"Pascal [Java] is for building pyramids – imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp [Haskell] is for building organisms – imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place.

...

the pyramid must stand unchanged for a millennium; the organism must evolve or perish."

– *Alan J. Perlis, Foreword to "Structure and Interpretation of Computer Programs", 1985, 1996*

Review

- `type` defines a type name as an abbreviation for other types
- `data` defines new new data structures (and a type) and constructors / deconstuctors
- `IO t` is the input/output monad
- `do` can be used to sequence input/output operations

Input-Output (IOAdder.hs, IOAdder2.hs)

- `IO t` is a type for input and output
- `type IO t = World -> (t,World)`
- See `putChar`, `getChar`, `getLine`, `putStr`
- These are normal functions:

  ```
  ask_polite q = ask (q++" please ")
  ```

- These can be sequenced using `do`.

  ```
  do v1 <- a1
     v2 <- a2
     ...
     vn <- an
     return (f v1 ... vn)
  ```

  Each *ai* is of type `IO ti` for some type *ti*

  vi is of type *ti*

  ai gets world from a_{i-1}, gives value to v_i and world to a_{i+1}
- When called from prompt, a_1 gets world from system.
- Type of do expression is `IO t` where t is return type of f

Putting types into classes (BSTree2.hs)

- `Show` is the class that contains the function:

```
show :: Show a => a -> String
```

- `Read` is the class that contains the function:

```
read :: Read a => String -> a
```

- To get a default implementation of `show` and `read`, we can do:

```
data BSTree k v = Empty
                | Node k v (BSTree k v) (BSTree k v)
          deriving (Show, Read)
```

- Most predefined types – except for functions — are in Show and Read.

Putting types into classes

- A class defines the set of functions defined for types in the class.
- You can see the functions for a class by doing:

  `:info Classname`

  in ghci.
- To put a type into a class do:

```
instance  Class Type where
    <define the minimal functions for the class>
```

- See Instanceeg.hs

Putting types into classes (BSTree2.hs)

- `Eq` is the class that contains the functions

```
(/=) :: Eq a => a -> a -> Bool
(==) :: Eq a => a -> a -> Bool
```

- If we don't want the default definitions we can declare `(BSTree k v)` to be an instance of the `Eq` class:

```
instance (Eq k,Eq v) => Eq (BSTree k v) where
   t1 == t2 = tolist t1 == tolist t2
```

  as long as `k` and `v` are in the `Eq` class.

- `fmap` is a generalization of `map` defined for any type in the `Functor` class. We can define the `fmap` on the values by:

```
instance Functor (BSTree k) where
    -- fmap :: (a -> b) -> BSTree k a -> BSTree k b
    fmap f Empty = Empty
    fmap f (Node key val t1 t2)
         = Node key (f val) (fmap f t1) (fmap f t2)
```

Putting types into classes (BSTree2.hs)

Putting BSTree into `foldable class`:

```
instance Foldable  (BSTree k) where
  foldr op base tree
      = foldr op base [v | (k,v) <- (tolist tree)]
```

This automatically defines: sum, foldl, null, length ...

```
:info Foldable
```

Using a default defintion of Foldable:

```
data BSTree k v = Empty
                | Node k v (BSTree k v) (BSTree k v)
                deriving (Show, Read, Foldable)
```

How to write a (Haskell) Program

- To solve a complex problem, break it into simpler problems.
- Motivate/design top-down
- Build bottom-up.
- Write a clear specification (API / intended interpretation) for each component; program to that specification.
- Ensure each part is modular, debuggable, with clear meaning.
- Test early and test often. Try to break your program. Try to prove your program is correct.
- Test every function when defined. Every component of a function should be already tested and debugged before use.
- Give the type signature for every function (so the compiler does not suggest bugs in tested code).
- Generalize components so they are as reusable as possible. Make sure you can find a previously written appropriate function when it is the one you want.
- Write functional components as much as possible. Try to minimize IO components.