Higher-Order Functions 02

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

The `zipWith` Function

- Consider the `zipWith` Function

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

- `zipWith` takes a function and two lists as arguments, and then joins the two lists by applying the function between corresponding elements

```
ghci> zipWith (+) [1,2,3] [3,7,5]
[4,9,8]
ghci> zipWith (+) [1,2,3] [3,7]
[4,9]
ghci> zipWith (+) [1,2] [3,7,5]
[4,9]
ghci> zipWith max [2,4,1] [1,5,3]
[2,5,3]
ghci> zipWith min [2,4,1] [1,5,3]
[1,4,1]
```

- Let's define our own `zipWith`

The `myZipWith` Function

- Obviously, if a given list is empty, `myZipWith` should return the empty list
- However, if given lists are `x:xs` and `y:ys` and the given function is `f`:
  - We need to apply the function `f` to `x` and `y`
  - The result becomes the first element of the resulting list
  - What about lists `xs` and `ys`?
  - Zip them with the same function and use the result as the rest of the resulting list
- Here is our `myZipWith`:

```
myZipWith _ []  _ = []
myZipWith _  _ [] = []
myZipWith f (x:xs) (y:ys) = f x y : myZipWith f xs ys
```

The `myZipWith` Function

- Some tests:

```
ghci> myZipWith (++) ["Hello", "Love"] [" World", " Haskell"]
["Hello World","Love Haskell"]
ghci> myZipWith (++) (myZipWith (++) ["Hello", "Love"] [" ", " "])
                                     ["World", "Haskell"]
["Hello World","Love Haskell"]
ghci> myZipWith (*) (replicate 5 10) [1..]
[10,20,30,40,50]
ghci> myZipWith (myZipWith (*)) [[1,2],[3,4]] [[5,6],[7,8]]
[[5,12],[21,32]]
```

- `myZipWith (*)` is a function of type `[c] -> [c] -> [c]`

The `flip` Function

- Let's practice with the function `flip`

```
ghci> :t flip
flip :: (a -> b -> c) -> b -> a -> c
```

- The `flip` function takes a function and returns a function where arguments are flipped
- Mathematically

$$\texttt{flip}(f(x,y)) = f(y,x)$$

- This can be easily implement in Haskell since functions are treated as data

```
myFlip :: (a -> b -> c) -> b -> a -> c
myFlip f x y = f y x
```

The `flip` Function

- Some tests:

```
ghci> zip [1,2,3] "Dog"
[(1,'D'),(2,'o'),(3,'g')]
ghci> myFlip zip [1,2,3] "Dog"
[('D',1),('o',2),('g',3)]
ghci> zipWith (/) [1,2,3] [4,5,6]
[0.25,0.4,0.5]
ghci> myFlip (zipWith (/)) [1,2,3] [4,5,6]
[4.0,2.5,2.0]
```

- Note that `zipWith (/)` is a function that takes two arguments

```
ghci> :t zipWith (/)
zipWith (/) :: Fractional c => [c] -> [c] -> [c]
```

The `map` Function

- The `map` function is provided by the standard library
- Let's look at its definition and try to understand how it works

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

- `map` applies the given function to every element on the given list

```
ghci> map (*2) [1,2,3,4]
[2,4,6,8]
ghci> map (++ "!!!") ["Dog", "Cat", "Fish"]
["Dog!!!","Cat!!!","Fish!!!"]
ghci> map fst [(1,5), (3,2), (4,1), (7,3)]
[1,3,4,7]
```

- Note that the result of `map (*2) [1,2,3,4]` is the same as

```
[x * 2 | x <- [1,2,3,4]]
```

- `map` tends to make code much more readable

The `filter` Function

- Similarly, the `filter` function is provided by the standard library
- The `filter` function takes a predicate and a list, and returns the list of elements that satisfy that predicate
- Try to come up with a definition...

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

- Here are some examples:

```
ghci> filter (<=10) [1..20]
[1,2,3,4,5,6,7,8,9,10]
ghci> filter even [1..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> filter (`elem` ['A'..'Z']) "Most Significant Bit"
"MSB"
```

The `filter` Function

- To apply multiple predicates:
  - Filtering multiple times

```
ghci> filter (<10) (filter even [1..20])
[2,4,6,8]
```

  - Join the predicate with the logical `&&` function

```
ghci> let f x = x < 10 && even x in filter f [1..20]
[2,4,6,8]
```

- How about another quick sort using filter?

```
quickSort [] = []
quickSort (x:xs) = sortedLessThanOrEq ++ [x] ++ sortedGreaterThan
          where sortedLessThanOrEq = quickSort (filter (<= x) xs)
                sortedGreaterThan  = quickSort (filter (> x) xs)
```

The `filter` Function

- **Exercise**: Given positive integers `d` and `n`, find the largest number under `d` that is divisible by `n`
  - Example: Largest number under 100000 that is divisible by 97 is 99910
  - For simplicity, assume that `d` and `n` are positive and `n < d`
  - Any idea?
- Try them all starting from `d - 1` down to `1`:

```
[d-1,d-2..1]
```

- The condition is `mod` by `n` is `0`
- Filter with the condition and get the first one only (`head`):

```
largest d n = head (filter p [d-1,d-2..1])
              where p x = mod x n == 0
```

- Some test

```
ghci> largest 100 97
97
ghci> largest 100000 97
99910
ghci> largest 100000 4971
99420
```

The `takeWhile` Function

- The `takeWhile` function takes a predicate and a list
- Starting at the beginning of the list, it returns the list's element as long as the predicate holds true

```
ghci> :t takeWhile
takeWhile :: (a -> Bool) -> [a] -> [a]
ghci> takeWhile (<'t') "I love Haskell"
"I lo"
ghci> takeWhile even [2,12,6,24,7,22,9,10]
[2,12,6,24]
ghci> takeWhile (/=' ') "What is the first word?"
"What"
```

The `takeWhile` Function

- **Exercise**: Find the sum of all odd squares that are less than `n`
- Example: sum of all odd squares that are less than 30 is $1^2 + 3^2 + 5^2 = 1 + 9 + 25 = 35$
- For simplicity generate all squares:

```
map (^2) [1,2..]
```

- Filter just those that are ood:

```
filter odd (map (^2) [1,2..])
```

- Take all elements start at the beginning until it is not less than a give number `n`:

```
takeWhile (<n) (filter odd (map (^2) [1,2..]))
```

- Now we have the list of all odd squares that are less than `n`
- Just need to sum them up:

```
sumOddSq n = sum (takeWhile (<n) (filter odd (map (^2) [1,2..])))
```

- Can also use the list comprehension:

```
sumOddSq' n = sum (takeWhile (<n) [x | x <- [y^2 | y <- [1,2..]], odd x])
```