Mapping and Folding

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

- Define the `foldr` and `map` functions.
- Use `foldr` and `map` to implement two common recursion patterns.

Let's Talk about Mapping

Incrementing Elements of a List

```
1 incL [] = []
2 incL (x:xs) = x+1 : incL xs
```

```
incL [7,5,6,4,2,-1,8]  ⇒ [8,6,7,5,3,0,9]
```

Mapping Functions the Hard Way

What do the following definitions have in common?

Example 1

```
1 incL [] = []
2 incL (x:xs) = x+1 : incL xs
```

Example 2

```
1 doubleL [] = []
2 doubleL (x:xs) = x*2 : doubleL xs
```

Mapping functions the hard way

Example 1

```
1 incL [] = [] ← Base Case
2 incL (x:xs) = x+1 : incL xs
                      ^^^^^^^
                      Recursion
```

Example 2

```
1 doubleL [] = [] ← Base Case
2 doubleL (x:xs) = x*2 : doubleL xs
                         ^^^^^^^^^
                         Recursion
```

- Only two things are different:
  - The operations we perform
  - The names of the functions

Mattox's Law of Computing

The computer exists to work for us; not us for the computer. If you are doing something repetitive for the computer, you are doing something wrong.
Stop what you're doing and find out how to do it right.

Mapping Functions the Easy Way

Map Definition

$$map\ f\ [x_0, x_1, \ldots, x_n] = [f\ x_0, f\ x_1, \ldots, f\ x_n]$$

```
1 map :: (a->b) -> [a] -> [b]
2 map f [] = []
3 map f (x:xs) = f x : map f xs
4
5 incL = map inc
6
7 doubleL = map double
```

- `inc` and `double` have been transformed into recursive functions.
- I dare you to try this in Java.

Let's Talk about Folding

What do the following definitions have in common?

Example 1

```
1 sumL [] = 0
2 sumL (x:xs) = x + sumL xs
```

Example 2

```
1 prodL [] = 1
2 prodL (x:xs) = x * prodL xs
```

foldr

Fold Right Definition

$$\mathit{foldr}\ f\ z\ [x_0, x_1, \ldots, x_n] \;=\; f\ x_0\ (f\ x_1\ \cdots\ (f\ x_n\ z)\cdots)$$

- To use `foldr`, we specify the function *and* the base case.

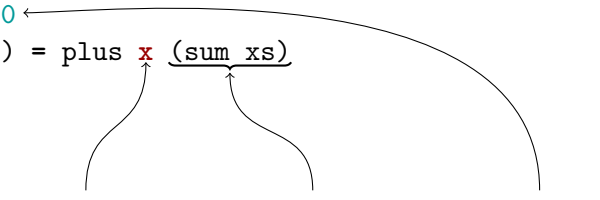
```
1 foldr :: (a -> b -> b) -> b -> [a] -> b
2 foldr f z [] = z
3 foldr f z (x:xs) = f x (foldr f z xs)
4
5 sumlist = foldr (+) 0
6 prodlist = foldr (*) 1
```

Encoding Recursion using `fold`

- Notice the pattern between the recursive version and the higher order function version.

Recursive Style

```
1 plus a b = a + b
2 sum [] = 0
3 sum (x:xs) = plus x (sum xs)
```




```
1 sum = foldr (\a b -> plus a b) 0
```

Some Things to Think About …

- These functions scale to clusters of computers.
- You can write `map` using `foldr`. Try it!
- You cannot write `foldr` using `map` — why not?

Other Higher Order Functions

- There are many other useful higher order functions!
- Investigate these:
  `all`, `any`, `zipWith`, `takeWhile`, `dropWhile`, `flip`
- Remember you can use `:t` in the REPL to reveal the type of an expression.