A. More Haskell

- Chapter 6: Higher-order functions
  - Revisit currying
  - map, filter
  - Unnamed ("lambda") functions; function definitions as `function_name = lambda function`. 
  - folding a list
- From Chapter 7: Making our own types.... 
  - Algebraic datatypes - enumerations, simple recursive structures, constructor patterns

Skip Ch 5 (recursion)

Ch 6 Higher order functions

- Recall: A higher order function is a function that takes another function as a parameter or returns a function as a result. Haskell uses curried functions: Instead of `f : Int x Int x Int -> Int` as in most languages in Haskell, we would have `f :: Int -> Int -> Int -> Int`.
- Arrow is right-associative, so `f :: Int -> (Int -> (Int -> Int))` is equivalent.
- So `f 17` is a function that you can use like any other function.
- (It's a partially-applied version of `f`.)

```haskell
> :{
|  f :: Int -> Int -> Int -> Int
|  f a b c = a * b + c
|  :}
> :t f
f :: Int -> Int -> Int -> Int
> f 3 5 8
23
> g = f 3  -- f 3 b c = 3 * b + c
> :t g
so g b c = 3 * b + c
> h = g 5
> :t h
h :: Int -> Int
> h 8
23
```

- Recall we needed to give `g 5` a name so that we wouldn't try to print its value (causes an error).
> f 5 -- can't print a function value

<interactive>:74:1: error:
  • No instance for (Show (Int -> Int)) arising from a use of ‘print’

• When you say `variable = expression`, it gives the variable a value but it doesn't print the value out.
  > z = 5
  > z -- evaluate z so I can see the value
  5
  > h = g 5
  > -- now we have an h; if I ask for it's value, I get an error
  > h

<interactive>:80:1: error:
  • No instance for (Show (Int -> Int)) arising from a use of ‘print’
    (maybe you haven't applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

• More functions on lists

• Recall `map :: (a -> b) -> [a] -> [b]` a function on a values, a list of a values, returns the list of results
  > map sqrt [2..5]
  [1.4142135623730951,1.7320508075688772,2.0,2.23606797749979]

• Similar is `filter`, which takes a boolean test function `(a -> Bool)` and an a list and returns the values of
  the list that pass the test:
  > positive x = if x > 0 then True else False
  > positive x = x > 0 -- is equivalent
  > map positive [3, 5, -1, 2, -9, 7, -2, -3]
  [True,True,False,True,False,True,False,False]
  > filter positive [3, 5, -1, 2, -9, 7, -2, -3]
  [3,5,2,7]

• Another example: find values divisible by 3
  > divisible_by_3 x = mod x 3 == 0 -- using mod as a prefix function
  > divisible_by_3 x = x `mod` 3 == 0 -- equivalent defn, using mod as a
    binary operator
  > divisible_by_3 6
  True
  > filter divisible_by_3 [27..83]
  [27,30,33,36,39,42,45,48,51,54,57,60,63,66,69,72,75,78,81]

• Filter and find last value
> largest_multiple_of_3 x = last (filter divisible_by_3 x)
> largest_multiple_of_3 [27..83]
81

- [Added 9/3] Function composition (infix period) is a built-in higher-order function: \((f \circ g) \, x = f \, (g \, x)\).

**Lambda Functions**
- It can be annoying to write a function like `largest_multiple_of_3` just to use it in one spot
- We can use unnamed functions instead. Below, \(x \rightarrow x \mod 3 == 0\) is a function that takes an \(x\) and returns true if \(x \mod 3\) is zero.

\[
\begin{align*}
> &-- \text{divisible\_by\_3} \, x = x \mod 3 == 0 \\
> &-- \text{divisible\_by\_3} = \lambda x -> x \mod 3 == 0 -- \text{same as previous line} \\
> &\text{divisible\_by\_3} \, 6 \\
&\text{True}
\end{align*}
\]
- Backslash is Haskell’s representation of greek lowercase lambda.
  - The actual lambda notation would be \(\lambda x \cdot x \mod 3 == 0\).
  - In fact, function declarations like \(f \, x = exp\) is short for \(f = \lambda x -> exp\)
  - This makes function definitions like other variable definitions like \(y = 3\) (variable = value). Warning: If \(f\) is recursive then \(f = \lambda x -> \text{recursive expression}\) doesn’t work in the regular lambda calculus (works okay here)\(^1\).

\[
\begin{align*}
> f \, a \, b \, c &= a \ast b + c \\
> f &= \lambda a -> \lambda b -> \lambda c -> a \ast b + c \\
&\quad \text{the function that takes a and returns a function that takes b}....
> f \, 3 \, 5 \, 8 \\
&23
\end{align*}
\]
- [added 9/3] The abbreviation \(\lambda a \, b \, c \rightarrow a \ast b + c\) means \(\lambda a -> \lambda b -> \lambda c -> a \ast b + c\)
- Lambdas are useful for short things you use once.

\[
\begin{align*}
> &\text{filter positive [3, 5, -1, 2, -9, 7, -2, -3]} \\
&[3,5,2,7] \\
> &\text{filter (\(\lambda x -> x > 0\) [3, 5, -1, 2, -9, 7, -2, -3]} \\
&[3,5,2,7] \\
> &\text{positive x = x > 0 -- usual definition syntax} \\
> &\text{positive = \(\lambda x -> x > 0\) -- using lambda}
\end{align*}
\]

-------------- ended here 2019-08-28

**Folding lists**
- Folding a list lets you combine its elements using some operation, like adding together a list of numbers.
- `foldl` takes a binary operation, a starting value, and the list to fold.

\(^1\) Forgot this proviso in lecture -- we’ll see later that Haskell’s laziness lets \(f = \lambda x -> ....\) use of \(f\) work recursively.
> foldl (-) 0 [3,5,8]
-16

• For foldl, the starting value goes on the left, and the operations are done left-to-right.
  > (((0 - 3) - 5) - 8)
  -16

• foldr goes right-to-left, with the starting value at the right end.
  > foldr (-) 0 [3,5,8]
  6
  > (3 - (5 - (8 - 0)))
  6

• Restricted to just looking at lists, the types of foldl and foldr are
  - foldl :: (b -> a -> b) -> b -> [a] -> b
  - foldr :: (a -> b -> b) -> b -> [a] -> b

• In foldl (-) 0 [3, 5, 8] and foldr (-) 0 [3,5,8], we use Int for type variables a and b and get
  - foldl :: (Int -> Int -> Int) -> Int -> [Int] -> Int
  - foldr :: (Int -> Int -> Int) -> Int -> [Int] -> Int

• Note that since we use Int for both a and b, the types of foldl and foldr are the same.

• Since + is associative, foldl and foldr return the same result when given the same arguments:
  foldl (+) 0 [3,5,8] and foldr (+) 0 [3,5,8] both return 16.

• Since + is commutative, you can reorder the elements: foldl (+) 0 [3,5,8] = foldl (+) 8 [0,5,3].

• On the other hand, - is not associative, so foldl and foldr can return different values on the same arguments. (We saw this above) Since - is not commutative, reordering the elements can change the result:
  foldl (-) 0 [1, 2] ≠ foldl (-) 0 [2, 1].

The Foldable type class

• The types of foldl and foldr actually use t a instead of [a], where Foldable t. (I.e., t is an instance of Foldable.) Here, t is a type constructor (an operator that takes one type and gives you back another), not a type itself. When we go from type a to type [a], we're applying the list type constructor [...] to the type a to get another type.

• Foldable is for building types that behave like lists: They have to allow folding and mapping, finding an element, sum and product (when Foldable is given a numeric type) and minimum and maximum (when Foldable is given an order-able (Ord) type).

• We'll define our own type constructors when we get to algebraic datatypes (soon!)

• In foldl (-) 0 [3, 5, 8], we use Int for the type variables a and b in (b -> a -> b) -> b -> t a -> b, and we use […] (list-of) as t.

Another Example of folding

• The Learn You ... book has an example that uses folding to define our own elem function. (elem y ys is true if y is a member of list ys.) The use of foldl here uses different types for a and b in

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

Specifically, for b we use Bool and a we use Int (more generally, any Eq type)
foldl :: (Bool -> Int -> Bool) -> Bool -> [Int] -> Bool

- So we'll define an `elem2` function and have `elem2 y ys` look for a `y` amongst the `ys`.
  - The function we pass to `foldl` accumulates the boolean result of the question “Have we found a `y` yet?” For the initial test we pass `False` (no `y` in `[]`) and then test the 1st element of the list. The result comes back `True` or `False` depending on whether or not it was `y`, and then we continue. Note once the accumulated result becomes `True`, it never becomes `False` after that.
  - Define `found y acc next = if acc then True else next == y -- i.e., (acc || next == y)`
  - Then `elem2 y ys = foldl (found y) False ys`
    - Starting with `false`, search left-to-right to see if we can find a `y`.
    - As we search, the accumulated result is the boolean “Have we found `y` yet?”

```
> found y acc next = (next == y || acc)
> elem2 y ys = foldl (found y) False ys
:~ elem2
elem2 :: (Foldable t, Eq a) => a -> t a -> Bool

> elem2 0 []
False
> elem2 3 [1,2,3,4]
True
```

- Let's look at how `elem2 3` behaves.
- We have `elem2 3 ys = foldl (found 3) False ys` and `found 3 acc next = (acc || next == 3)`
- To shorten things, define `f = found 3`, then `acc `f` val` returns true if the accumulated value is already true, otherwise it checks the value against `3`, returning true for the new accumulated search result if it does find `3` and false if it doesn't.

```
> f = found 3
> foldl f False [1,2,3,4]     -- find a 3?
> True
> foldl f False [1,2]         -- don't find a 3
> False
> acc1 = False `f` 1          -- check 1st element of list
> acc1
False
> acc2 = acc1 `f` 2           -- check 2nd element of list
> acc2
False
> acc3 = acc2 `f` 3           -- 3rd element of list finds a 3
> acc3
True
> acc4 = acc3 `f` 4           -- no matter what next value is
> acc4
True
```

```
The ghci commands :info and :browse

- The ghci command :info gives you information about a function or class. For a type class, it tells you what operations it supports and what instances there are.

  ```
  > :info head
  head :: [a] -> a  -- Defined in 'GHC.List'
  > :info Eq
  class Eq a where
  (==) :: a -> a -> Bool
  (=/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (=/=) #-}
  -- Defined in 'GHC.Classes'
  instance Eq a => Eq [a] -- Defined in 'GHC.Classes'
  -- [... many other instances omitted ...]
  ```

- The ghci command :browse Prelude gives you information about the standard Prelude library. (The general syntax is :browse package_name, but we haven't talked about packages yet.) For type classes, it doesn't show the instances, but even so, there's almost 300 lines of output.

  ```
  > :browse Prelude
  -- lines omitted
  (&&) :: Bool -> Bool -> Bool  -- logical and
  (++) :: [a] -> [a] -> [a]  -- list concatenation
  (.) :: (b -> c) -> (a -> b) -> a -> c  -- function composition
  -- lines omitted
  class Eq a where
  (==) :: a -> a -> Bool
  (=/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (=/=) #-}  -- no list of instances
  type FilePath = String
  -- lines omitted
  ```

Algebraic types

- We can define our own types like trees, etc.

  ```
  > data Color = Red | Blue
  > :info Color
  data Color = Red | Blue  -- Defined at <interactive>:224:1
  > :info Red
  data Color = Red | ...  -- Defined at <interactive>:224:14
  ```
• Define a function on Colors:

```haskell
> :{
>     opposite Red = Blue -- Red used as a pattern, Blue as an expression
>     opposite Blue = Red -- and vice versa
> }:
```

• Oops -- forgot that datatypes create values that aren’t automatically printable.

```haskell
> opposite Red

<interactive>:231:1: error:
• No instance for (Show Color) arising from a use of ‘print’
• In a stmt of an interactive GHCi command: print it
```

• Oops again -- they aren’t automatically testable for equality

```haskell
> opposite Red == Blue

<interactive>:232:1: error:
• No instance for (Eq Color) arising from a use of ‘==’
• In the expression: opposite Red == Blue
• In an equation for ‘it’: it = opposite Red == Blue
```

• Nice feature: deriving clauses ask Haskell to generate equality and print functions for a datatype.

```haskell
> data Color = Red | Blue deriving (Eq, Show)
> Red -- Now a color value can be printed
Red
> :{
|     opp Red = Blue
|     opp Blue = Red
| }:
> opp Red
Blue
> opp Red == Blue -- can test for equality
True
```

• We can have constructor functions if we supply a parameter type to a constructor name. Below, NONE is a constructor constant, AnInt is a constructor function, and Int is the argument type for AnInt. You can use AnInt in expressions and patterns.

```haskell
> data MaybeInt = NONE | AnInt Int deriving (Eq, Show)
> AnInt 17
> :{
|     gotOne :: MaybeInt -> [Char]
|     gotOne NONE = "NO"
|     gotOne ( AnInt 0 ) = "ZERO"
|     gotOne ( AnInt _ ) = "NON-ZERO"
> }:
```
• Since \texttt{MaybeInt} does not appear on the right side of the \texttt{data} definition, it is not a recursive type. (For it to appear on the r.h.s., it would have to be a parameter (or part of a parameter) to one of a constructor function.)

• \texttt{MaybeInt} is actually just like the Prelude \texttt{Maybe Int}

• data \texttt{Maybe a} = \texttt{Nothing} | \texttt{Just a} -- \texttt{a} is a type parameter

• For an example of a recursive datatype, here’s a simple tree type, with the Haskell-supplied \texttt{==} and \texttt{show}.

(Leafs are labeled by integers.)

\begin{verbatim}
> data IntTree = Leaf Int | Node IntTree IntTree deriving (Eq, Show)
> :t Leaf 0
Leaf 0 :: intTree
> Leaf 0 == Leaf 1
False
> Node (Leaf 0) (Leaf 1) -- create a tree value; it gets printed
Node (Leaf 0) (Leaf 1)
> t1 = Node (Leaf 0) (Leaf 1)
> t2 = Node t1 t1
> t2 -- a tree with t1 and t2 as its left and right subtrees
Node (Node (Leaf 0) (Leaf 1)) (Node (Leaf 0) (Leaf 1))
\end{verbatim}

\textbf{Datatype constructor patterns}

• You can define functions on \texttt{IntTrees} using \texttt{Leaf} and \texttt{Node} (similarly to using : and [ ] on lists)

• Simple recursive routine to sum the values in a tree:

\begin{verbatim}
> :{
|   sumTree :: IntTree -> Int
|   sumTree (Leaf x) = x
|   sumTree (Node sub1 sub2) = sumTree sub1 + sumTree sub2
|   :}
> sumTree (Leaf 0)
0
> t1
Node (Leaf 0) (Leaf 1)
> sumTree t1
1
> t2
Node (Node (Leaf 0) (Leaf 1)) (Node (Leaf 0) (Leaf 1))
> sumTree t2
2
>}
\end{verbatim}
Activity Questions, Lecture 4

Review
1. What is a higher order function?
2. What is the associativity of arrow? (I.e., how do you parenthesize the type \( a \rightarrow b \rightarrow c \))
3. What is a curried / uncurried function? What is partial application of a curried function?
4. You can’t print functions because function types aren’t instances of … ?
5. What does map do? map \( f \) \( x \) = what list comprehension?
6. What does filter do? filter \( f \) \( x \) = what list comprehension? (Hint: Add a boolean test to a list comprehension.)

Lambda Functions
7. What is the syntax for an unnamed lambda? How does it relate to the lambda calculus representation?
8. \( \lambda x \rightarrow \lambda y \rightarrow expr \) can be abbreviated as … ?
9. Why use lambda expressions?
10. What is the usual way to write the declaration \( f = \lambda x \rightarrow \lambda y \rightarrow expr \) ? The declaration \( f = \text{lambda function} \) illustrates what principle?

Folding lists
11. What do foldl or foldr with arguments \( f \) \( x \) \( [v_1, v_2, v_3, ..., v_n] \) return? If \( x : : t1 \) and the \( v \)'s are : : \( t2 \), what type does \( f \) have to have under foldl? foldr? If \( f \) is associative, then what property holds? What if \( f \) is commutative?
12. When we say that lists are instances of Foldable, we mean ____ is an instance of ____ ?
13. Give a simple recursive definition for foldl; give one for foldr.

The ghci commands :info and :browse
14. What do :info name and :browse package do?

Algebraic types
15. The declaration \( \text{data Type} = \text{Name1} | \text{Name2} \) creates what kind of type? Give an example.
16. Repeat, on \( \text{data Type0} = \text{Name1} | \text{Name2 of Type2} \) where Type2 doesn’t involve Type0
17. Repeat when Type2 does involve Type0. Take \( \text{data Type} = \text{Name1} | \text{Name2 of (Type2, Type)} \) as an example.
18. How does using \( \text{data Type a = ...} \) change the meaning of Type? How can a be used on the r.h.s. of the declaration?
19. What does the clause deriving (Eq, Show) do when added to a data declaration?
20. How are constructor constant names used as expressions and patterns? Discuss examples.
21. How are constructor names used as patterns? Discuss examples.
Solutions to Selected Activity Questions

5. \( \text{map } f \ x = [f \ v \mid v \leftarrow x] \)
6. \( \text{filter } f \ x = [v \mid v \leftarrow x, f \ v] \)
7. The lambda calculus function \( \lambda x. \ expr \) is written in Haskell as \( \backslash x \rightarrow expr \).
8. \( \backslash x \rightarrow \backslash y \rightarrow expr \) is the same as \( \backslash x \ y \rightarrow expr \).
9. Lambda functions can be useful for short functions you only use a small number of times.
10. We usually write \( f = \backslash x \rightarrow \backslash y \rightarrow expr \) as \( f \ x = expr \). The declaration \( f = \lambda \) function is an illustrates first-class functions (it shows a function declaration is just a variable declaration where the value is a function).

...  
15. \text{data } \text{Type} = \text{Name1} \mid \text{Name2} \) creates an enumerated type where the names are the enumerated constants. \text{data } \text{Flavor} = \text{Sweet} \mid \text{Salty} \mid \text{Sour} \) would be an example.
16. \text{data } \text{Type0} = \text{Name1} \mid \text{Name2 of Type2} \) (where Type2 doesn’t involve Type0) creates a nonrecursive data structure. The \text{MaybeInt} structure was an example (we can create a \text{MaybeInt} value using a constant or with a function applied to an int).
17. If Type2 does involve Type0, we get a recursive data structure \text{data } \text{Type} = \text{Name1} \mid \text{Name2 of (Type2, Type)} \) makes Type values essentially just lists of Type2 values. \text{data } \text{IntList} = \text{None} \mid \text{Node of (Int, IntList)} \) would be an example.
18. Declaring \text{data } \text{Type} a = … makes the new type polymorphic over type variable \( a \). \text{Type} is now a type constructor, not a type. The name \( a \) can be used on the r.h.s. of the declaration as part of an of \text{ParameterType clause}. The Prelude’s \text{Maybe a} = \text{Nothing} \mid \text{Just a} \) is an example.
19. The \text{deriving} (\text{Eq}, \text{Show}) clauses tell the Haskell compiler to automatically generate functions \( = \) and \( /= \) and \text{show :: Type -> Char}. The \text{show} function is used to create a printable representation of Type values, which you need if you want to print them.
20. As expressions, constructor constants (declared without of … clauses) are values of the defined Type. Examples are \text{Red} and \text{Blue} for \text{Color}, \text{NONE} for \text{MaybeInt}, and \text{Nothing} for \text{Maybe a}. As patterns, a constructor constant matches the corresponding value of the Type. Function definition clause \text{opp Red = ...} was an example for \text{Color}, clause \text{gotOne NONE = ...} was an example for \text{AnInt}.
21. As expressions, constructor functions (declared with of \( T \)) take a parameter of type \( T \) and produce a value of the defined type. Examples are \text{AnInt :: Int -> MaybeInt}, \text{Just :: a -> Maybe a}, \text{Leaf :: Int -> IntTree}, and \text{Node :: (IntTree, IntTree) -> IntTree}. As patterns, constructor functions check a Type value to see how it was built; it can also extract the data from a Type value. The function declaration clauses \text{gotOne (AnInt 0) = ...} and \text{gotOne (AnInt _)} = ... are examples. For \text{IntTree} values, the function definition clause \text{sumTree (Leaf x) = x} is an example: It takes an \text{IntTree} value, asks if it was built using \text{Leaf}, and if it was, it binds \( x \) to the value it is a \text{Leaf} of, and we see \( x \) used on the r.h.s. of the clause. Similarly, with \text{sumTree (Node sub1 sub2) = ...}, the
function takes an IntTree value, asks if it was built using Node, and if it was, binds sub1 and sub2 to the two parts of the node value.