module L4 where

import Data.Char
import System.Random
IO
Questions about yesterday?

Recall that:

- A **Functor** is something that we can `fmap` over.
- An **Applicative** is a **Functor** plus `pure` and `<*>` called `sequence`.
- A **Monad** is an **Applicative** plus `>>=` called `bind`.

Examples: Maybe, `[]` and IO are monads!

Another Monad function we saw:

\[(>>>) :: (>>>) :: \text{Monad } m \Rightarrow m a \rightarrow m b \rightarrow m b\]
History of the IO Monad

Since 1992 the most famous Monad is IO.
https://youtu.be/re96UgMk6GQ?t=31m20s
(watch this from 31:20 until 38:22)
“Most of you use languages that were invented, and you can tell, can’t you. This is my invitation to you to use programming languages that are discovered.”

Philip Wadler: Propositions as Types
Here are some useful standard IO functions:

λ> :t readFile
readFile :: FilePath -> IO String

λ> :t writeFile
writeFile :: FilePath -> String -> IO ()

λ> :t getLine
getLine :: IO String

λ> :t putStr
putStr :: String -> IO ()

λ> :t putStrLn
putStrLn :: String -> IO ()

(* This is also the title of the Haskell book by Bryan O'Sullivan, Don Stewart, and John Goerzen. See http://book.realworldhaskell.org/.)
dialogue :: IO ()
dialogue = do
  putStrLn "Hello! Who are you?"
  name <- getLine
  putStrLn $ "Nice to meet you, " ++ name ++ "!"
  let capName = map toUpper name
  putStrLn $ "Or should I say " ++ capName ++ "?"

Note that it is not ∗ possible to "get out of" IO again. You should not try to
write functions like f :: IO String -> String.
(∗ It is, but unsafePerformIO is called like that for a reasons.)
dialogue :: IO ()
dialogue = do
    putStrLn "Hello! Who are you?"
    name <- getLine
    putStrLn $ "Nice to meet you, " ++ name ++ "!"
    let capName = map toUpper name
    putStrLn $ "Or should I say " ++ capName ++ "?"

Note that it is not* possible to “get out of IO” again. You should not try to write functions like \( f :: IO \text{ String} \rightarrow \text{ String} \).

(* It is, but unsafePerformIO is called like that for a reasons.)
sequence

The simplest thing one can do with a set of actions is put them in sequence. The command for this collects a list of monadic actions into a monadic list of actions. Read the last sentence again.

In do notation:

```haskell
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (x:xs) = do v <- x
                  vs <- sequence xs
                  return (v:vs)
```
sequence with IO

\[
\text{\texttt{\lambda> :t replicate}}
\]

\text{\texttt{replicate :: Int -> a -> \([a]\)}}
\[
\text{\texttt{\lambda> sequence (replicate 3 getLine)}}
\]

\text{bob}

\text{alice}

\text{carol}

\text{["bob","alice","carol"]}
sequence with IO

\( \lambda \) :t replicate
replicate :: Int -> a -> [a]
\( \lambda \) sequence (replicate 3 getLine)
bob
alice
carol
["bob", "alice", "carol"]

Since GHC 7.10 the sequence function was generalised from lists to a whole type class of Traversable structures. So its real type is this:

\( \lambda \) :t sequence
sequence :: (Monad m, Traversable t) => t (m a) -> m (t a)
If you are curious about what GHC actually does internally, read this blog post: https://www.fpcomplete.com/blog/2015/02/primitive-haskell
Randomness
Random Integers

Getting a random integer involves interaction with the outside world, because some aspect of the world (be it known or unknown to us) determines the output. (In UNIX this is usually /dev/random or /dev/urandom.)

\[ \text{getRandomInt :: Int} \rightarrow \text{IO Int} \]
\[ \text{getRandomInt } n = \text{getStdRandom (randomR (0, n))} \]

This gives:

\( \lambda > \text{getRandomInt 20} \)
16
\( \lambda > \text{getRandomInt 20} \)
18
Random Integer Lists

The following generates an integer list with entries in the range \([0..k]\) and lengths in the range \([0..n]\):

```haskell
getInts :: Int -> Int -> IO [Int]
getInts _ 0 = return []
getInts k n = do
  x <- getRandomInt k
  xs <- getInts k (n-1)
  return (x:xs)
```

Again, we can also write this without `do`:

```haskell
getInts' :: Int -> Int -> IO [Int]
getInts' _ 0 = return []
getInts' k n = getRandomInt k >>= \ x ->
  getInts k (n-1) >>= \ xs -> return (x:xs)
```
Finally, we can also choose the parameters randomly:

```haskell
genIntList :: IO [Int]
genIntList = do
    k <- getRandomInt 20
    n <- getRandomInt 10
    getInts k n
```

This gives, e.g.:

```haskell
λ> genIntList
[0,0,0,0]
λ> genIntList
[-1,-5,-3,-2,-1,6,2,-8]
λ> genIntList
[15,-10,7,-15,5,-13,15,11,13,-11]
```
Trees
Example: trees

Binary-branching trees with things of type \( a \) at the leaves:

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)
deriving (Eq, Ord, Show)
```
instance Functor Tree

First, trees can be viewed as functors. Let’s put this in a declaration, and define \( \text{fmap} \) for trees.

\[
\text{instance} \quad \text{Functor} \quad \text{Tree} \quad \text{where}
\]

\[
\text{-- fmap :: (a -> b) -> Tree a -> Tree b}
\]

\[
\text{fmap f (Leaf x)} = \text{Leaf (f x)}
\]

\[
\text{fmap f (Branch left right)} = \text{Branch (fmap f left)} \quad \text{(fmap f right)}
\]

Try this out, do an example in ghci.

Note that \( \text{fmap} \) can never change the shape of the tree! How does this compare to what \( \text{fmap} \) does in the Maybe monad?
instance Applicative Tree

instance Applicative Tree where

-- pure :: a -> Tree a
pure = Leaf

-- (<*>) :: Tree (a -> b) -> Tree a -> Tree b
(<*>) ftree (Leaf x) = fmap ($ x) ftree
(<*>) ftree (Branch xl xr) = Branch (ftree <*> xl) (ftree <*> xr)

Think about what this does, try out an example!
An alternative instance would use the following sequence function, recursing first on the of functions and then making full copies of the Tree a:

\[
\text{star} :: \text{Tree} (a \to b) \to \text{Tree} a \to \text{Tree} b
\]

\[
\text{star} (\text{Leaf } f) \text{ atree} = \text{fmap } f \text{ atree}
\]

\[
\text{star} (\text{Branch } fl \text{ fr}) \text{ atree} = \text{Branch} (fl \ `\text{star}` \ atree) (fr \ `\text{star}` \ atree)
\]

⇒ Exercise: Check whether both ways to implement instance Applicative Tree fulfil the laws for Functor and Applicative mentioned in the previous lecture.
instance Monad Tree

Wrapping up an arbitrary thing in a tree is done by means of Leaf.

Binding a leaf Leaf x to a function f :: a -> Tree b is just a matter of applying f to x.

Binding a tree Branch left right to f is just a matter of recursing over the branches. This gives the following instance declaration of trees as monads:

instance Monad Tree where
  -- return :: a -> Tree a
  return = Leaf
  -- (>>=) :: Tree a -> (a -> Tree b) -> Tree b
  (>>=) (Leaf x) f = f x
  (>>=) (Branch left right) f = Branch (left >>= f) (right >>= f)
If you also want things of type a at the intermediate nodes and allow arbitrary number of branches, this type works:

```haskell
data ArbTree a = Node a [ArbTree a]
```

Note that we no longer need an extra Leaf case. Leafs are just Nodes where the `[ArbTree a]` list of children is empty.

⇒ Exercise: Can you also make `ArbTree a` Functor etc.?
More Practical Stuff
**stack and cabal**

For projects with more than one `.hs` or `.lhs` file you should have a `stack.yaml` file in the same folder and a `.cabal` package description.

A minimal `stack.yaml`: is this:

```yaml
resolver: lts-11.11
```

This will also make sure that your code still works next year. 👍
And a minimal .cabal file:

name: lectures
version: 0.0.0
build-type: Simple
cabal-version: >= 1.10

library
default-language: Haskell2010
exposed-modules: L1, L2, L3, L4, E1, E2, E3
ghc-options: -Wall
build-depends: base >=4.9 && <5, QuickCheck, random, tf-random
You can use https://github.com/funcspec/report-example as a template for your project, or create a new package with lots of boilerplate by running stack new.
If you want to work in pairs, it makes sense to use the version control system git to collaborate. It takes half an hour to learn, but then you will never have a “Dropbox problem” overwriting each others work.

For a tutorial, click here, for a cheat sheet, click here.
If you want to work in pairs, it makes sense to use the version control system git to collaborate. It takes half an hour to learn, but then you will never have a “Dropbox problem” overwriting each others work.

For a tutorial, click here, for a cheat sheet, click here.
Git Hosting

The most widely used hosted services for *git* are *github* and *gitlab*.

You may submit your report via email, but using *git* and a service like *github* or *gitlab* is strongly preferred if you want to get help or comments on your code from us during the next week.

If you are having problems setting up a (private) repository, please email us your username and we will create a repository for you at https://github.com/funcspec or https://gitlab.com/funcproglog.
See you again at 14:00 in F1.15.