Square Root (example)

Consider the following square-root function that generates an error if its argument is negative (see ROOT.hs):

```
root x
 = if x < 0
    then fail ("negative! "++show x)
    else return $ sqrt x
```

It is monadic code, because it uses `fail` and `return`
Which monad? `IO`, `Maybe`, `[]`, ?
Let's ask:

```
> :i root
root :: (Floating r, Monad m, Ord r, Show r) => r -> m r
```

It should work in any monad!

Running root

```
> root 4 :: Maybe Double
Just 2.0


> root 4 :: [Double]
[2.0]


> root 4  -- in GHCi the IO monad is used as the default
2.0       -- and the return value is dislayed
```

It works!

Running root

```
> root (-1) :: Maybe Double
Nothing


> root (-1) :: [Double]
[]


> root (-1)  :: IO Double
*** Exception: user error (negative! -1.0)
```

It does errors too
but with the IO monad, fail gives a *runtime* error

Sum of two roots (example)

Now, we define `sum2roots` in terms of `root`

```
sum2roots x y  =  do rx <- root x
                     ry <- root y
                     return (rx + ry)
```

Monadic as well, as we use the do-notation.

```
> sum2roots 4 9 :: Maybe Double
Just 5.0


> sum2roots (4) (-9) :: [Double]
[]


> sum2roots (4) (-9)   -- IO Monad again
*** Exception: user error (negative! -9.0)
```

All works as expected here as well.

Error messages only in `IO`

- In the above examples, only `IO` actually showed the error message
- Can we see the errors in non-IO, i.e. regular function code?
- We might, using the `Either` type constructor:

```
data Either a b = Left a | Right b
```

- Instead of `Maybe a`, we could use `Either String a`, so an error would return as `Left "..error message.."`
- `Either` has a `Monad` instance:

```
instance Monad (Either a) where
  return x                   =  Right x
  (Left errstr) >>= _    =  Left errstr
  (Right x)     >>= f    =  f x
```

Running `sum2roots` in `Either`

```
> sum2roots (4) (9) :: Either String Double
Right 5.0
> sum2roots (4) (-9) :: Either String Double
*** Exception: negative! -9.0
```

!! The correct answer is fine (uses `Right`), but the error is a Haskell exception, and not a `Left` value.
Look at this:

```
 > sum2roots (4) (9) :: Either a Double
Right 5.0
> sum2roots (4) (-9) :: Either a Double
*** Exception: negative! -9.0
```

We have an arbitrary type `a`, but we still get the same effect!
What's going on?

Default `fail` needed in generic `Either` monad

- The Prelude `Monad` instance is for `Either e`, where `e` is an arbitrary type
- So what can `fail errmsg` do? It cannot return `Left errmsg`, becuse `e` may not itself be `String`
- The `fail` method is not mandatory, and it defaults to

```
fail errmsg =  error errmsg
```

So we get a runtime error, instead of a left value

Wanted: `Monad` instance for `Either String`

- We could try to define an instance in our code, with

```
fail errmsg = Left errsmg
```

- We hit two problems:
  - It clashes with (overlaps with) the Prelude instance.
    We might used import trickery to hide the Prelude instance,
  - Even so,we still get an error:
    class instances for type-constructors that are partially applied (e.g. `Either e`) have a restriction that the arguments can only be type variables, and not concrete types.
- The solution is to use `newtype` to clone `Either String`, and then give it an instance

Cloning and Instancing Either String

-
  ```
  newtype ES a = ES (Either String a)
  ```

-
  ```
  instance Monad ES where
    return x                  =  ES $ Right x
    (ES (Left errstr)) >>= _  =  ES $ Left errstr
    (ES (Right x))     >>= f  =  f x
    fail errstr               =  ES $ Left errstr
  ```

  We have do a lot of wrap/unwrap of data-constructor ES, but that's the edit/compile time cost of newtype
  (no runtime impact at all).
- However, all that wrap/unwrap overhead is buried in the monad instance above, so it doesn't appear in the code for either root or sum2roots.

ES Monad instance, with annotations

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  fail   :: String -> m a


instance Monad ES where

  return (x :: a)                 =  ES (Right (x :: a)) :: ES a

  (ES (Left errstr) :: ES a) >>= _
                                     =  ES (Left errstr) :: ES b

  (ES (Right (x :: a))) :: ES a >>= (f :: a -> ES b)
                                                      =  f x :: ES b

  fail (errstr :: String)            =  ES (Left errstr) :: ES a
```

Running ES

Our new monad instance works just fine:

```
> sum2roots (4) (9) :: ES Double
ES (Right 5.0)
> sum2roots (4) (-9) :: ES Double
ES (Left "negative! -9.0")
> sum2roots (-4) (-9) :: ES Double
ES (Left "negative! -4.0")
```

Monad is very generic

When we write code 'monadic-style', using only the Monad class functions then we get highly-reusable code that works with any Monad instance.
This versatility of the class system gives Haskell a lot of its most powerful abstractions.
There is a library of operations Control.Monad which specifies many operations in a generic fashion so that they work in any monad

Monadic Table Lookup

```
type Dict k d = [ (k, d) ]

mfind :: (Show k, Ord k, Monad m) => Dict k d -> k -> m d
mfind [] key = fail ( "mfind - Couldn't find: " ++ show key )
mfind ((k,v):kds) key
 | key == k     =  return v
 | otherwise  =  mfind kds key
```

Monadic Evaluation (eval, 1)

```
eval :: Monad m => Dict Id Double -> Expr -> m Double

eval _ (Val x) = return x
eval d (Var i) = mfind d i

eval d (Add x y) = evalOp d (+) x y
eval d (Mul x y) = evalOp d (*) x y
eval d (Sub x y) = evalOp d (-) x y
```

More to come . . .

Monadic Operator Evaluation

```
evalOp :: Monad m
       => Dict Id Double -> (Double -> Double -> Double )
       -> Expr -> Expr -> m Double

evalOp d op x y
 = do a <- eval d x
      b <- eval d y
      return (a `op` b)
```

Monadic Evaluation (eval, 2)

```
-- eval :: Monad m => Dict Id Doubel -> Expr -> m Double

eval d (Dvd x y)
 = do b <- eval d y
      if b == 0.0
       then fail "divide by zero"
       else do a <- eval d x
               return (a/b)

eval d (Def x e1 e2)
 = do v <- eval d e1
      eval (define x v d) e2
```

A better dictionary

A lookup table as a list of pairs is very inefficient. Potentially, we have to search through the entire table in order to find the one we want. If we assume that the elements can be kept in some kind of order then the definition can be improved.
A *binary search tree* (`Tree`) will improve efficiency. We will want to have empty leaves, and a branch that stores node-data (typically a key/value pair) along with two subtrees.

```
data Tree nd
  = Leaf
  | Branch (Tree nd) nd (Tree nd)
```

Using a Binary Tree as a Dictionary

We take the node type (`nd` in previous slides) to be a key-data pair, where the key is a string and we have integer data.

```
type TDict = Tree (String, Int)
```

If all we have is a `Leaf`, then we build a `Branch` with empty sub-trees:

```
insert Leaf key value = Branch Leaf (key, value) Leaf
```

Otherwise we compare keys to decide where the insert should occur:

```
insert (Branch l (k, v) r) key value
    | key < k   =  Branch (insert l key value) (k, v) r
    | key > k   =  Branch l (k, v) (insert r key value)
    | key == k  =  Branch l (k, value) r
```

Looking up Binary Tree

A `Leaf` has no key-data contents:

```
lookup Leaf _ = Nothing
```

With a `Branch` we compare keys to see where to search

```
lookup (Branch l (k, v) r) key
  | key < k   =  lookup l key
  | key == k  =  Just v
  | key > k   =  lookup r key
```

Exercise: rewrite this monadic style !

Map and Fold, or: Iteration in Functional Style

- We have seen some built-in and user-defined datatypes (e.g. Lists (`[t]`), `Maybe`), and examples of function definitions that use them.
- We are now going to take a more systematic look at the relationship between datatypes and code.
  - Analagous to the classic text:
    Wirth, Niklaus,*Algorithms + Data Structures = Programs*, Prentice-Hall, ISBN 978-0-13-022418-7, 1976.

Haskell lists reconsidered

- In Haskell, the type "list of type t", is written `[t]`.
  - So, in a type-expression, `[_]` is a *type-constructor*: a type-valued function taking a type as argument and returning a type as result.
- A "list of t" (`[t]`) has two forms:
  1. "empty", or "nil", written `[]`
  2. a non-empty "cons" node with two sub-components, the first (`x`) of type `t`, the second (`xs`) of type `[t]`, written `x:xs`
- The notations `[]` and `:` are *data-constructors* for the type `[t]`.
  That is, they are both *functions* from zero or more arguments to results of type `[t]`.
  - `[]` is a data-constructor taking no argument: `[] ::  [t]`.
  - `:` is a data-constructor taking two arguments:
    `(:)  ::  t -> [t] -> [t]`

Structure vs. Contents

- Consider the list `[4,2,7]`, or written without syntactic sugar:
  `4:(2:(7:[]))`
- It has a *structure*, defined by the pattern of data-constructors used:
  `4:(2:(7:[]))`
- It has *contents*, defined by data values supplied as arguments to data-constructors:
  `4:(2:(7:[]))`
- We can define two key functions: one that changes contents whilst leaving structure unchanged, whilst the other does the opposite.

Map: replace content

- We can define a function that allows us to systematically transform content, whilst leaving the structure intact: `map`
- A map function can be defined for every `data`-type we have or define.
- It typically takes two arguments:
  1. A function that transforms the content
  2. an instance of the datatype
- It returns an instance of the datatype with the structure unchanged, but the **contents** replaced by the result of applying the function

Map for Haskell lists

- `map` takes a function changing list elements (values) and a list as arguments, and returns the modified list as result:
  `map :: (s -> t) -> [s] -> [t]`
- With the empty list, all we have is structure, so there is no change
  `map f [] = []`
- With a cons-node, we can modify the first value, and recurse to do the rest:
  `map f (x:xs) = (f x):(map f xs)`
- Example
  `map (+1) (4:(2:(7:[]))) =  5:(3:(8:[]))`

Fold: replace content

- We can define a function that allows us to systematically transform structure, whilst leaving the values “almost intact”: `fold`
- A fold function can be defined for every `data`-type we have or define.
- It typically takes the following arguments:
  1. Functions/values to replace data-constructors (structure)
- It returns an instance of the datatype with the **data constructors** replaced by the corresponding functions.
- The resulting expression has contents unchanged initially (thanks to laziness), but once evaluated, everything will collapse (“fold”) into some final value

Fold for Haskell lists

- `fold` takes functions replacing `[]` and `:` and a list as arguments, and returns a matching expression structure as a result

  ```
  fold :: t -> (s -> t -> t) -> [s] -> t
  ```
- With the empty list, we replace the zero-argument constructor with the (nullary) replacement function (value):

  ```
  fold z op [] = z
  ```
- With a cons-node, we replace the two-argument constructor with the two-argument replacement function, and recurse:

  ```
  fold z op (x:xs) = x ‘op‘ (fold z op xs)
  ```
- Example

  ```
  fold 0 (+) (4:(2:(7:[])) ) =  4+(2+(7+0))
  ```

Map is a Fold

- We can implement `map` as a `fold`
- (Do live in class?)

Solution:

```
mapAsFold :: (a -> b) -> [a] -> [b]
mapAsFold f = fold [] (mop f)

mop :: (a -> b) -> a -> [b] -> [b]
mop f x ys = f x : ys
```

Tree Map

Remember — we leave the structure alone and modify the content
Type Signature:

```
mapTree :: (a -> b)   -- function to change content
        -> Tree a     -- input tree
        -> Tree b     -- result tree
```

Base-Case (`Leaf`):

```
mapTree f Leaf = Leaf
```

Recursive Case (`Branch`):

```
mapTree f (Branch ltree n rtree)
 = Branch (mapTree f ltree)
          (f n)
          (mapTree f rtree)
```

Tree Fold

Remember — we leave the content alone and modify the structure

```
Leaf :: Tree nd
Branch :: Tree nd -> nd -> Tree nd -> Tree nd
```

Type Signature:

```
foldTree :: a                       -- replacement for Leaf
         -> (a -> nd -> a -> a) -- replacement for Branch
         -> Tree nd             -- input tree
         -> a                   -- final value
```

First constructor (`Leaf`):

```
foldTree zero f Leaf = zero
```

Second constructor (`Branch`):

```
foldTree zero f (Branch ltree n rtree)
 = f (foldTree zero f ltree)
     n
     (foldTree zero f rtree)
```