Functional Programming in Haskell

1 Overview and Syntax

- Algebraic Datatypes and Pattern Matching

```haskell
data Example = C1 | C2 Int Int

val :: Example
val = C2 1312 420

first :: Example -> Maybe Int
first d = case d of
  C2 a b -> Just a,
  _      -> Nothing
```

- Polymorphism: lower case type e.g. `a -> a` where a is a type variable
- Type classes: predicates on types, used as constraints (like Traits/Interfaces)

```haskell
class Show a where
  show :: a -> Result

instance Show Example where
  show e = ...

showBoth :: (Show a, Show b) => a -> b -> Result
```

- Lazy Evaluation $\implies$ potential speedups, infinite lists are possible (see `foldr` vs `foldl` exercise)
- Purely functional, no side effects (instead, return descriptions of side effects and compose them using monads)
- Weird Syntax

```haskell
1 /= 0 == True    -- not !=
not False == True -- `not' written out but `&&' and `||' exist
[1,2,3] !! 0 == 1 -- indexing operator !!
"ab" == ['a','b'] -- strings are UTF8-Linked Lists
```

- Pattern guards in pattern matching: Give additional condition as boolean predicate $\implies$ use catch-all `otherwise` or _ to ensure exhaustive matching since predicate could be undecidable
- Curried functions are standard, uncurried provides tuple instead (less laziness?)
- Pattern match and destructure expressions anywhere

```haskell
addCurried = \(x,y) -> x + y -- destructure tuple in lambda args
empty (x:xs) = False         -- match on function argument
```

- like a Boeing whistleblower on the court date, there are no statements