I/O

Stephen A. Edwards

Columbia University

Fall 2022

At Long Last: Hello World

```haskell
-- hello.hs
main = putStrLn "Hello, World!"
```

To run it directly:

```
$ stack runhaskell hello
Hello, World!
```

To compile it into an executable:

```
$ stack ghc -- --make hello
[1 of 1] Compiling Main             ( hello.hs, hello.o )
Linking hello ...
$ ./hello
Hello, World!
```

I/O Actions

```
-- hello.hs
main = putStrLn "Hello, World!"
```

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()      -- Returns an IO action
Prelude> :k IO
IO :: * -> *                     -- An IO action may convey a result
Prelude> :t ()
() :: ()                         -- () is the only literal of type ()
Prelude> :k ()
() :: *                          -- a concrete type with single literal
```

Every IO action (e.g., printing, reading), produces an IO object

Output-only actions (e.g., printing), return `IO ()`

Input actions (e.g., reading a line), return something like `IO String`

Sequencing is Fundamental to I/O: *do* Blocks

```haskell
-- hello2.hs
main :: IO ()
main = do
  putStrLn "Hello. What is your name?" -- Print the string
  name <- getLine                      -- Read a line; bind result to name
  putStrLn $ "Hello, " ++ name
```

```
$ stack runhaskell hello2
Hello. What is your name?
Stephen
Hello, Stephen
```

```
*Main> :t getLine
getLine :: IO String
```

Indentation rules for *do* blocks same as those for *where*, *let*, and *do*.

I/O Actions Are Expressions That Produce an IO *t*

Effectively an implicit _ <- if you don't write your own (except the last line)

```
-- putstrln1.hs
main = do
  result <- putStrLn "Hello World"    -- Not that you'd want to...
  print result                        -- putStrLn . show
```

```
*Main> :l putstrln1
[1 of 1] Compiling Main             ( putstrln1.hs, interpreted )
Ok, one module loaded.
*Main> main
Hello World
()
*Main> :t print
print :: Show a => a -> IO ()
```

Let Blocks: The Third Type of *do* Block Statement Syntax

```
-- let1.hs
import Data.Char(toUpper)

main = do            -- The three kinds of syntax for do block statements:
  putStr "First Name? "                -- 1/3: expr
  fname <- getLine                     -- 2/3: name <- expr
  putStr "Last Name? "
  lname <- getLine
  let fshout = map toUpper fname       -- 3/3: let decls
      lshout = map toUpper lname       -- in not used in do blocks
  putStrLn $ "WELCOME " ++ fshout ++ " " ++ lshout
```

```
$ stack runhaskell let1
First Name? Stephen
Last Name? Edwards
WELCOME STEPHEN EDWARDS
```

Let is for pure Haskell; <- takes a result from an I/O action

I/O actions are just normal Haskell expressions until connected to *main*

```haskell
-- let2.hs
printTwo = putStrLn "Two"

main = do
  putStrLn "One"
  let printFour = putStrLn "Four"
      getMyLine = getLine
      printThree = putStrLn "Three"
  printTwo
  printThree
  putStr "Type something "
  myLine <- getMyLine
  printFour
  putStrLn $ "You typed \"" ++
     myLine ++ "\""
```

```
$ stack runhaskell let2
One
Two
Three
Type something OK
Four
You typed "OK"
```

The I/O actions in the *let* block don't do anything until they're referenced in the *do* block

Word Reverser Program → droW resreveR margorP

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

main = do
  line <- getLine
  if null line then       -- if-then-else is an expression, so both
    return ()             -- branches must return the same thing but
  else do                 -- return doesn't do quite what you think
    putStrLn $ reverseWords line
    main
```

```
$ stack runhaskell reverser
able elba stressed diaper looter debut deeps devil peels
elba able desserts repaid retool tubed speed lived sleep
tacocat deified civic radar rotor kayak aibohphobia
tacocat deified civic radar rotor kayak aibohphobia
```

Aibohphobia: Fear of palindromes

Return Encapsulates a Value in a *do* Block

```
readFromUser :: IO String
readFromUser = getLine

justReturn :: IO String
justReturn = do
  putStrLn "justReturn invoked"
  return "this string"

main :: IO ()
main = do
  line1 <- readFromUser
  putStrLn line1
  line2 <- justReturn
  putStrLn "after justReturn"
  putStrLn line2
```

A *do* block returns the value of the last expression, which must be of type IO *t* and cannot be a *let* or <-.

Return is a vacuous I/O action that puts a value in an IO *t*

Set the return value of a *do* block with a *return* at the end

```
$ stack runhaskell do1
I typed this
I typed this
justReturn invoked
after justReturn
this string
```

Return does not return control; <- is the inverse of *return*

```haskell
-- do2.hs
main :: IO ()
main = do
  return "tree falls in the forest" -- No one is listening
  return ()                         -- No control transfer
  a <- return "something "       -- Effectively let a = "something "
  b <- do                           -- do runs actions in sequence
    return "silence"                -- Also does not return
    putStrLn "return did not return"
    return "else "                  -- “else” is bound to b
  let c = "was returned"
  putStrLn $ a ++ b ++ c
```

```
$ stack runhaskell do2
return did not return
something else was returned
```

Basic I/O Functions

```
putChar  :: Char   -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()        -- Adds a newline
print    :: Show a => a -> IO ()   -- putStrLn . show
```

```
getChar     :: IO Char                 -- End-of-file throws an exception
getLine     :: IO String               -- Read up to newline
getContents :: IO String               -- Read entire input (lazily)
interact    :: (String -> String) -> IO ()  -- Read, apply f, print
readIO      :: Read a => String -> IO a     -- Parse a string in a do
readLn      :: Read a => IO a               -- Read a line and parse
```

```
import Data.Char(toUpper)
main :: IO ()
main = interact $
         map toUpper
```

```
$ stack runhaskell interact < interact.hs
IMPORT DATA.CHAR(TOUPPER)
MAIN :: IO ()
MAIN = INTERACT $
         MAP TOUPPER
```

Implementations of Output Functions

putChar is a primitive

```
putStr          :: String -> IO ()   -- Equivalent to the Prelude def.
putStr []       = return ()          -- Produces an IO ()
putStr (x:xs)   = do putChar x
                     putStr xs       -- Recurse
```

```
putStrLn    :: String -> IO ()
putStrLn s =   do putStr s
                  putStr "\n"     -- Print a newline after the string
```

```
print       :: Show a => a -> IO ()
print x     =  putStrLn (show x) -- Transform to string with show
```

Implementations of Input Functions

```haskell
getLine     :: IO String
getLine     =  do c <- getChar
                  if c == '\n' then return "" else
                     do s <- getLine -- Recurse: get the rest
                        return (c:s)
```

```haskell
interact    ::  (String -> String) -> IO ()
interact f  =  do hSetBuffering stdin  NoBuffering -- Disable
                  hSetBuffering stdout NoBuffering --  buffering
                  s <- getContents     -- Lazily read all the input
                  putStr (f s)         -- Starts before input is done
```

When is an *if* without an *else* for *do* blocks

```haskell
when :: Bool -> IO () -> IO () -- Prelude definition is more general
when p s = if p then s else return ()
```

```haskell
-- when.hs
import Control.Monad(when) -- “Monad” in Category Theory is “Action”

main :: IO ()
main = do c <- getChar
          when (c /= ' ') $ do putChar c
                               main
```

The default is line buffering: a whole line is read before it is examined

```
$ stack runhaskell when
This-will-stop-at-the-first-space did it?
This-will-stop-at-the-first-space$
```

sequence Applies a List of I/O Actions and Captures the Result

```
sequence :: [IO a] -> IO [a] -- Prelude definition is more general
```

```
main :: IO () -- Like Unix head: print the first 10 input lines
main = do
    inputLines <- sequence $ replicate 10 getLine
    sequence_ $ map putStrLn inputLines -- sequence_ discards result
```

mapM or mapM_, which discards the result, is better for the second *sequence*

```
mapM  :: (a -> IO b) -> [a] -> IO [b] -- Not the actual type;
mapM_ :: (a -> IO b) -> [a] -> IO ()  --  Prelude def. is more general
```

```
main :: IO ()
main = do
    inputLines <- sequence $ replicate 10 getLine
    mapM_ putStrLn inputLines -- Apply putStrLn to lines, return IO ()
```

forM and *forM_* are just *mapM* with arguments reversed

Why? Because it makes *forM* look like a traditional *for* loop (well, *foreach*)

```
import Control.Monad(forM, forM_)

main :: IO ()
main = do
  colors <- forM ([1..4] :: [Int]) $ \a -> do
    putStrLn $ "What color is #" ++ show a ++ "?"
    getLine                          -- Result saved in colors
  putStrLn "You ranked the colors"
  forM_ colors putStrLn              -- forM_ returns IO ()
```

The version in *Learn You a Haskell...* is redundant:

```
colors <- forM [1,2,3,4] (\a -> do -- Unnecessary parentheses
   putStrLn $ "Which .."
   color <- getLine
   return color)           -- This is what getLine would return anyway
```

```
*Main> main
What color is #1?
Red
What color is #2?
Green
What color is #3?
Blue
What color is #4?
Black
You ranked the colors
Red
Green
Blue
Black
```

```
mapM f as = sequence (map f as)    -- Prelude definitions
forM = flip mapM
```

Forever Loops Forever

```haskell
-- forever.hs
import Control.Monad(forever)
import Data.Char(toUpper)

main :: IO ()
main = forever $ do
  l <- getLine
  putStrLn $ map toUpper l
```

```
$ stack runhaskell forever < forever.hs
-- FOREVER.HS
IMPORT CONTROL.MONAD(FOREVER)
IMPORT DATA.CHAR(TOUPPER)

MAIN :: IO ()
MAIN = FOREVER $ DO
  L <- GETLINE
  PUTSTRLN $ MAP TOUPPER L
forever: <stdin>: hGetLine: end of file
```

```haskell
import System.IO(openFile, IOMode(ReadMode), hGetContents,
                 hClose, hPutStrLn, stderr)
import System.Exit(exitFailure); import Data.Char(isAlpha, toLower)
import System.Environment(getArgs, getProgName)

main :: IO () -- Report whether each line of a file is a palindrome
main = do args <- getArgs
          case args of
            [filename] -> do                    -- Expects one filename
              h <- openFile filename ReadMode
              contents <- hGetContents h        -- Read the file
              mapM_ (putStrLn . isAPalindrome) $ lines contents
              hClose h
            _ -> do pn <- getProgName           -- Usage message
                    hPutStrLn stderr $ "Usage: "++pn++" <filename>"
                    exitFailure                 -- Terminate the program

isAPalindrome :: String -> String -- Report whether the string is one
isAPalindrome s = s ++ ": " ++ show (ls == reverse ls)
                where ls = map toLower $ filter isAlpha s
```

palindromes.txt:

```
Able was I saw elba
Taco cat
Race car
Palindrome
A man, a plan, a canal, Panama!
```

```
$ stack runhaskell palindrome palindromes.txt
Able was I saw elba: True
Taco cat: True
Race car: True
Palindrome: False
A man, a plan, a canal, Panama!: True
```

```haskell
-- System.Environment    Command-line args; environment variables
getArgs :: IO [String]       -- The list of command-line arguments
getProgName :: IO String     -- Name of the invoked program (argv[0])

-- System.IO    File Handle; open; close; read; write; "h" I/O action variants
type FilePath = String
openFile :: FilePath -> IOMode -> IO Handle
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
stderr :: Handle                             -- Handle for standard error
hGetContents :: Handle -> IO String          -- getContents from a Handle
hPutStrLn :: Handle -> String -> IO ()       -- putStrLn to a Handle
hClose :: Handle -> IO ()                    -- Close the (file) handle
withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r
readFile :: FilePath -> IO String

-- System.Exit    Like exit() in the C standard library
exitFailure :: IO a     -- Terminate program with a failure code
```

```haskell
import System.IO(withFile, IOMode(ReadMode), hGetContents,
                 hPutStrLn, stderr)
import System.Exit(exitFailure); import Data.Char(isAlpha, toLower)
import System.Environment(getArgs, getProgName)

main :: IO ()
main = do args <- getArgs
          case args of
            [filename] -> do
              withFile filename ReadMode (\h -> do  -- Simpler
                contents <- hGetContents h
                mapM_ (putStrLn . isAPalindrome) $ lines contents)
            _ -> do pn <- getProgName
                    hPutStrLn stderr $ "Usage: "++pn++" <filename>"
                    exitFailure

isAPalindrome :: String -> String
isAPalindrome s = s ++ ": " ++ show (ls == reverse ls)
                  where ls = map toLower $ filter isAlpha s
```

```haskell
import System.IO(readFile)
import System.Exit(die); import Data.Char(isAlpha, toLower)
import System.Environment(getArgs, getProgName)

main :: IO ()
main = do args <- getArgs
          case args of
            [filename] -> do
              contents <- readFile filename -- Even simpler
              mapM_ (putStrLn . isAPalindrome) $ lines contents
            _ -> do pn <- getProgName
                    die $ "Usage: " ++ pn ++ " <filename>"

isAPalindrome :: String -> String
isAPalindrome s = s ++ ": " ++ show (ls == reverse ls)
                  where ls = map toLower $ filter isAlpha s
```

```haskell
-- More in System.IO
hGetChar      :: Handle -> IO Char
hGetLine      :: Handle -> IO String
hPutStr       :: Handle -> String -> IO ()
hFlush        :: Handle -> IO ()
data BufferMode
  = NoBuffering | LineBuffering | BlockBuffering (Maybe Int)
hSetBuffering :: Handle -> BufferMode -> IO ()
openTempFile  :: FilePath -> String -> IO (FilePath, Handle)
writeFile     :: FilePath -> String -> IO ()
appendFile    :: FilePath -> String -> IO ()

-- System.Directory
removeFile    :: FilePath -> IO ()
renameFile    :: FilePath -> FilePath -> IO ()
renamePath    :: FilePath -> FilePath -> IO ()
listDirectory :: FilePath -> IO [FilePath]
```

ByteString: Faster strings

```
type String = [Char]
```

Data.ByteString implements strings as packed Word8 (byte) arrays: compact and faster

Data.ByteString is strict (no laziness, infinite lists, etc.)

Data.ByteString.Lazy is “lazy” on 64K blocks

Data.ByteString.Char8 and Data.ByteString.Lazy.Char8 work with Char8 arrays instead of Word8

"grep" with String

```
import Data.List(isInfixOf)
import System.Environment(getArgs, getProgName)
import System.Exit(die)

main :: IO ()
main = do args <- getArgs
          (pat, filename) <- case args of
            [p, f] -> return (p, f)
            _ -> do pn <- getProgName
                    die $ "Usage: "++pn++" <pattern> <filename>"
          file <- readFile   filename
          putStr   $ grep         pat  file

grep :: String       -> String        -> String
grep pat input =
     unlines   $ filter (isInfixOf   pat) $ lines   input
```

"grep" with Data.ByteString.Char8

```
import qualified Data.ByteString.Char8 as B
import System.Environment(getArgs, getProgName)
import System.Exit(die)

main :: IO ()
main = do args <- getArgs
          (pat, filename) <- case args of
            [p, f] -> return (p, f)
            _ -> do pn <- getProgName
                    die $ "Usage: "++pn++" <pattern> <filename>"
          file <- B.readFile filename
          B.putStr $ grep (B.pack pat) file
                              -- pack :: String → Bytestring
grep :: B.ByteString -> B.ByteString -> B.ByteString
grep pat input =
     B.unlines $ filter (B.isInfixOf pat) $ B.lines input
```

“grep” with Data.ByteString.Lazy.Char8

```
import qualified Data.ByteString.Lazy.Char8 as B
import System.Environment(getArgs, getProgName)
import System.Exit(die)

main :: IO ()
main = do args <- getArgs
          (pat, filename) <- case args of
            [p, f] -> return (p, f)
            _ -> do pn <- getProgName
                    die $ "Usage: "++pn++" <pattern> <filename>"
          file <- B.readFile filename
          B.putStr $ grep (B.pack pat) file
                               -- pack :: String → Bytestring
grep :: B.ByteString -> B.ByteString -> B.ByteString
grep pat input =
     B.unlines $ filter (isInfixOf   pat) $ B.lines input where
   isInfixOf p s = any (B.isPrefixOf p) $ B.tails s
```

Quick Experiment

Selecting 3500 lines that contain “fe” from a 49M/218 kl log file:

```
$ stack ghc -- --make -O bgrep.hs
$ /usr/bin/time -f "%E %M" ./bgrep fe /tmp/log > /dev/null
```

Version	Time	Memory	Note
String	2600 ms	6.2 MB	[Char]
ByteString.Lazy	1300 ms	6.2 MB	64K blocks
ByteString	110 ms	56 MB	Single byte array; naïve isInfixOf
grep	40 ms	2.5 MB	GNU implementation; >3000 LoC

Exceptions

TL;DR: Don't use 'em; use something like *Maybe* or *Either*

Work best in I/O contexts (sequential evaluation; lots to go wrong)

Only I/O code can catch exceptions, but they may be thrown anywhere

Some of the I/O exception handling functions in System.IO.Error:

```
catchIOError          :: IO a -> (IOError -> IO a) -> IO a
isUserError           :: IOError -> Bool
isDoesNotExistError   :: IOError -> Bool
isPermissionError     :: IOError -> Bool
ioeGetFileName        :: IOError -> Maybe FilePath
```

More extensive exception facilities in Control.Exception

Line Count with some error checking

```haskell
import System.Environment(getArgs)
import System.IO.Error(catchIOError, isUserError,
            isDoesNotExistError, ioeGetFileName, isPermissionError)
import System.Exit(die)
import qualified Data.ByteString.Char8 as B

main :: IO ()
main = do [filename] <- getArgs               -- Match may fail
          contents <- B.readFile filename -- Many possible failures
          print $ length $ B.lines contents

  `catchIOError` \ e -> die $ case ioeGetFileName e of

    Just fn | isDoesNotExistError e -> fn ++ ": No such file"
            | isPermissionError e   -> fn ++ ": Permission denied"
    _       | isUserError e         -> "Usage: lc <filename>"
            | otherwise             -> show e
```

Line Count in Action

```
$ stack ghc -- --make -O -Wall lc.hs
[1 of 1] Compiling Main             ( lc.hs, lc.o )
Linking lc ...
$ ./lc
Usage: lc <filename>
$ ./lc foo bar
Usage: lc <filename>
$ ./lc foo
foo: No such file
$ ./lc /var/log/btmp
/var/log/btmp: Permission denied
$ ./lc /var/log/syslog
4705
```