COMP105 Lecture 16

More Higher Order Functions

Outline

Today

- `scan`
- `takeWhile`
- `dropWhile`
- `zipWith`

Relevant book chapters

- Programming In Haskell Chapter 7
- Learn You a Haskell Chapter 6

Recap: `foldr`

Folds turn a list into a **single value**

```
ghci>  foldr (+) 0 [1,2,3,4]
10
```

```
ghci>  foldr (*) 1 [1,2,3,4]
24
```

```
ghci> foldr (++) [] ["one", "two", "three"]
"onetwothree"
```

Recap: `foldr`

`foldr` has an **accumulator** that is modified as the list is processed

```
ghci> foldr1 (\ x acc -> x ++ " " ++ acc)
                         ["one", "two", "three"]
"one two three"
```

```
acc = "three"
acc = "two" ++ " " ++ "three"
acc = "one" ++ " " ++ "two three"
```

scan

Scan is like fold, but it outputs the accumulator at each step

```
ghci> scanr (+) 0 [1,2,3,4]
[10,9,7,4,0]
```

```
ghci> scanr (*) 1 [1,2,3,4]
[24,24,12,4,1]
```

```
ghci> scanr1 (\ x acc -> x ++ " " ++ acc)
                                ["one", "two", "three"]
["one two three","two three","three"]
```

scanr implementation

```haskell
scanr' :: (a -> b -> b) -> b -> [a] -> [b]

scanr' _ init [] = [init]
scanr' f init (x:xs) =
    let
        recursed = scanr' f init xs
        new = f x (head recursed)
    in
        new : recursed
```

scan variants

There are also **left to right** versions of `scan`

```
ghci> scanl (+) 0 [1..10]
[0,1,3,6,10,15,21,28,36,45,55]
```

```
ghci> scanr (+) 0 [1..10]
[55,54,52,49,45,40,34,27,19,10,0]
```

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

Fibonacci the higher order way

```
fib_pairs n = scanl (\ (a, b) _ -> (b, a + b))
                                    (0, 1) [1..n]
```

```
ghci> fib_pairs 7
[(0,1),(1,1),(1,2),(2,3),(3,5),(5,8),(8,13),(13,21)]
```

```
fib_to_n n = map fst (fib_pairs n)
```

```
ghci> fib_to_n 7
[0,1,1,2,3,5,8,13]
```

Exercise

What do these functions do?

```
mystery list = scanl (\ acc x -> 2*x + acc) 0 list
```

```
mystery2 list = scanl1 (\ acc x -> max acc x) list
```

```
mystery3 list = foldr (\ x (a, b) ->
                          if x `elem` "aeiou"
                          then (x:a, b)
                          else (a, x:b))
                      ("", "") list
```

takeWhile

takeWhile takes from a list **while a condition** is true

```
ghci> takeWhile (<=5) [1..10]
[1,2,3,4,5]
```

```
ghci> takeWhile (/=' ') "one two three"
"one"
```

```
ghci> takeWhile (\ x -> length x <= 2)
                                  ["ab", "cd", "efg"]
["ab","cd"]
```

takeWhile implementation

```haskell
takeWhile' :: (a -> Bool) -> [a] -> [a]

takeWhile' _ [] = []
takeWhile' f (x:xs)
    | f x = x : takeWhile' f xs
    | otherwise = []
```

dropWhile

dropWhile **drops** from a list while a condition is true

```
ghci> dropWhile (==1) [1,1,2,2,3,3]
[2,2,3,3]
```

```
ghci> dropWhile (`elem` ['a'..'z']) "smallBIG"
"BIG"
```

```
ghci> dropWhile (\x -> x < 10 && x > 0) [1,2,3,10,4,5]
[10,4,5]
```

dropWhile implementation

```haskell
dropWhile' :: (a -> Bool) -> [a] -> [a]

dropWhile' _ [] = []
dropWhile' f (x:xs)
    | f x = dropWhile' f xs
    | otherwise = x:xs
```

takeWhile and dropWhile example

```
split_words "" = []
split_words string =
    let
        first = takeWhile (/=' ') string
        up_to_space = dropWhile (/=' ') string
        after_space = dropWhile (==' ') up_to_space
    in
        first : split_words after_space
```

```
ghci> split_words "one two   three"
["one","two","three"]
```

words and unwords

The `split_words` function is called **words**

```
ghci> words "   foo   bar baz "
["foo","bar","baz"]
```

```
ghci> unwords ["foo","bar","baz"]
"foo bar baz"
```

Recap: zip

```
ghci> zip [1,2,3,4] [5,6,7,8]
[(1,5),(2,6),(3,7),(4,8)]
```

```
add_two_lists l1 l2 =
let
    zipped = zip l1 l2
in
    map (\ (x, y) -> x + y) zipped
```

```
ghci> add_two_lists [1,2,3,4] [5,6,7,8]
[6,8,10,12]
```

zipWith

zipWith zips two lists together **using a function**

```
ghci> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]
```

```
ghci> zipWith (++) ["big", "red"] ["dog", "car"]
["bigdog","redcar"]
```

```
ghci> zipWith (\ x y -> if x then y else -y)
                    [True, False, False] [1,2,3]
[1,-2,-3]
```

zipWith implementation

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

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

zipWith examples

```
mult_by_pos list = zipWith (*) list [0..]
```

```
ghci> mult_by_pos [2,3,4,5]
[0,3,8,15]
```

zipWith examples

```
interleave str1 str2 =
    let
        zipped = zipWith (\ x y -> x : y : []) str1 str2
    in
        concat zipped
```

```
ghci> zipWith (\ x y -> x : y : []) "abc" "123"
["a1","b2","c3"]
```

```
ghci> interleave "abc" "123"
"a1b2c3"
```

Exercise

```
mystery4 list = takeWhile (`elem` ['0'..'9']) list

mystery5 str1 str2 = [head str1] ++ show (length str2)

mystery6 list = zipWith mystery5 list list
```

Exercises

1. Write a function `prefix_product` which takes a list, and returns a list containing the product of all prefixes of the list. So `prefix_product [1,2,3,4]` should return `[1,2,6,24]`

2. Write a function `prefixes` that returns all prefixes of a given string. So `prefixes "abc"` should return `["","a","ab","abc"]`

3. Without looking at the definition in Prelude, write a function `scanr1'` that implements `scanr1`.

Exercises

4. Write a function `remove_spaces` that removes any leading spaces from a string.

5. Write a function `up_to` that takes two parameters `c` and `list` and returns the elements of `list` before the first occurrence of `c`.

6. Write a function `after c list`, which returns all elements of `list` after the first occurrence of `c`.

Exercises

7. Write a function `divide_lists` that takes two integer lists `l1` and `l2` and returns a new list consisting of each element of `l1` divided by the corresponding element of `l2`.

8. (*) Use `reverse` and `scanr` to produce an implementation of `scanl'`.

Summary

- `scan`
- `takeWhile`
- `dropWhile`
- `zipWith`

Next time: Higher order programming examples