Today

- Higher-Order Functions (cont)
- Logic Programming Basics

Assignments

- HW 10 due
- Proj Status Update 2 due
- HW 11 out
- Extra credit 1 out (due by end of semester)
- Proj Presentations next Thurs
Accumulating values (from the left) with foldl

foldl :: (a -> b -> a) -> a -> [b] -> a

- (a -> b -> a) is the **step function**
- first a is an **accumulator**
- [b] are the input values
- last a is the **accumulated value**
- new_accumulator = step_function accumulator b_value

Prelude> foldl (+) 0 [1, 2, 3]
6

- This just sums up the list of values

**How it works ...**

```haskell
foldl _ acc [] = acc
foldl step acc (x:xs) = foldl step (step acc x) xs
```

```haskell
foldl (-) 9 [5, 3, 1]
==&gt; foldl (-) (9 - 5) [3, 1]
==&gt; foldl (-) (((9 - 5) - 3) - 1) []
==&gt; ((9 - 5) - 3) - 1
==&gt; 0
```
Accumulating values from the right with foldr

\[ \text{foldr :: (b -> a -> a) -> a -> [b] -> a} \]

Prelude> foldr (+) 0 [1, 2, 3]  -- same as foldl in this case
6

- Similar to foldl but works right-to-left
- new_accumulator = step_function b_value accumulator
- ... where the b_value is from the list

How it works ...

\[
\begin{align*}
\text{foldr} & \quad \text{acc} \, [] \ = \ \text{acc} \\
\text{foldr} \ \text{step} \ \text{acc} \ (x:xs) \ &= \ \text{step} \ x \ \text{foldr} \ \text{step} \ \text{acc} \ xs
\end{align*}
\]

\[
\text{foldr} \ (-) \ 9 \ [5, 3, 1] \quad \text{-- try 3 instead of 9}
\]

\[
\begin{align*}
\Rightarrow & \quad 5 \ - \ \text{foldr} \ (-) \ 9 \ [3, 1] \\
\Rightarrow & \quad 5 \ - \ (3 \ - \ \text{foldr} \ (-) \ 9 \ [1]) \\
\Rightarrow & \quad 5 \ - \ (3 \ - \ (1 \ - \ \text{foldr} \ (-) \ 9 \ [1])) \\
\Rightarrow & \quad 5 \ - \ (3 \ - \ (1 \ - \ 9)) \\
\Rightarrow & \quad -6
\end{align*}
\]
Many recursive functions follow the fold pattern

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

Q: How can `filter` be defined using `foldr`?

```haskell
filter' p xs = foldr step [] xs
  where step x acc
        | p x = x : acc
        | otherwise = acc
```

- For example ...

```haskell
filter' odd [1,2,3]
  ==> foldr step [] [1,2,3]
  ==> step 1 (foldr step [] [2,3])
  ==> 1 : (foldr step [] [2,3])
  ==> 1 : (step 2 (foldr step [] [3])))
  ==> 1 : (foldr step [] [3])
  ==> 1 : (step 3 foldr step [] [])
  ==> 1 : (3 : (foldr step [] []))
  ==> 1 : (3 : [])
```

Q: How can `filter` be defined using `foldl`?

```haskell
filter' p xs = foldl step [] xs
  where step acc x
        | p x = acc ++ [x]
        | otherwise = acc
```
We can also define map using foldr

\[ \text{map'} :: (a -> b) -> [a] -> [b] \]
\[ \text{map'} f \ xs = \text{foldr step [ ]} \ xs \]
\[ \text{where step} \ x \ ys = f \ x : ys \]

For example ...

\[ \text{map'} \ \text{odd} \ [1,2,3] \]
\[ ==> \text{foldr step [ ]} \ [1,2,3] \]
\[ ==> \text{step 1} (\text{foldr step [ ]} \ [2,3]) \]
\[ ==> \text{odd 1} : (\text{foldr step [ ]} \ [2,3]) \]
\[ ==> \text{odd 1} : (\text{step 2} (\text{foldr step [ ]} \ [3])) \]
\[ ==> \text{odd 1} : (\text{odd 2} : (\text{foldr step [ ]} \ [3])) \]
\[ ==> \text{odd 1} : (\text{odd 2} : (\text{step 3} (\text{foldr step [ ]} \ []))) \]
\[ ==> \text{odd 1} : (\text{odd 2} : (\text{odd 3} : (\text{foldr step [ ]} \ []))) \]
\[ ==> \text{odd 1} : (\text{odd 2} : (\text{odd 3} : [])) \]

Why care about these higher-order functions?

- In general, should use them whenever possible ...
- ... Can make functions easier to understand (shorter)
- ... Well behaved (fewer bugs)
- ... Optimization
List Comprehensions

List comprehensions mimic set definitions ("set builder" notation):

\[ A \times B = \{(a, b) \mid a \in A \land b \in B\} \]

- The cartesian product operation

Using list comprehensions:

\[
\text{cprod } xs \ ys = [\langle x, y \rangle \mid x \leftarrow xs, y \leftarrow ys]
\]

- here \(xs\) and \(ys\) have to be in scope

Another example:

\[
[x \ast 2 \mid x \leftarrow [1..10]]
\]

With a predicate (to filter)

\[
[x \ast y \mid x \leftarrow [1..10], y \leftarrow [1..3], \text{even } x]
\]

In general, a lot like lambda functions

- but for defining lists "on the fly"
- syntactic sugar for defining lists (list set builders)
- can use anywhere you’d expect a list
Wrapping up

What we didn’t cover

- A lot! ... Haskell has many features

- The IO type (recall “purity”)
  - Various I/O operations
  - All the normal things you’d expect to write real apps

- Monads
  - A pattern (data type) to chain together a list of operations
  - Functional machinery to sequence commands
  - The do expression is an example
    
    ```haskell
    do input <- readFile inputFile
    putStrLn (function input)
    ```
  - Appears imperative, but is still functional ...

- The Monad typeclass has a sequencing function >>=

  ```haskell
  Prelude> :type (>>=)
  (>>=) :: (Monad m) => m a -> (a -> m b) -> m b
  ```
  - extracts (unwraps) value on left from the Monad
  - passes it to a function that returns a wrapped result value
• And a `return` function

```haskell
Prelude> :type return
return :: (Monad m) => a -> m a
```

– Takes a value and wraps it into a Monad member

• Here is a (silly) example

```haskell
wrapIt x = Just x

go = wrapIt "hi" >>=
    \v1 -> wrapIt (head v1) >>=
    \v2 -> return v2

Prelude> go
Just 'h'
```

– `return` for `Maybe` is defined to create a `Just` value
– i.e., `return x = Just x`

• The `do` keyword provides a shorthand ...

```haskell
go' = do
    v1 <- wrapIt "hi"
    v2 <- wrapIt (head v1)
    return v2
        # return does the wrapping
```

• Here is a better example using the `IO` monad

```haskell
do
    putStrLn "What is your name?"
    name <- getline
    putStrLn ("Nice to meet you " ++ name)
```