1 Type Classes

1.1 Using Type Classes

To motivate this, we will build a small library for *Environments* mapping keys `k` to values `v`. Recall that, in Nano, we represented environments as `[(Id, Value)]`; however, what if we want to represent keys that are not `Id` or values that are not `Value`?

Let us define a new **polymorphic datatype** `Env`.

```
data Env k v
    = Def   v                       -- Default value to be used for missing keys
    | Bind  k v (Env k v)           -- Bind key 'k' to value 'v'
    deriving (Show)
```

So, for example,

```
$ let env0 = add "cat" 10.0 (add "dog" 20.0 (Def 0))

$ get "cat" env0
10

$ get "dog" env0
20

$ get "horse" env0
0
```

Let us implement some of the key functions.

- `add`: Adds a `key` and `val` pair, returning a new environment.

  ```
  add :: k -> v -> Env k v -> Env k v
  add key val env = Bind key val env
  -- or
  -- add = Bind
  ```

- `get`: Gets the value associated with the `key`.

  ```
  get :: k -> Env k v -> v
  get key (Def v)         = v
  get key (Bind k v env)
      | k == key      = v
      | otherwise     = get key env
  ```

Note that this gives a type error, especially for `get`. The issue is that we require `k` and `key` to have `Eq` snce we're comparing two keys. So,

```
get :: Eq k => k -> Env k v -> v    -- Changed this line
get key (Def v)         = v
get key (Bind k v env)
    | k == key      = v
    | otherwise     = get key env
```

1.2 Explicit Type Annotations

Consider the standard typeclass `Read`, where its simplified implementation is shown below:

```
class Read a where
    read :: String -> a
```

Note that `Read` is the *opposite* of `Show`.

- It requires that every instance `T` can parse a string and turn it into `T`.
- Just like with `Show`, most standard types are instances of `Read`.

(Quiz.) What does the expression `read "2"` evaluate to?

(a) Type error

(b) `"2"`

(c) `2`

(d) `2.0`

(e) Run-time error

The answer is **A**. There are multiple ways to "read" `"2"`. In general, note that the definition of `read` has that the return type is `a`, a generic type.

So, **explicit type annotation** is needed to tell Haskell what to convert the string to.

```
$ (read "2") :: Int
2

$ (read "2") :: Float
2.0

$ (read "2") :: String
**** Exception: Prelude.read: no parse

$ (read "\"2\"") :: String
"2"

$ read "()"
()
```

1.3 Creating Type Classes

Type classes are useful for many different things. Let's see an example with **JSON**. Here's an example JSON:

```
{ "name"    : "Nadia"
, "age"     : 37.0
, "likes"   : [ "poke", "coffee", "pasta" ]
, "hates"   : [ "beets" , "milk" ]
, "lunches" : [ {"day" : "mon", "loc" : "rubios"}
            , {"day" : "tue", "loc" : "home"}
            , {"day" : "wed", "loc" : "curry up now"}
```

```
                , {"day" : "thu", "loc" : "home"}
                , {"day" : "fri", "loc" : "santorini"} ]
}
```

Each JSON value is either

- a base value like a string, number, or boolean,
- an (ordered) array of values, or
- an object, i.e. a set of string-value pairs.

1.3.1 JSON Datatype

We can represent a subset of JSON values with the Haskell data type

```
data JVal
    = JStr  String
    | JNum  Double
    | JBool Bool
    | JObj  [(String, JVal)]
    | JArr  [JVal]
    deriving (Eq, Ord, Show)
```

So, the example JSON would look like

```
js1 =
    JObj [("name", JStr "Nadia")
        ,("age",  JNum 36.0)
        ,("likes",   JArr [ JStr "poke", JStr "coffee", JStr "pasta"])
        ,("hates",   JArr [ JStr "beets", JStr "milk"])
        ,("lunches", JArr [ JObj [("day",  JStr "mon")
                                 ,("loc",  JStr "rubios")]
                          , JObj [("day",  JStr "tue")
                                 ,("loc",  JStr "home")]
                          , JObj [("day",  JStr "wed")
                                 ,("loc",  JStr "curry up now")]
                          , JObj [("day",  JStr "thu")
                                 ,("loc",  JStr "home")]
                          , JObj [("day",  JStr "fri")
                                 ,("loc",  JStr "santorini")]
                          ])
        ]
```

This is a pain to write out. Instead, let us serialize Haskell Values to JSON.

- Base types `String`, `Double`, `Bool` are serialized as base JSON values.
- Lists are serialized into JSON arrays.
- Lists of key-value pairs are serialized into JSON objects.

1.3.2 Type Classes

We can define a type class

```
class JSON a where
    toJson :: a -> JVal
```

so that a type `a` can be converted to JSON. Then, we can work on the basic types:

```
instance JSON Double where
    toJson = JNum

instance JSON Bool where
    toJson = JBool

instance JSON String where
    toJson = JStr
```

We can also work on more complicated types.

```
instance JSON a => JSON [a] where
    toJson xs = JArr [toJson x | x <- xs]
```

Here, if `a` is an instance of `JSON`, then there is a generic recipe to convert lists of `a` values. Similarly, for key-value lists, we have:

```
instance (JSON a) => JSON [(String a)] where
    toJson kvs = JObj [ (k, toJson v) | (k, v) <- kvs]
```