Higher Order Functions Activity

Mattox Beckman

Why

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.

Learning Objectives

1. Discover how to reduce code by using `map`, `foldr`, and `zipWith`.
2. Discover how to take a fix-point.
3. Transform the interface of a function using `curry`, `uncurry`, and `flip`.

Mapping and Folding

1. Use `map :: (a->b) -> [a] -> [b]` to write a function that negates the elements of a list. Here is the recursive version.

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

2. Use `foldr :: (a->b->b) -> b -> [a] -> b` 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
```

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

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

Other HOFs

4. Try to write the higher order function `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`. Here's a sample run:

```
Prelude> zipWith (+) [10,20,30] [33,44,94]
[43,64,124]
```

   It is like `map`, but for two lists instead of one. If one list is longer than the other, simply discard the remaining elements.

5. Here is a related function `foldl :: (b -> a -> b) -> b -> [a] -> b`:

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

   Compare and contrast it to `foldr`.

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
```

Currying

7. 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
```

8. 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
```