Day 20.

1. Qualified Polymorphism

We have monomorphic things—like integer addition—and polymorphic things—like identity or list length. Does this allow for all the reuse we might want?

Not really. Consider the list element function.. something like:

```
elem _ []     = False
elem x (y:ys) = x == y || elem x ys
```

What sort of type would we want to give this function? On the one hand, we would like it to have some degree of polymorphism—we'd like to be able to apply `elem` to lists of integers, Booleans, pairs of integers, and so forth. On the other hand, we can't make it completely polymorphic—it takes a long time to compare functions for equality.

Here's another example:

```
insert x []  = [x]
insert x (y:ys)
 | x < y     = x:y:ys
 | otherwise = y : insert x ys
```

Again, this works well for integers and Booleans, and very poorly for functions. There are also values that we can easily compare for equality but are less easily ordered—sets of values, for example.

Our approach to this in languages like Haskell is called *qualified polymorphism*. Qualified polymorphism supports additional restriction on the instantiation of type variables. For example, we can express the type of `elem`

```
elem :: Eq a ⇒ a → [a] → Bool
```

What does `Eq a` mean? Intuitively, it means that `a` should only be instantiated with types that support equality—so we want these types:

```
elem :: Int → [Int] → Bool
elem :: Bool → [Bool] → Bool
```

but not

```
elem :: (Int → Int) → [Int → Int] → Bool
```

What does it mean for a type to support equality? Concretely, it means that there's an equality operator for these types: we have

```
(==) :: Int → Int → Bool
(==) :: Bool → Bool → Bool
```

but not

```
(==) :: (Int → Int) → (Int → Int) → Bool
```

Of course, we can identify some categories of types like this—equality, ordering, text representations, &c. But in general, programmers might need to add their own categories. In Haskell, we call these *type classes*. Classes are defined and instantiated in user code. For example, we can define the `Eq` class like this:

```
class Eq t
  where (==) :: t → t → Bool
```

And we could add *instances* to it—that is, assert that types belong to the `Eq` class like this:

```
instance Eq Bool
  where b1 == b2 = not (b1 `xor` b2)

instance Eq t ⇒ Eq [t]
  where []     == []     = True
        (x:xs) == (y:ys) = x == y && xs == ys
        _      == _      = False
```

There's a whole unhelpful tradition in making analogies about type classes: "type classes are like X but...", where X is most commonly interfaces. Unpacking the "but" is usually more work than just understanding type classes directly. For example, because Haskell doesn't have subtyping, any discussion about the relationship between type classes and interfaces is going to be more about the difference between parametric and subtype polymorphism than anything else. A type class is a set of types, defined by common operations; that's enough for now.

2. Monads and Effects

There's one particular set of common operations that's kept turning up this semester. If we think back to the first interpreter I wrote in class, we had a couple of operations to chain together operations that could fail:

```
done :: a → Maybe a
done x = Just x

andThen :: Maybe a → (a → Maybe b) → Maybe b
Just x  `andThen` f = f x
Nothing `andThen` f = Nothing
```

We weren't happy with just tracking failing operations tho, we also wanted to add some logging:

```
done :: a → ([String], a)
done x = ([], x)

andThen :: ([String], a) → (a → ([String], b)) → ([String], b)
(ws, x) `andThen` f = (ws ++ ws', y)
  where (ws', y) = f x
```

In the meantime, we've implemented some effects. For example, we talked about state, in which we interpreted commands as state transformers. We can fit this into a similar pattern:

```
done :: a → s → (a, s)
```

```
done x s = (x, s)

andThen :: (s → (a, s)) → (a → s → (b, s)) → s → (b, s)
st1 `andThen` k = λs → let (x, s') = st1 s in f x s'
```

We've even talked about non-determinism. We can represent non-determinism by computing a list of values (like a simple version of the distributions you're building in homework 4)

```
done :: a → [a]
done x = [x]

andThen :: [a] → (a → [b]) → [b]
xs `andThen` f = concat (map f xs)
```

So we have a series of types—or, more correctly, *type constructors*—with common operations. This is a great candidate for a type class. Let's imagine what it might look like:

```
class Effect e
  where done :: a → e a
        andThen :: e a → (a → e b) → e b
```

We can add some instances of this class.

```
instance Effect Maybe
  where done x              = Just x
        Just x  `andThen` f = f x
        Nothing `andThen` f = Nothing

instance Effect ((,) [String])
  where done x              = ([], x)
        (ws, x) `andThen` f = (ws ++ ws', y)
          where (ws', y) = f x

instance Effect []
  where done x         = [x]
        xs `andThen` f = concat (map f xs)
```

In fact, this idea is useful enough that it's a standard part of Haskell.

```
class Monad m
  where return :: a → m a
        (>>=)  :: m a → (a → m b) → m b
```

Type constructors like this are called *monads*—or triples, or Kleisli triples—and were proposed as a mathematical tool to model effects in impure programming languages. Later, people realized that they could be used to implement effects in pure languages. They're so central to Haskell that they get special syntax—`do` notation. Again, there's a cottage industry in explaining monads. The important point is that you don't need them explained—you've already invented them.