CS 421 --- Higher Order Functions Activity

Manager	Keeps team on track	
Recorder	Records decisions	
Reporter	Reports to class	
Reflector	Assesses team performance	

Learning Objectives

Mapping, folding, and zipping allow us to abstract away common list computations. Knowing how to use them will make you more productive as a programmer.

1. Reduce code size by using `map`, `foldr`, and `zipWith`.
2. Discover how to take a fix-point.
3. Use type signatures to implement `curry`, `uncurry`, and `flip`.

Mapping and Folding

Consider the following code, implementing three common higher order functions:

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

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr z xs)

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

Problem 1) Use `map` to write a function that negates the elements of a list. Here is the recursive version.

```
negList [] = []
negList (x:xs) = - x : negList xs
```

Problem 2) Use `foldr` to write a function that returns the sums of the squares of the elements of a list.[1] (E.g., `sumSqr [3,4]` will return `25`.) Here is the recursive version.

```
sumSqr [] = 0
sumSqr (x:xs) = x * x + sumSqr xs
```

Problem 3) Could you have used `map` to rewrite the above function? Why or why not?

Problem 4) How would you describe the relationship between `map` and `zipWith`?

Infinities

Consider this code, which deals with ``infinite'' lists.

```
foo = 1 : foo
bar = 1 : map (+1) bar
baz = map (**) bar
quuz = 1 : 1 : zipWith (+) quuz (tail quuz)
```

Problem 5) What do each of the above definitions do? Remember to use `take` if you try to type these in to the REPL.

Problem 6) The fix-point of function f is a value x such that $f(x) = x$. Write a function `fix :: (a -> a) -> a -> a` that takes a function f and returns its fix-point.

```
Prelude> cos 1
0.5403023058681398
Prelude> cos (cos 1)
0.8575532158463934
Prelude> fix cos 1
0.7390851332151607
```

Write the function `fix`.

[1] The type signature of `foldr` is actually a bit more general than this, but we will talk about that later.

List Comprehensions

List comprehensions are similar to higher order functions, and can allow you to write very compact code.

```
Prelude> stuff = [8,6,7,5,3,0,9]
Prelude> [ x+1 | x <- stuff ]
[9,7,8,6,4,1,10]
Prelude> [ x+1 | x <- stuff, x > 5]
[9,7,8,10]
Prelude> [ x+1 | x <- stuff, x > 5, even x]
[9,7]
Prelude> [ x + y | x <- stuff, y <- [10,20]]
[18,28,16,26,17,27,15,25,13,23,10,20,19,29]
```

Problem 7) What is the purpose of the `x <- stuff` expression?

Problem 8) What is the purpose of `x > 5`, and `even x`?

Problem 9) How do you describe the order in which `x` and `y` are created in the last example?

Problem 10) What does the following code do?

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

Currying

Problem 11) Write a function `curry :: ((a,b) -> c) -> a -> b -> c` that takes a function that takes a pair and returns an equivalent function that takes its arguments one at a time.

```
Prelude> let plus (a,b) = a + b
Prelude> :t plus
Num a => (a,a) -> a
Prelude> let cplus = curry plus
Prelude> cplus 10 20
30
```

Problem 12) Write a function `flip :: (a -> b -> c) -> (b -> a -> c)` that takes a function that takes two arguments and returns an equivalent function where the arguments have been reversed.

```
Prelude> let sub a b = a - b
Prelude> flip sub 10 2
-8
```

Problem 13) Consider the types of `flip` and `curry`. Can you write another function that has either of those types? Try to prove it.