Computer Science 312

Higher-Order Functions in Haskell

Convert a String to Uppercase

```haskell
import qualified Data.Char (toUpper)

toUpper :: String -> String
toUpper [] = []
toUpper (x:xs) = Data.Char.toUpper x : toUpper xs
```

Transform a list of characters into another list of characters, by taking each character in the source list into an expression

The pattern is head/tail recursion

Use a List Comprehension

```
import qualified Data.Char (toUpper)

toUpper :: String -> String
toUpper [] = []
toUpper (x:xs) = Data.Char.toUpper x : toUpper xs
```

```
import qualified Data.Char (toUpper)

toUpper :: String -> String
toUpper str = [Data.Char.toUpper ch | ch <- str]
```

Expression

Each character

Source list

List Comprehensions

Syntax:

```
[<expression> | <item> <- <source list>]
```

Read as “Take each item from a source list into an expression.”

```
Prelude> [x ^ 2 | x <- [1..4]]
[1, 4, 9, 16]

Prelude> [sqrt x | x <- [4, 9, 16]]
[2.0, 3.0, 4.0]
```

Can Also Use Pattern Matching

Syntax:

```
[<expression> | <pattern> <- <source list>]
```

```
type StudentList = [(String, Float)]

averageGPA :: StudentList -> Float
averageGPA list = sum [gpa | (name, gpa) <- list] / length list
```

```
Prelude> averageGPA [("Stan",3.56),("Ann",4.0),("Bill",2.95)]
3.50333333333333
```

Can Add Conditions for Filtering

Syntax:

```
[<expression> | <item> <- <source list>, <cond-1>, ..,
                                        <cond-n>]
```

Fetch all student names whose GPAs are >= 3.5

```
type StudentList = [(String, Float)]

honorRoll :: StudentList -> [[Char]]
honorRoll = [name | (name, gpa) <- list, gpa >= 3.5]
```

```
Prelude> honorRoll [("Stan",3.56),("Ann",4.0),("Bill",2.95)]
["Stan","Ann"]
```

Quicksort in Python – 21 Lines

```
def quicksort(lyst):
    quicksortHelper(lyst, 0, len(lyst) - 1)

def quicksortHelper(lyst, left, right):
    if left < right:
        pivotLocation = partition(lyst, left, right)
        quicksortHelper(lyst, left, pivotLocation - 1)
        quicksortHelper(lyst, pivotLocation + 1, right)

def partition(lyst, left, right):
    middle = (left + right) // 2
    pivot = lyst[middle]
    lyst[middle] = lyst[right]
    lyst[right] = pivot
    boundary = left
    for index in range(left, right):
        if lyst[index] < pivot:
            swap(lyst, index, boundary)
            boundary += 1
    swap (lyst, right, boundary)
    return boundary

def swap(lyst, i, j):
    lyst[i], lyst[j] = lyst[j], lyst[i]
```

Quicksort in Haskell – 5 Lines

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    quicksort [y | y <- xs, y <= x] ++ [x] ++
    quicksort [y | y <- xs, y > x]
```

- Select the head as the pivot item
- Shift the smaller items to its left
- Shift the larger items to its right
- Quicksort each of those lists
- Glue ‘em together around the pivot item

Three Ways to Transform a List

Head/tail recursion

```
roots :: Floating a => [a] -> [a]
roots [] = []
roots (x:xs) = sqrt x : roots xs
```

Three Ways to Transform a List

Head/tail recursion

```
roots :: Floating a => [a] -> [a]
roots [] = []
roots (x:xs) = sqrt x : roots xs
```

List comprehension

```
roots :: Floating a => [a] -> [a]
roots list = [sqrt x | x <- list]
```

Three Ways to Transform a List

Head/tail recursion

```
roots :: Floating a => [a] -> [a]
roots [] = []
roots (x:xs) = sqrt x : roots xs
```

List comprehension

```
roots :: Floating a => [a] -> [a]
roots list = [sqrt x | x <- list]
```

Map

```
roots :: Floating a => [a] -> [a]
roots list = map sqrt list
```

Map Hides **`(x:hs)`** Recursion

Head/tail recursion

```
roots :: Floating a => [a] -> [a]
roots [] = []
roots (x:xs) = sqrt x : roots xs
```

Generalize to a map by adding a function argument

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

Function types are expressed in parentheses in the signatures

lambda Expressions

List comprehension

```
Prelude> [x ^ 2 | x <- [1..5]]
[1,4,9,16,25]
```

Map with named function

```
Prelude> map square [1..5]
[1,4,9,16,25]
```

Map with a lambda expression

```
Prelude> map (\x -> x ^ 2) [1..5]
[1,4,9,16,25]
```

Syntax

```
\<argument-1> .. <argument-n> -> <expression>
```

Filtering

Head/tail recursion

```
allEvens :: Integral a => [a] -> [a]
allEvens [] = []
allEvens (x:xs)
    | even x = x : allEvens xs
    | otherwise = allEvens xs
```

Filtering

Head/tail recursion

```
allEvens :: Integral a => [a] -> [a]
allEvens [] = []
allEvens (x:xs)
    | even x = x : allEvens xs
    | otherwise = allEvens xs
```

List comprehension

```
allEvens :: Integral a => [a] -> [a]
allEvens list = [x | x  <- list, even x]
```

Filtering

Head/tail recursion

```
allEvens :: Integral a => [a] -> [a]
allEvens [] = []
allEvens (x:xs)
    | even x = x : allEvens xs
    | otherwise = allEvens xs
```

List comprehension

```
allEvens :: Integral a => [a] -> [a]
allEvens list = [x | x  <- list, even x]
```

Filter

```
allEvens :: Integral a => [a] -> [a]
allEvens list = filter even list
```

Filtering

Head/tail recursion

```
roots :: Floating a => [a] -> [a]
roots [] = []
roots (x:xs) = sqrt x : roots xs
```

Generalize to a filter by adding a predicate argument

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

Example

```
Prelude> filter (\x -> x `mod` 3 == 0) [1..27]
[3,6,9,12,15,18,21,24,27]
```

Reducing a List to a Single Item

```
Prelude> sum [1..10]
55

Prelude> product [1..10]
3628800
```

Reducing a List to a Single Item

```
mySum :: Num a => [a] -> a
mySum [x] = x
mySum (x:xs) = x + mySum xs
```

```
mySum [3, 5, 7] ->
    3 + mySum [5, 7] ->
        5 + mySum[7]->
            <- 7
        <- 12
    <- 15
```

Computations move from right to left.

Reducing = Folding

- Reducing is called ***folding*** in Haskell. There are four flavors of folding:
  - **`foldr1`** – folds from the right, list must be nonempty
  - **`foldr`** – folds from the right, list can be empty, must supply a base value
  - **`foldl1`** – folds from the left, list must be nonempty
  - **`foldl`** – folds from the left, list can be empty, must supply a base value

Generalize: Folding from the Right

```
mySum :: Num a => [a] -> a
mySum [x] = x
mySum (x:xs) = x + mySum xs
```

A function of two args

```
myFoldr1 :: (a -> a -> a) -> [a] -> a
myFoldr1 _ [x] = x
myFoldr1 f (x:xs) = f x (myFoldr1 f xs)
```

There must be at least one item in the list!

Include the Empty List, Too

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

```
mySum :: Num a => [a] -> a
mySum list = myFoldr (+) 0 list
```

```
myProduct :: Num a => [a] -> a
myProduct list = myFoldr (*) 1 list
```

Folding from the Left

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

```
myFoldl (+) 0 [3, 5, 7] ->
    myFoldl (+) 3 [5, 7] ->
        myFoldl (+) 8 [7] ->
            myFoldl(+) 15 [] ->
            <- 15
        <- 15
    <- 15
```

Tail Recursive, but Lazy!

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

- Haskell functions postpone evaluating arguments until they're needed for computations (***lazy evaluation***)
- Thus, the evaluation of the second argument, the function application **`(f x baseValue)`**, is postponed ***all the way down***
- This requires extra overhead to track these calls

For next time

Exploiting laziness

Currying

Partial functions

Infinite lists!