Notes from 9/28

Literate Haskell

A file ending with `.hs` is assumed to be Haskell code. It can contain comments using this syntax:

```
-- Double dash for a line comment
{- Delimited comments like
   this that can go on for multiple lines.
 -}
```

But instead of a regular Haskell file, you can save your file ending with `.lhs`, which stands for Literate Haskell. What that means is that you're typing comments **by default**. To embed some code, you add > to the left margin of each line:

```
> square x = x*x
```

There's no need for you to use Literate Haskell in this course, but I wanted you to understand how I'm using it to produce notes and assignment solutions. When viewing web pages on my site that are produced with Literate Haskell, you'll see a link to the source just above the table of contents – which is in the upper right or the bottom of the page. As long as the filename ends in `.lhs`, you can load that source directly into `ghci` or use `runghc` on it.

Assignment 3 help/review

Let's focus on `mapLeaves`.

Here was the data type:

```haskell
data Tree lv bv
  = Leaf {leafValue :: lv}
  | Branch {branchValue :: bv, left, right :: Tree lv bv}
  deriving (Show, Eq)
```

This is the signature we want:

```haskell
mapLeaves :: (lv1 -> lv2) -> Tree lv1 bv -> Tree lv2 bv
```

The -> represents the type of a function. When you have a -> b -> c you can treat that as a function with two parameters (a and b) and the result type c. Or, the default way that the -> is parenthesized is right-associative (a -> (b -> c)). Any function with >1 parameter can be used this way – it is called **partial application.**

```
ghci> f x y = 3*x + 2*y        -- A function with two parameters
ghci> :t f                     -- Its type:
f :: Num a => a -> a -> a      -- Think of it as (Num -> (Num -> Num))
ghci> f 5 6                    -- Complete application produces number
27
ghci> f 5                   -- Partial application produces function
f 5 :: Num a => a -> a      --  which has type (Num -> Num)
ghci> g = f 5               -- Can bind function to new variable
ghci> g 6                   -- And then provide next parameter
27
ghci> g 8
31
```

Here is our implementation of mapLeaves:

```
mapLeaves fn (Leaf lv) = Leaf (fn lv)
mapLeaves fn (Branch bv left right) =
  Branch bv (mapLeaves fn left) (mapLeaves fn right)
```

You should notice that most of these recursive functions on trees follow the same structure. There are two cases, the base case for Leaf and a recursive case for Branch. The recursive case actually uses recursion twice, on both left and right. Then it does some other stuff outside of the recursion to merge the results together somehow.

```
ghci> mapLeaves (+5) (Branch 'A' (Leaf 9) (Branch 'C' (Leaf 10) (Leaf 12)))
Branch {branchValue = 'A',
        left = Leaf {leafValue = 14},
        right = Branch {branchValue = 'C',
                        left = Leaf {leafValue = 15},
                        right = Leaf {leafValue = 17}}}
```

In that example, we're applying (+5) to each leaf. So the values start as numbers and remain numbers. But mapLeaves is also capable of changing the type. Here we use show to convert the numbers to strings:

```
ghci> mapLeaves show (Branch 'A' (Leaf 9) (Branch 'C' (Leaf 10) (Leaf 12)))
Branch {branchValue = 'A',
        left = Leaf {leafValue = "9"},
```

```
        right = Branch {branchValue = 'C',
                        left = Leaf {leafValue = "10"},
                        right = Leaf {leafValue = "12"}}}
```

Or here's an interesting example using the `const` function. `const` takes two parameters but returns its first one. With partial application, we can create a function that always returns the same result. So we can keep the same tree structure and branch values but replace every leaf value with the same string.

```
ghci> :t const
const :: a -> b -> a
ghci> const 9 3
9
ghci> const 9 "hello"
9
ghci> p = const "hello"
ghci> p "bye"
"hello"
ghci> p pi
"hello"
ghci> mapLeaves (const "BOO") (Branch 'A' (Leaf 9) (Branch 'C' (Leaf 10) (Leaf 12)))
Branch {branchValue = 'A',
        left = Leaf {leafValue = "BOO"},
        right = Branch {branchValue = 'C',
                        left = Leaf {leafValue = "BOO"},
                        right = Leaf {leafValue = "BOO"}}}
```

Bounded stack data type

A stack is a last-in first-out (LIFO) sequence with operations to push, pop, top. We'll implement a *bounded* stack where the number of elements is limited to some number specified when the stack is created.

```haskell
data BoundedStack a
  = BoundedStack { capacity :: Int, elements :: [a] }
  deriving Show
```

To design this, it can be helpful to first specify all the signatures of the functions. **Note:** compared to what we did in class, I changed the order of the two parameters to push. Passing the stack *after* the new element to add makes some of the composition syntax easier.

```haskell
new :: Int -> BoundedStack a
push :: a -> BoundedStack a -> BoundedStack a
```

```
pop :: BoundedStack a -> BoundedStack a
top :: BoundedStack a -> a
```

Creating a new stack means we specify the capacity but let the initial list of elements be the empty list.

```
new n = BoundedStack { capacity = n, elements = [] }
```

In order to push, we need to ensure (with a guard) that the capacity is positive. Then when we create the `BoundedStack`, we decrement the capacity and shove the new `elem` onto the front of the list.

```
push elem (BoundedStack cap elems)
  | cap > 0 = BoundedStack (cap-1) (elem:elems)
  | otherwise = error "Sorry, stack is full"
```

For now, if the precondition `cap > 0` is not satisfied, we'll just use `error` to throw an exception with a custom message. Here are some examples using `new` and `push`.

```
ghci> s1 = new 3
ghci> s2 = push 7 s1
ghci> s3 = push 8 s2
ghci> s4 = push 9 s3
ghci> s4
BoundedStack {capacity = 0, elements = [9,8,7]}
ghci> s5 = push 10 s4
ghci> s5
*** Exception: Sorry, stack is full
CallStack (from HasCallStack):
  error, called at /home/league/c/c/cs695/20170928.lhs:160:19 in main:Main
```

Note that we don't see the exception right after the last `push`, but only once we try to *inspect* the result. This is a result of *laziness* in Haskell, which we'll discuss later.

We used separate variable names (`s1`, `s2`, `s3`) for each stage of the computation, just so we can inspect them more easily. But if you don't need to do that, you can sequence them together with normal function composition; such as:

```
ghci> push 9 (push 8 (push 7 (new 3)))
BoundedStack {capacity = 0, elements = [9,8,7]}
ghci> push 9 $ push 8 $ push 7 $ new 3
BoundedStack {capacity = 0, elements = [9,8,7]}
ghci> (push 9 . push 8 . push 7) (new 3)
BoundedStack {capacity = 0, elements = [9,8,7]}
```

You should try to write pop and top. Here are some placeholders:

```
pop = error "pop : not implemented yet"
top = error "top : not implemented yet"
```

Error management

There is the `error` function that takes a string and causes the program to halt (with an exception) and displays the string. That's a really blunt tool.

We generally don't like error because it turns the function you're writing into a "partial function" – one that doesn't respond properly for every possible input. (The built-in head and `tail` are partial functions.)

If you do use `error`, it's a good idea to make sure the string is searchable within the code:

```
  error "BoundedStack: pop: not implemented"
```

vs

```
  error "not implemented"
```

or

```
  error "OOPS"
```

which will make those messages harder to find.

The `Maybe` type

This type is Haskell's answer to "null pointers". It's equivalent to this data type:

```
data Maybe a
  = Just a
  | Nothing
```

A function in the standard Haskell prelude that produces a maybe value is `lookup`. It searches through a list of key-value pairs for a matching key. This type of list is often called an **association list**.

```
friends :: [(String,Int)]
friends = [("Bob",1989), ("Alice",1993), ("Carla",1979)]
```

```
ghci> lookup "Alice" friends
Just 1993
ghci> lookup "Doug" friends
Nothing
```

What if we want to *act* on a value that has a `Maybe` type? You can use the `case/of` construct, which does pattern-matching on the result of an expression, much like we do when writing functions with multiple cases.

```
ageOf :: String -> [(String,Int)] -> Maybe Int
ageOf name people =
  case lookup name people of
    Nothing -> Nothing
    Just year -> Just (2017 - year)
```

So the above function performs the lookup of the name to retrieve a year, and then subtracts the year from 2017 to determine the person's age. But if the lookup fails, it handles that and also just returns `Nothing` for the person's age.

```
ghci> ageOf "Alice" friends
Just 24
ghci> ageOf "Bob" friends
Just 28
ghci> ageOf "Doug" friends
Nothing
```

The `Either` type

This is another way to signal errors or unusual conditions in Haskell.

```
data Either a b
  = Left a
  | Right b
```

Typically, the `Right` constructor is used as the correct result, and the `Left` constructor is used for some erroneous result. But we can also just use `Either` and the `Left`/`Right` constructors to unify two types into a common one. Recall that lists must contain elements of the same type, so it's an error to try to have a list containing both numbers and characters:

```
ghci> [5,7,8]
[5,7,8]
ghci> ['a','b','c']
"abc"
ghci> ['a',7,8,'c']
<interactive>:260:6: error:
    • No instance for (Num Char) arising from the literal ‘7’
```

One way around this is to construct a list of `Either` values, where we use `Left` to tag characters, or `Right` to tag numbers.

```
ghci> stuff = [Left 'a', Right 7, Right 8, Left 'c']
ghci> :t stuff
stuff :: Num b => [Either Char b]   -- List of either char or number
```

To use `Either` for errors, we can define a special-purpose version of `lookup` that produces a custom error message tagged with `Left` if the key is not found.

```
myLookup :: String -> [(String,a)] -> Either String a
myLookup key list =
  case lookup key list of
    Nothing -> Left ("key not found: " ++ key)
    Just value -> Right value
```

```
ghci> myLookup "Alice" friends
Right 1993
ghci> myLookup "Bob" friends
Right 1989
ghci> myLookup "Doug" friends
Left "key not found: Doug"
```

Here's an interesting function that keeps all the `Right` values in a list and throws away all the `Left` values.

```
rights :: [Either a b] -> [b]
rights [] = []
rights (Left _ : xs) = rights xs
rights (Right x : xs) = x : rights xs
```

Functors

In Haskell, `Functor` is a type class. That means it describes a set of types that have particular properties or functions. You can think of `Functor` as describing types that are **mappable**: Lists are mappable; `Maybe` is mappable; and the right side of `Either` is mappable.

```
ghci> :t map                        -- map is specific to lists
map :: (a -> b) -> [a] -> [b]
ghci> :t fmap                       -- fmap works for any functor
fmap :: Functor f => (a -> b) -> f a -> f b
ghci> map (+4) [1..5]
[5,6,7,8,9]
```

```
ghci> fmap (+4) [1..5]           -- on lists, fmap same as map
[5,6,7,8,9]
ghci> fmap (+4) (Just 5)         -- Maybe is a functor
Just 9
ghci> fmap (+4) Nothing
Nothing
ghci> fmap (+4) (Right 5)        -- Either is a functor,
Right 9
ghci> fmap (+4) (Left 5)         -- but only on a Right value
Left 5
```

Let's specify that Tree is a functor instance. Like `Either`, the `Tree` takes two type variables. To simplify, we'll demonstrate this using a new type `Tree1` that just takes one type variable. It won't store any data at the branches, just at the leaves.

```
data Tree1 a
  = Leaf1 a
  | Branch1 (Tree1 a) (Tree1 a)
  deriving Show

instance Functor Tree1 where
  fmap fn (Leaf1 a) = Leaf1 (fn a)
  fmap fn (Branch1 left right) =
    Branch1 (fmap fn left) (fmap fn right)

exampleTree :: Tree1 Int
exampleTree =
  Branch1 (Branch1 (Leaf1 5) (Leaf1 6))
          (Branch1 (Branch1 (Leaf1 7) (Leaf1 8))
                   (Leaf1 9))

ghci> fmap (*2) exampleTree
Branch1 (Branch1 (Leaf1 10) (Leaf1 12))
        (Branch1 (Branch1 (Leaf1 14) (Leaf1 16)) (Leaf1 18))
ghci> fmap show exampleTree
Branch1 (Branch1 (Leaf1 "5") (Leaf1 "6"))
        (Branch1 (Branch1 (Leaf1 "7") (Leaf1 "8")) (Leaf1 "9"))
ghci> fmap (const "Yow") exampleTree
Branch1 (Branch1 (Leaf1 "Yow") (Leaf1 "Yow"))
        (Branch1 (Branch1 (Leaf1 "Yow") (Leaf1 "Yow")) (Leaf1 "Yow"))
```

Main program

I'm not using this for anything, but if you compile this with `runghc`, it requires a `main`, so here it is.

```
main = return ()
```