators, just like numbers can be combined using $+, -, /$,etc

Higher Order Functions

Functions are Data in Haskell, and can be treated just like data of any other type. This is known as first class functions and allows for the following useful properties

- Functions can be combined using operators, just like numbers can be combined using $+, -, /$,etc
- Haskell provides Lambda Expressions, which allow to define functions directly as expressions without a name

Higher Order Functions

Functions are Data in Haskell, and can be treated just like data of any other type. This is known as first class functions and allows for the following useful properties

- Functions can be combined using operators, just like numbers can be combined using $+, -, /$,etc
- Haskell provides Lambda Expressions, which allow to define functions directly as expressions without a name
- Functions can be used as inputs or outputs of other functions. A function that takes another function(s) as argument(s) are known as Higher Order Functions

Higher Order Functions

Functions are Data in Haskell, and can be treated just like data of any other type. This is known as first class functions and allows for the following useful properties

- Functions can be combined using operators, just like numbers can be combined using $+, -, /$,etc
- Haskell provides Lambda Expressions, which allow to define functions directly as expressions without a name
- Functions can be used as inputs or outputs of other functions. A function that takes another function(s) as argument(s) are known as Higher Order Functions
- Syntax allows for partial application of functions, so that partially applied functions return another function as a result (i.e by currying)

Function Composition

- A function is f composed of g (written in mathematics as $f \cdot g$) to specify the function that applies g, and feeds its output into f

Function Composition

- A function is f composed of g (written in mathematics as $f \cdot g$) to specify the function that applies g, and feeds its output into f

- This is specified by the definition of function composition in Haskell

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Function Composition

- A function is f composed of g (written in mathematics as $f \cdot g$) to specify the function that applies g, and feeds its output into f

- This is specified by the definition of function composition in Haskell

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

- Defining new functions in terms of composition is usually done with implicit parameterization

```
f = (foldr (+) 0) . (map (+1))
    -- f [0,0,0] = 3
```

The Identity Function

- The most boring function Haskell has to offer!

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

- or is it .... consider the following properties

```haskell
map id xs == xs
f . id == id . f
f . id == f
```

Combining Functions with Operators

- The $ operator is a function combinator useful as an alternative to parenthesis (sometimes)

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

- Consider these two versions of the same function

```
--Version 1
jimmy xs ys = foldr max 0 (filter even
                (map (+1) (concat (xs:[ys]))))

-- Version 2
jimmy xs ys = foldr max $ filter even $
                map (+1) $ concat $ xs:[ys]
```

Combining Functions with Operators

- Consider another simple operator we could define to help reduce parenthesis buildup
  Note: unlike previous functions this is not predefined in Prelude

```
(|>) :: a -> (a -> b) -> b
x |> f = f x
```

Combining Functions with Operators

- Consider another simple operator we could define to help reduce parenthesis buildup
  Note: unlike previous functions this is not predefined in Prelude

```
(|>) :: a -> (a -> b) -> b
x |> f = f x
```

- Consider these two versions of the same code

```
import Data.List
import Data.Char
-- Version 1
putStrLn $
 map toLower $
 concat      $
 intersperse " "
 ["HELLO","GOODBYE"]
```

```
-- Version 2
["HELLO","GOODBYE"] |>
   intersperse " "  |>
   concat           |>
   map toLower      |>
   putStrLn
```

The Syntax of Application and $->$

- Function Application is left associative, which means

```
f x y == (f x) y
f x y /= f (x y)
```

- The function symbol $->$ is right associative, which means

```
a -> b -> c
--means
a -> (b -> c)
-- NOT
(a -> b) -> c
```

Currying and UnCurrying

- The standard definition of a Haskell function uses currying and allows for partial application

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

- while an uncurried function can be constructed by bundling the arguments into a tuple

```haskell
add :: (Int,Int) -> Int
add (x,y) = x + y
```

Note: unless we anticipate having a function operate on tuples, we generally like our functions curried

Higher Order Functions

- We've seen many functions that take other functions as arguments: map, foldr, filter, etc.
- These are known as Higher Order Functions. To refresh our memory, consider the following

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f xs     []     = []
zipWith f []     ys     = []
zipWith f (x:xs) (y:ys) = (f x y) : zipWith f xs ys
```

- Note: zipWith takes the function f as an argument, the parenthesis in the type signature indicate this

Higher Order Functions

- Functional programming makes extensive use combining Higher Order Functions to express various computation through familiar patterns
- And Haskell provides use with many mechanism's to express ourselves. Consider the following code

```haskell
even x = x `mod` 2 == 0
succ x = x + 1

f :: Integral a => [a] -> [a]
f xs = filter even (map succ xs)
f xs = filter even $ map succ xs                -- OR
f = \xs -> filter even $ map succ xs            -- OR
f = filter even . map succ                      -- OR
f = filter (\x -> x `mod` 2 == 0) . map (+1)
```

Constructors Are Functions To!

- Consider the following data type

```
data Student = StudentC { name :: String
                        , ident :: Int }
```

- and the following code

```
buildData :: [String] -> [Int] -> [Student]
buildData xs ys = zipWith StudentC xs ys
```

Exercise 1

Define a function

```
iter :: Int -> (a -> a) -> (a -> a)
```

that iterates application of a function like so

```
iter 3 f = f . f . f
```

Exercise 1

Define a function

```
iter :: Int -> (a -> a) -> (a -> a)
```

that iterates application of a function like so

```
iter 3 f = f . f . f
```

Solution

```
iter :: Int -> (a -> a) -> a
iter 0 f = id
iter n f = f . (iter (n-1) f)
```

Exercise 2

Find operator sections sec1 and sec2 so that

```
map sec1 . filter sec2
```

has the same effect as

```
filter (>0) . map (+1)
```

Exercise 2

Find operator sections sec1 and sec2 so that

```
map sec1 . filter sec2
```

has the same effect as

```
filter (>0) . map (+1)
```

Solution:

```
map (+1) . filter (>=0)
```

Exercise 3

Consider the data type

```
data Positive = Positive Integer
    deriving Show
```

Construct a function

```
fromIntList :: [Integer] -> [Positive]
```

such that only Integers ≥ 0 are included in the output (and do so using function composition of map and filter)

Exercise 3

Consider the data type

```
data Positive = Positive Integer
    deriving Show
```

Construct a function

```
fromIntList :: [Integer] -> [Positive]
```

such that only Integers ≥ 0 are included in the output (and do so using function composition of map and filter)

Solution:

```
fromIntList = map Positive . filter (>=0)
```

Exercise 4

Define a function

```
curry :: ((a, b) -> c) -> a -> b -> c
```

that takes an uncurried function and converts it into a curried one

Exercise 4

Define a function

```
curry :: ((a, b) -> c) -> a -> b -> c
```

that takes an uncurried function and converts it into a curried one

```
curry f x y = f (x,y)
```

Exercise 5

Consider the Tree type

```
data BinTree a = Node (BinTree a) (BinTree a) a
               | Leaf a
          deriving (Show,Eq,Foldable)
```

define a function

```
treeToList :: BinTree a -> [a]
```

that converts a BinTree to a list using only foldr

Exercise 5

Consider the Tree type

```
data BinTree a = Node (BinTree a) (BinTree a) a
               | Leaf a
          deriving (Show,Eq,Foldable)
```

define a function

```
treeToList :: BinTree a -> [a]
```

that converts a BinTree to a list using only foldr

Solution:

```
treeToList = foldr (:) []
```

Exercise 6

Consider the Tree type

```
data Tree a = TNode [Tree a] a
          deriving (Show,Eq,Foldable)
```

define a function

```
treeToList :: Tree a -> [a]
```

that converts a Tree to a list using only foldr

Exercise 6

Consider the Tree type

```
data Tree a = TNode [Tree a] a
        deriving (Show,Eq,Foldable)
```

define a function

```
treeToList :: Tree a -> [a]
```

that converts a Tree to a list using only foldr

Solution:

```
treeToList = foldr (:) []
```

Exercise 7