CSE130 Discussion Section

Week 9 - IO Monads

2021/05/28

Monad

```
class Monad m where

    -- bind

    (>>=)  :: m a -> (a -> m b) -> m b

    -- return

    return :: a -> m a
```

Either Monad

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right
```

Either Monad

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

>>> return 1 :: Either Int Int
```

Either Monad

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

>>> return 1 :: Either Int Int
Right 1
```

Either Monad

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

>>> Left 1 >>= \v -> Right (v + 1)
```

Either Monad

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right
```

```
>>> Left 1 >>= \v -> Right (v + 1)
Left 1
```

Either Monad

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

>>> Right 1 >>= \v -> Right (v + 1)
```

Either Monad

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

>>> Right 1 >>= \v -> Right (v + 1)
Right 2
```

Either Monad

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

>>> Right 1 >>= \v -> Left (v + 1)
```

Either Monad

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

>>> Right 1 >>= \v -> Left (v + 1)
Left 2
```

Either Monad

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

>>> Right 1 >>= Left
>>> Right 1 >>= \v -> Left v
```

Either Monad

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

>>> Right 1 >>= Left
Left 1
```

Either Monad for EThr

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

evalE env (EInt i) = ??
evalE env (EThr e) = ??
evalE env (EBin Plus e1 e2) = ??
```

Either Monad for EThr

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

evalE env (EInt i) = ?? -- this should return a normal value
```

Either Monad for EThr

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

evalE env (EInt i) = return (VInt i)
```

Either Monad for EThr

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

evalE env (EInt i) = return (VInt i)
evalE env (EInt i) = Right (VInt i)
```

Either Monad for EThr

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

evalE env (EThr e) = ??
```

Either Monad for EThr

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

evalE env (EThr e) = ??
```

1) evaluate `e` to value `eitherVal` (use `evalE`), remember `eitherVal` has the type `Either Value Value`

Either Monad for EThr

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right
```

```
evalE env (EThr e) = ??
```

2) if `eitherVal` is an exception (`eitherVal` is a `Left`), then return the exception directly, otherwise (`eitherVal` is a `Right`), take the value in `eitherVal`, wrap it with `Left` (turn into an exception) and return

Either Monad for EThr

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

evalE env (EThr e) = ??
eitherVal >>= Left
```

Either Monad for EThr

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

evalE env (EBin Plus e1 e2) = ??
```

1) evaluate e1, e2 to value v1, v2 (use evalE), remember v1,v2 has the type Either Value Value
2) if v1 or v2 is an exception (v is a Left), then return the exception directly, otherwise (v1, v2 are both Right), take the values, compute addition, wrap the addition result with Right and return

Either Monad for EThr

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

evalE env (EBin Plus e1 e2) = ??
```

First option:

```
evalE env e1 >>= \v1 ->
evalE env e2 >>= \v2 ->
Right (evalOp Plus v1 v2)
```

Either Monad for EThr

```
instance Monad (Either e) where
    Left  l >>= _ = Left l
    Right r >>= k = k r

    return = Right

evalE env (EBin Plus e1 e2) = ??
```

Second option:

```
do
    v1 <- evalE env e1
    v2 <- evalE env e2
    return (evalOp Plus v1 v2)
```

IO Monad

Useful IO functions

```
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b
(=<<) :: (a -> IO b) -> IO a -> IO b
(>>) :: IO a -> IO b -> IO b

print :: Show a => a -> IO ()
putStrLn :: String -> IO ()
putStr :: String -> IO ()

readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
doesFileExist :: FilePath -> IO Bool
```

IO Monad

Useful IO functions

```
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b
(=<<) :: (a -> IO b) -> IO a -> IO b
(>>) :: IO a -> IO b -> IO b

>>> return 1
1

>>> return 3 >>= \v -> return (v * v)
9

>>> (\v -> return (v * v)) =<< return 3
9
```

IO Monad

Useful IO functions

```
print :: Show a => a -> IO ()
putStrLn :: String -> IO ()
putStr :: String -> IO ()

>>> return (reverse [1,2,3]) >>= print
[3,2,1]

>>> putStrLn “first” >> putStrLn “second”
first
second
```

equivalent to

```
do
  putStrLn “first”
  putStrLn “second”
```

IO Monad

Useful IO functions

```
print :: Show a => a -> IO ()
putStrLn :: String -> IO ()
putStr :: String -> IO ()

>>> putStrLn “first” >> putStrLn “second”
first
second

>>> putStr “first” >> putStr “second”
firstsecond
```

IO Monad

Useful IO functions

```
print :: Show a => a -> IO ()
putStrLn :: String -> IO ()
putStr :: String -> IO ()

>>> print “first”
“first”
                    with quotes!
>>> putStrLn “first”
first
```

IO Monad

Useful IO functions

```
(import System.IO)
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()

(import System.Directory)
doesFileExist :: FilePath -> IO Bool

>>> readFile “test.txt” >>= print
…
>>> writeFile “test.txt” “second”

>>> doesFileExist “test.txt”
False
```

IO Monad

```
reverseWords :: String -> String
reverseWords = unwords . map reverse . words


main = do
    line <- getLine
    if null line
        then return ()
        else do
            let reversedLine = reverseWords line
            putStrLn reversedLine
```

IO Monad

```
reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

```
words :: String -> [String]
words “hello world” == [“hello”, “world”]
```

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            let reversedLine = reverseWords line
            putStrLn reversedLine
```

IO Monad

```
reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

```
unwords :: [String] -> String
unwords [“hello”, “world”] == “hello world”
```

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            let reversedLine = reverseWords line
            putStrLn reversedLine
```

IO Monad

```
reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

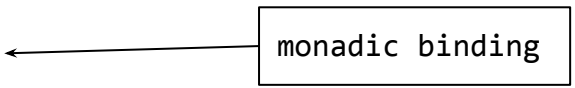
```
(.) :: (b -> c) -> (a -> b) -> a -> c
function composition
```

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            let reversedLine = reverseWords line
            putStrLn reversedLine
```

IO Monad

```
reverseWords :: String -> String
reverseWords = unwords . map reverse . words


main = do
    line <- getLine
    if null line
        then return ()
        else do
            let reversedLine = reverseWords line
            putStrLn reversedLine
```

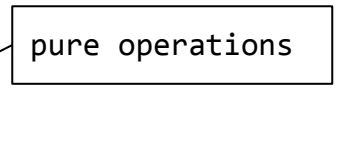


IO Monad

```
reverseWords :: String -> String

reverseWords = unwords . map reverse . words


main = do
    line <- getLine
    if null line
        then return ()
        else do
            let reversedLine = reverseWords line
            putStrLn reversedLine
```



IO Monad

for the assignment

- commands always start with ‘:’
- use `L.isPrefixOf` and `pfx*` to match the command name
- use `chomp` to get contents following the command name, which is the filename to run or load
- use `putStrLn` to print things to the console
- use `return` to put things into the IO monad