"I think that it's extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free perfect use of these machines. I don't think we are. I think we're responsible for stretching them, setting them off in new directions and keeping fun in the house. I hope the field of computer science never loses its sense of fun."

– Alan J. Perlis, 1977 (quoted in dedication to "Structure and Interpretation of Computer Programs", 1985)

Review

- Haskell is a functional programming language
- Strongly typed, but with type inference

  Bool
  Num, Int, Integer, Fractional, Floating, Double
  Eq, Ord
  Tuple, List, Function
- Classes, type variables
- List comprehension $[f\ \ x \mid x\texttt{<-}list,\ \ cond\ \ x]$
- $foldr\ \ \oplus\ \ v\ [a1, a2, ..an] = a1 \oplus (a2 \oplus (\ldots \oplus (an \oplus v)))$
- $foldl\ \ \oplus\ \ v\ [a1, a2, ..an] = (((v \oplus a1) \oplus a2) \oplus \ldots) \oplus an$
- reduction
- call-by-value, call-by-name, lazy evaluation
- type, data

Type and data

- data defines new data structures (and a type)
  Example:

  ```
  data FValue =  BooleanF Bool
               | NumberF Int
               | StringF [Char]
               | MissingF
  ```

  defines
  - a new type: *FValue*
  - 4 new constructors: *BooleanF*, *NumberF*, *StringF*, *MissingF*.
- The constructors have dual roles:
  - constructors with arguments give functions that can create a new value of that type
  - constructors with no arguments are constants
  - constructors can be used patterns for deconstructing the type (accessors) on left side of = or in function definitions.

Clicker Question

Suppose the function

```
myfun Toves = 7
myfun (Slithy x) = x+2
```

has inferred type

```
myfun :: Gyre -> Integer
```

and compiles and never gives a runtime error. Which is not true:

A `Gyre` is a type

B `Slithy` is a function of type `Integer -> Gyre`

C `Toves` is a constant of type `Gyre`

D There must be a declaration

```
data Gyre = Toves
          | Slithy Integer
```

E One of the above is false.
(So if all A-D are true, this is the answer.)

Recursive data types (BSTree2.hs)

- data definitions can be recursive: Example:

  ```
  -- a binary search tree that maps integers to strings
  data BSTree = Empty
              | Node Int String BSTree BSTree
  ```

  defines a new type, *BSTree*, and two new constructors *Empty* and *Node*.
- The constructors can be used as
  - constants (Empty) or functions (Node) to create a BSTree
  - patterns for deconstructing the type
- The data structures can be parametrized by types:

  ```
  -- a binary search tree
  --   k is the key type; v is the value type
  data BSTree k v = Empty
                  | Node k v (BSTree k v) (BSTree k v)
  ```

data (BSTree2.hs)

- A binary search tree:

```
-- a binary search tree
--   k is the key type; v is the value type
data BSTree k v = Empty
                | Node k v (BSTree k v) (BSTree k v)
```

- What should `lookup key tree` return?
  What if the key isn't in the tree?
- How can `insert key value tree` also return the old value of key?
  What if there wasn't an old value?

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 (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 also defines: sum, foldl, null, length....
See

```
:info Foldable
```

Using a default defintion:

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