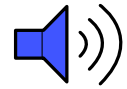
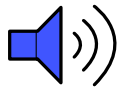
Lecture 11 – File IO : find files 

Outline:

- Review of homework 6
  Use state monad to thread a mutable state in recursion.
- File IO
  Study file IO with an example of finding files. This lecture corresponds to Chapter 9 of *Real World Haskell*.
- Handle IO Exceptions
  Exceptions in file IO must be handled or risking subtle bugs.
- Predicate
  Use type class to define a mini-DSL for file query predicates.
- Find files with filter predicates
  Recursion with IO monads.

Random numbers 

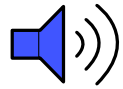
- A long long time ago, you submitted homework 6, which is to define a function generate a Vec of random noises given a seed, length, and a lower and an upper bound of the noise.

```
makeNoise :: Random a => Int     -- seed
                      -> Int     -- length
                      -> a       -- lower bound
                      -> a       -- upper bound
                      -> Vec a
```

- You should use State monad to implement this function.

```
makeNoise seed n low high = Vec
 $ evalState randomSeq -- run state monad to get [a]
 $ mkStdGen seed       -- random number generator

  where randomSeq = sequenceA -- state monad :: State g [a]
        $ take n              -- size n list :: [State g a]
        $ repeat              -- inf list    :: [State g a]
        $ state               -- state monad :: State g a
        $ randomR (low, high) -- g -> (a, g)
```

From random function to State monad



- randomR returns a function
  `a generator -> (a random number, a new generator)`

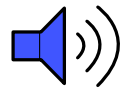
```
randomR (low, high)
  :: (Random a, RandomGen g) => g -> (a, g)
```

- We can convert the returned function into a state monad

```
state $ randomR (low, high)
  :: (Random a, RandomGen g) => State g a
```

- This is because `state` can be defined as:

```
state :: (s -> (a, s)) -> State a
```

Run a list of State monads 

- Make a finite list of state monads

```
take n $ repeat $ state $ randomR (low, high)
  :: (Random a, RandomGen g) => [State g a]
```

- We can use sequenceA to run state monads one by one, passing output state of one monad to input state of the next.

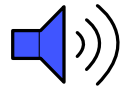
```
sequenceA $ take n $ repeat $ state $ randomR (low, high)
  :: (Random a, RandomGen g) => State g [a]
```

- This is because sequenceA can be defined as:

```
sequenceA ::
  (Traversable t, Applicative f) => t (f a) -> f (t a)
```

  In this case, [] is the Traversable t while State is the Applicative f. Specifically, sequenceA uses liftA2 (:) to cons (State g a) with (State g [a]) and return (State g [a]).

How does sequenceA work?



- In our context, `sequenceA` can be defined as:

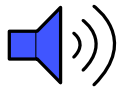
```
sequenceA :: [State s a] -> State s [a]

sequenceA [] = pure []
sequenceA (a:b) = liftA2 (:) a $ sequenceA b
```

- Since State is a monad, we can think of liftA2 as:

```
liftA2 f m1 m2 = do a1 <- m1
                    a2 <- m2
                    return $ f a1 a2
```

This allows the output state of m1 to be passed to the input state of m2 during the use of liftA2 (:).

File IO

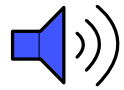


- Modules of interests

```haskell
import System.Directory
 -- actions on directories and files
 -- existence test
 -- permissions
 -- timestamps

import System.FilePath
 -- file path manipulation

import System.IO
 -- open/close/read/write/append file
```

- This lecture focuses on finding files.

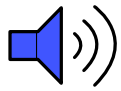
```haskell
import System.Directory (doesDirectoryExist, listDirectory,
                         getModificationTime, getFileSize,
                         getPermissions, Permissions(..))

import System.FilePath ((</>), takeExtensions)
import Data.Time (UTCTime(..))
import Control.Exception (handle, SomeException(..))
import Control.Monad (forM, mapM_, filterM, guard)
```

Find file path recursively 

- Take a directory path and returns all file paths in the directory (including the directory itself)

```haskell
type FilePath = String                          -- built-in type alias

getRecursiveContents :: FilePath -> IO [FilePath]

getRecursiveContents dir = do
  names <- listDirectory dir                          -- IO [FilePath]

  paths <- forM names $ -- forM :: [a] -> (a->m b) -> m [b]
   \name -> do

     let path = dir </> name                -- </> makes "dir/name"
     isDirectory <- doesDirectoryExist path
     if isDirectory
       then getRecursiveContents path          -- IO [FilePath]
       else return [path]                      -- IO [FilePath]

  return $ dir : (concat paths)                -- IO [FilePath]
```

About forM



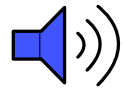
- forM is defined for Traversable types and is a flipped mapM. Both functions run a traversable of monads, accumulate their side effects, and return a monad of the traversable.

```
1 forM::(Traversable t,Monad m) => t a -> (a -> m b) -> m (t b)
2
3 mapM::(Traversable t,Monad m) => (a -> m b) -> t a -> m (t b)
```

- If we only care about the side effects in the traversable (e.g. print), we can use the following two version of the function.

```
1 forM_ :: (Foldable t, Monad m) => t a -> (a -> m b) -> m ()
2
3 mapM_ :: (Foldable t, Monad m) => (a -> m b) -> t a -> m ()
```

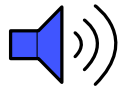
Find file path recursively



- `forM` below takes `[FilePath]` and a function `FilePath -> IO [FilePath]` and returns `IO [[FilePath]]`.

```
1   names <- listDirectory dir      -- names :: [FilePath]
2
3   let f = \name -> do -- f :: FilePath -> IO [FilePath]
4       let path = dir </> name
5       isDirectory <- doesDirectoryExist path
6       if isDirectory
7         then getRecursiveContents path
8         else return [path]
9
10  paths <- forM names f          -- paths :: [[FilePath]]
```

Find file with simple filter

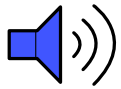


- With getRecursiveContents function, we can find files that satisfy simple filtering predicate.

```
1  simpleFind :: (FilePath -> Bool)  -- filter predicate
2             -> FilePath            -- directory path
3             -> IO [FilePath]       -- resulting paths
4
5  simpleFind p path = do
6    names <- getRecursiveContents path
7
8    -- filter :: (a->Bool) -> [a] -> [a]
9    return $ filter p names
10
11 -- find pdf files in the current directory
12 main = simpleFind (\x -> takeExtension x == ".pdf") "."
13        >>= (mapM_ putStrLn)  -- print the paths
```

Complex filters may be based on file size, permissions, modification time, extensions, path names, and their logical combinations.

Get more file information



- First define a FileInfo type to hold file information.

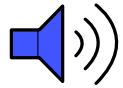
```
data FileInfo = FileInfo {
                      filePath          :: FilePath,
                      filePermissions   :: Maybe Permissions,
                      fileSize          :: Maybe Integer,
                      fileLastModified  :: Maybe UTCTime
                    }
```

  We may not get permissions, size, or last modified time for all files – thus Maybe types.

- Permissions contains some Boolean flags.

```
data Permissions = Permissions {readable   :: Bool,
                                writable   :: Bool,
                                executable :: Bool,
                                searchable :: Bool}
```

Get more file information



- Handle exceptions in IO actions and return Maybe type.

```haskell
maybeIO :: IO a -> IO (Maybe a)

maybeIO act = handle (\(SomeException _) -> return Nothing)
                     (fmap Just act)

handle :: Exception e => (e -> IO a) -> IO a -> IO a
```

- Retrieve FileInfo from a path

```haskell
getInfo :: FilePath -> IO FileInfo

getInfo path = do
  perms    <- maybeIO (getPermissions path)
  size     <- maybeIO (getFileSize path)
  modified <- maybeIO (getModificationTime path)

  return $ FileInfo path perms size modified
```