

# Building a Console Application in Haskell

Rebecca Skinner

<2022-12-08 Thu>

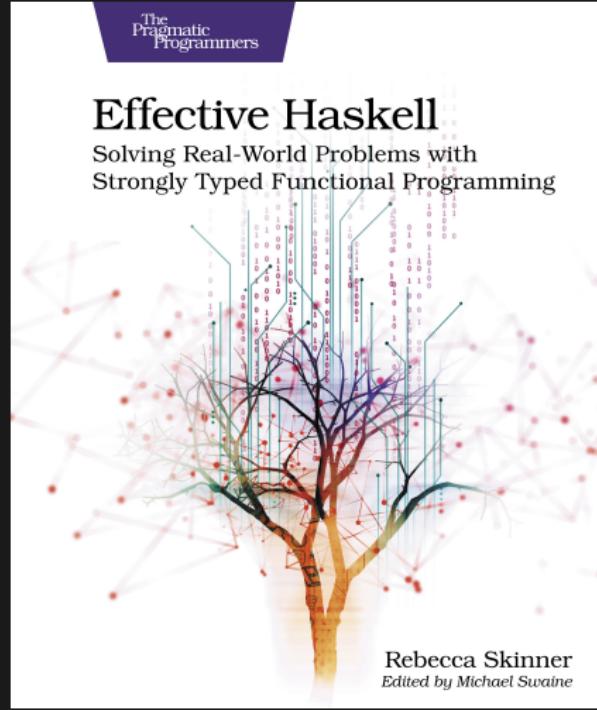
# Prelude

# Hello, World

- ▶ About Me: Rebecca Skinner
  - ▶ Lead Software Engineer at Mercury
  - ▶ Author of Effective Haskell
- ▶ @cercerilla on Twitter and Cohost
- ▶ <https://rebeccaskinner.net>
- ▶ <https://github.com/rebeccaskinner/>



# Effective Haskell



Rebecca Skinner  
*Edited by Michael Swaine*



<https://tinyurl.com/2744kfu7>

Now in Beta!

# About This Talk

During this talk we're going to discover how to build basic command line tool in Haskell. As we go, you'll:

- ▶ Learn how Haskell programs use IO actions to deal with the real world
- ▶ Find out how to do simple terminal and file IO
- ▶ See examples of how to mix IO and pure functional code effectively
- ▶ Follow along with implementing pure functional code to work with text

**Most importantly:** You'll get an intuition for how to think about building Haskell programs that can serve as a basis for future learning.

# HCat

```
{--# LANGUAGE RecordWildCards #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE TypeApplications #-}

module HCat where

import qualified System.IO.Error as IOError
import qualified Control.Exception as Exception

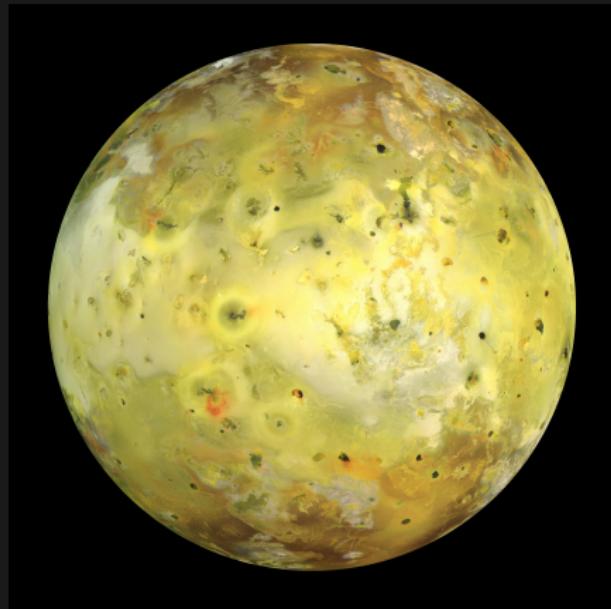
import qualified System.Environment as Environment
import qualified System.Exit as Exit
import qualified Data.List as List
import qualified Data.Maybe as Maybe
import qualified Data.Char as Char
import qualified Text.Printf as Printf
import qualified Data.Text as Text
import qualified Data.Text.IO as TextIO
import qualified Data.ByteString as BS
import System.IO ( openFile
                  , hPutStr
                  , hFlush
                  , hClose
                  , hFileSize
                  , hGetContents
                  , stdin
                  , stdout
                  , hGetChar
                  , hSetBuffering
                  , hGetBuffering
                  , hSetEcho
                  , BufferMode (..)
                  , Handle
                  , IOMode (..) )
import qualified System.Info
import qualified System.Process as Process
import Text.Read
import Control.Monad
import qualified Data.Time.Clock as Clock
import qualified Data.Time.Format as TimeFormat

-- import qualified System.Posix.User as PosixUser
import qualified System.Directory as Directory

--START:FileInfo
./src/HCat.hs | permissions: rw- | 19337 bytes | modified: 2022-01-25 04:15:03 | page: 1 of 15 |
```

# Understanding IO

# A True Color Photo of Side Effects



A side effect in its natural environment.

# The Trouble with IO

Haskell is a **pure functional** language, but most of the things we want our programs to do revolve around **side effects**!

# The Trouble with IO

Haskell is a **pure functional** language, but most of the things we want our programs to do revolve around **side effects**!

- ▶ Reading and writing files

# The Trouble with IO

Haskell is a **pure functional** language, but most of the things we want our programs to do revolve around **side effects**!

- ▶ Reading and writing files
- ▶ Printing text to the screen

# The Trouble with IO

Haskell is a **pure functional** language, but most of the things we want our programs to do revolve around **side effects**!

- ▶ Reading and writing files
- ▶ Printing text to the screen
- ▶ Handling user input

# Can We Have a Little Bit of IO?

What if we cheat just a little?

# Can We Have a Little Bit of IO?

What if we cheat just a little?

---

```
writeReadFile =
  let
    _ = writeFile "example.txt" "Hello, Haskell"
    fileContents = readFile "example.txt"
  in print fileContents
```

---

# Can We Have a Little Bit of IO?

What if we cheat just a little?

---

```
writeReadFile =
  let
    _ = writeFile "example.txt" "Hello, Haskell"
    fileContents = readFile "example.txt"
  in print fileContents
```

---

- ▶ Nothing will happen until we evaluate `writeReadFile`

# Can We Have a Little Bit of IO?

What if we cheat just a little?

---

```
writeReadFile =
  let
    _ = writeFile "example.txt" "Hello, Haskell"
    fileContents = readFile "example.txt"
  in print fileContents
```

---

- ▶ Nothing will happen until we evaluate `writeReadFile`
- ▶ When we evaluate `writeReadFile` we'll get whatever random contents were in `example.txt`

# Can We Have a Little Bit of IO?

What if we cheat just a little?

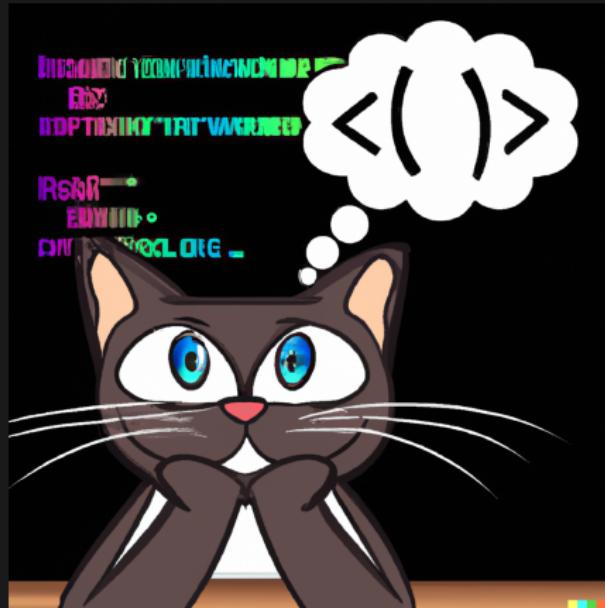
---

```
writeReadFile =
  let
    _ = writeFile "example.txt" "Hello, Haskell"
    fileContents = readFile "example.txt"
  in print fileContents
```

---

- ▶ Nothing will happen until we evaluate `writeReadFile`
- ▶ When we evaluate `writeReadFile` we'll get whatever random contents were in `example.txt`
- ▶ We won't ever write "**Hello, Haskell**" to the file, because we're not using result of `writeFile`!

# Let's Dream of a Better Way



Let's dream up a better way

# IO, the Lazy Way

If we want to be lazy, we need to work for it by making sure every new side effect **must depend on** the previous one.



# Sometimes Things Are Easy

In some cases, there is a natural dependency between side effects.

# Sometimes Things Are Easy

In some cases, there is a natural dependency between side effects.

- ▶ Reading a file, then printing the contents

# Sometimes Things Are Easy

In some cases, there is a natural dependency between side effects.

- ▶ Reading a file, then printing the contents
- More often, there isn't an obvious dependency

# Sometimes Things Are Easy

In some cases, there is a natural dependency between side effects.

- ▶ Reading a file, then printing the contents

More often, there isn't an obvious dependency

- ▶ Writing a log message before opening a file
- ▶ Writing data to a file, then reading the contents
- ▶ Printing a message to the screen then waiting on user input

# A Pointer To The Real World

We needed to **sequence** our side effects correctly because there's an implicit data dependency we haven't considered: **the state of the real world.**

# A Pointer To The Real World

We needed to **sequence** our side effects correctly because there's an implicit data dependency we haven't considered: **the state of the real world**.



data RealWorld

# Welcome to the Real World

We can use a reference to the `RealWorld` to add a dependency between all of our calls:

# Welcome to the Real World

We can use a reference to the RealWorld to add a dependency between all of our calls:

---

```
writeReadFile world0 =  
  let  
    (world1, _) = writeFile world0 "example.txt" "Hello, Haskell"  
    (world2, fileContents) = readFile world1 "example.txt"  
  in print world2 fileContents
```

---

# Welcome to the Real World

We can use a reference to the RealWorld to add a dependency between all of our calls:

---

```
writeReadFile world0 =  
  let  
    (world1, _) = writeFile world0 "example.txt" "Hello, Haskell"  
    (world2, fileContents) = readFile world1 "example.txt"  
  in print world2 fileContents
```

---

But it sucks.

# Typing IO Operations



Let's make a type!

---

```
data SideEffect a =  
  SideEffect { runSideEffects :: RealWorld -> (RealWorld, a) }
```

---

# Side Effects Are Programs

Think of `SideEffect a` as a **program** that returns a value of type `a`.

`SideEffect String` : A program that runs and outputs a `String`

`SideEffect Int` : A program that runs and outputs an `Int`

`SideEffect` programs are not pure functional programs. They rely on, and change, the `RealWorld`.

# Side Effect Examples

Let's look at some examples of SideEffect programs. We'll imagine some internal helper functions that will do the unsafe low level IO operations:

---

```
readFile :: FilePath -> SideEffect String
readFile filename = SideEffect $ \realWorld ->
  let (realWorld', contents) = internalReadFile filename realWorld
  in (realWorld', contents)

writeFile :: FilePath -> String -> SideEffect ()
writeFile filename contents = SideEffect $ \realWorld ->
  let realWorld' = internalWriteFile filename contents realWorld
  in (realWorld', ())

print :: String -> SideEffect ()
print message = SideEffect $ \realWorld ->
  let realWorld' = internalPrint message realWorld
  in (realWorld', ())
```

---

# Combining Side Effects

A `SideEffect` program can do things that have side effects, like reading from and writing to files, but that's pretty limiting. We can do a lot more if we can have a `SideEffect` program that executes other `SideEffect` programs and uses the results.

---

```
data SideEffect a =
  SideEffect { runSideEffects :: RealWorld -> (RealWorld, a) }

joinSideEffects :: SideEffect (SideEffect a) -> SideEffect a
joinSideEffects outerSideEffect = SideEffect $ \world ->
  let (world', innerSideEffect) = runSideEffects outerSideEffect world
  in runSideEffects innerSideEffect world'
```

---

# First One, Then The Other

Most of the time, we want to write a `SideEffect` program that does one side effect **and then** does another one. It turns out that this is just another way of saying that we have one `SideEffect` program that calls the first effect, and uses its value to call the second one:

---

```
data SideEffect a =
  SideEffect { runSideEffects :: RealWorld -> (RealWorld, a) }

sequenceSideEffects :: SideEffect a -> (a -> SideEffect b) -> SideEffect b
sequenceSideEffects sideEffect makeNextSideEffect =
  joinSideEffects $ SideEffect $ \world ->
    let (world', val) = runSideEffects sideEffect world
    in (world', makeNextSideEffect val)
```

---

# Write, Read, Print

Let's go to write our program again, using the things we've just built:

---

```
writeReadFile :: SideEffect ()
writeReadFile =
  writeFile "example.txt" "Hello, Haskell"
  `sequenceSideEffects` (\_ -> readFile "example.txt")
  `sequenceSideEffects` (\contents -> print contents)
```

---

How does this version compare?

# Write, Read, Print

Let's go to write our program again, using the things we've just built:

```
writeReadFile :: SideEffect ()  
writeReadFile =  
  writeFile "example.txt" "Hello, Haskell"  
  `sequenceSideEffects` (\_ -> readFile "example.txt")  
  `sequenceSideEffects` (\contents -> print contents)
```

How does this version compare?

- ▶ Every side effect depends on its predecessor, so they all happen in the right order

# Write, Read, Print

Let's go to write our program again, using the things we've just built:

```
writeReadFile :: SideEffect ()  
writeReadFile =  
  writeFile "example.txt" "Hello, Haskell"  
  `sequenceSideEffects` (\_ -> readFile "example.txt")  
  `sequenceSideEffects` (\contents -> print contents)
```

How does this version compare?

- ▶ Every side effect depends on its predecessor, so they all happen in the right order
- ▶ Our code is focused on the work it needs to do, without having to explicitly pass around references to the real world

# Write, Read, Print

Let's go to write our program again, using the things we've just built:

```
writeReadFile :: SideEffect ()  
writeReadFile =  
  writeFile "example.txt" "Hello, Haskell"  
  `sequenceSideEffects` (\_ -> readFile "example.txt")  
  `sequenceSideEffects` (\contents -> print contents)
```

How does this version compare?

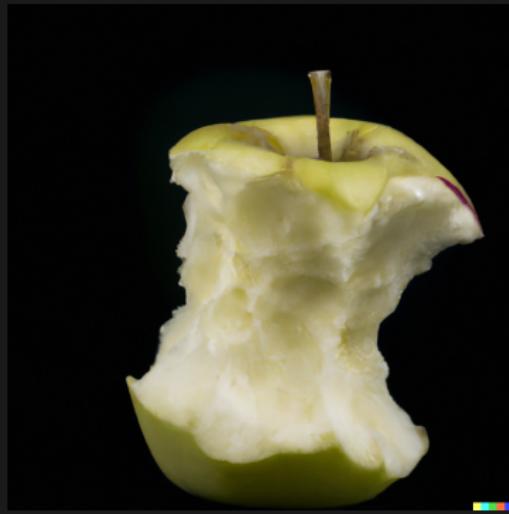
- ▶ Every side effect depends on its predecessor, so they all happen in the right order
- ▶ Our code is focused on the work it needs to do, without having to explicitly pass around references to the real world
- ▶ Our code program is still a **pure functional program**. Instead of doing side effects directly, we **generate a program** that would have side effects if it were run. The programs themselves are still pure values.

# That's Not All

Before we get back to HCat

# That's Not All

Before we get back to HCat



One more thing

# That's No Side Effect

```
sayHello :: IO ()  
sayHello =  
    putStrLn "Hello, World!"
```

U:~\*\*- Hello.hs 18% LS (Haskell +8 FlyC ivy ElDoc) 12:02AM 0.36

*Ceci n'est pas une side effect.*

# That's No Side Effect

It turns out our imaginary `SideEffect` type isn't entirely imaginary.

# That's No Side Effect

It turns out our imaginary `SideEffect` type isn't entirely imaginary.

- ▶ Instead of `SideEffect a` we say `IO a`

# That's No Side Effect

It turns out our imaginary `SideEffect` type isn't entirely imaginary.

- ▶ Instead of `SideEffect a` we say `IO a`
- ▶ Instead of `sequenceSideEffects` we say `>=`

# That's No Side Effect

It turns out our imaginary `SideEffect` type isn't entirely imaginary.

- ▶ Instead of `SideEffect a` we say `IO a`
- ▶ Instead of `sequenceSideEffects` we say `>=`
- ▶ Instead of `SideEffect program` we say `IO action`

# That's No Side Effect

It turns out our imaginary `SideEffect` type isn't entirely imaginary.

- ▶ Instead of `SideEffect a` we say `IO a`
- ▶ Instead of `sequenceSideEffects` we say `>=`
- ▶ Instead of `SideEffect program` we say `IO action`

---

```
writeReadFile :: IO ()  
writeReadFile =  
    writeFile "example.txt" "Hello, Haskell"  
    >>= (\_ -> readFile "example.txt")  
    >>= print
```

---

## To do List

Writing a long chain of calls to `>=` gets tiresome. Instead we can use **do notation**:

---

```
writeReadFile :: IO ()  
writeReadFile = do  
    writeFile "example.txt" "Hello, Haskell"  
    contents <- readFile "example.txt"  
    print contents
```

---

# To do List

Writing a long chain of calls to `>=` gets tiresome. Instead we can use **do notation**:

---

```
writeReadFile :: IO ()  
writeReadFile = do  
    writeFile "example.txt" "Hello, Haskell"  
    contents <- readFile "example.txt"  
    print contents
```

---

- ▶ Each line in a **do** block corresponds to `>=`

# To do List

Writing a long chain of calls to `>=` gets tiresome. Instead we can use **do notation**:

---

```
writeReadFile :: IO ()  
writeReadFile = do  
    writeFile "example.txt" "Hello, Haskell"  
    contents <- readFile "example.txt"  
    print contents
```

---

- ▶ Each line in a **do** block corresponds to `>=`
- ▶ The `<-` arrow names the output of an IO action

# To do List

Writing a long chain of calls to `>=` gets tiresome. Instead we can use **do notation**:

---

```
writeReadFile :: IO ()  
writeReadFile = do  
    writeFile "example.txt" "Hello, Haskell"  
    contents <- readFile "example.txt"  
    print contents
```

---

- ▶ Each line in a **do** block corresponds to `>=`
- ▶ The `<-` arrow names the output of an IO action
- ▶ When we run a Haskell program, the initial state of the real world is used to run an IO action named **main**.

$\mathsf{HCat}$

# Return of the HCat



# Back To The Code

Now that we understand how to write code that has side effects and interacts with the real world, let's put it to practice with an **MVP**:

# Back To The Code

Now that we understand how to write code that has side effects and interacts with the real world, let's put it to practice with an **MVP**:

---

```
module Main where

main :: IO ()
main = readFile "example.txt" >>= putStrLn
```

---

# The M-est of MVPs

Success! we can read a file and print it out to the screen!

## The M-est of MVPs

Success! we can read a file and print it out to the screen!

...but only a single hard-coded file

# The M-est of MVPs

**Success!** we can read a file and print it out to the screen!

... but only a single hard-coded file

... and it's not actually paginated

# The M-est of MVPs

**Success!** we can read a file and print it out to the screen!

... but only a single hard-coded file

... and it's not actually paginated

... or formatted for our terminal window

# The M-est of MVPs

**Success!** we can read a file and print it out to the screen!

... but only a single hard-coded file

... and it's not actually paginated

... or formatted for our terminal window

Let's take one problem at a time

# Getting Into Arguments



we need to deal with arguments

# Getting Into Arguments

We can use `getArgs` to get command line arguments but we'll need to deal with user error.

# Getting Into Arguments

We can use `getArgs` to get command line arguments but we'll need to deal with user error.

---

```
module HCatArgs where
  import System.Environment

  targetFileName :: IO FilePath
  targetFileName = do
    args <- getArgs
    case args of
      [filename] ->
        pure filename
      _otherwise ->
        ioError $ userError "please provide a single filename"

  main :: IO ()
  main = do
    contents <- readFile =<< targetFileName
    putStrLn contents
```

---

# Error Handling in IO Actions

Dealing with errors in IO actions can be complicated because there are a lot of options:

# Error Handling in IO Actions

Dealing with errors in IO actions can be complicated because there are a lot of options:

- ▶ Plain IO Errors
- ▶ Using Either or Maybe values for failure
- ▶ Custom exceptions
- ▶ Monad Transformers

# Error Handling in IO Actions

Dealing with errors in IO actions can be complicated because there are a lot of options:

- ▶ Plain IO Errors
- ▶ Using Either or Maybe values for failure
- ▶ Custom exceptions
- ▶ Monad Transformers

**Opinion:** Getting too fancy too early will cause more problems than it solves. Start with the simplest thing that can possibly work.

# What About Libraries?

Why parse arguments directly instead of using a library?

# What About Libraries?

Why parse arguments directly instead of using a library?

- ▶ Handling arguments yourself is good practice while learning
- ▶ Some good libraries use language features you probably haven't learned yet

# Terminal Size

The size of our terminal will determine our page count. We can get the terminal size with the `tput` program on \*nix systems.

# Terminal Size

The size of our terminal will determine our page count. We can get the terminal size with the `tput` program on \*nix systems.

---

```
module HCat where
import System.Process
data TerminalDimension = TerminalLines | TerminalCols
data ScreenDimensions =
    ScreenDimensions {screenRows :: Int, screenColumns :: Int}

getTerminalSize :: IO ScreenDimensions
getTerminalSize = do
    termLines <- tput TerminalLines
    termCols <- tput TerminalCols
    pure ScreenDimensions
        { screenRows = termLines
        , screenColumns = termCols }

tput :: TerminalDimension -> IO Int
tput dimension = do
    outputData <- readProcess "tput" [cmd] ""
    pure . read . head . lines $ outputData
    where
        cmd = case dimension of
            TerminalLines -> "lines"
            TerminalCols -> "cols"
```

---

# Word Wrapping

Given the size of our terminal, we can wrap the text to fit.

# Word Wrapping

Given the size of our terminal, we can wrap the text to fit.

---

```
wordWrap :: Int -> String -> [String]
wordWrap lineLength lineText =
  case splitAt lineLength lineText of
    (fullLine, "") -> [fullLine]
    (hardwrappedLine, rest) ->
      let (nextLine, remainder) = softWrap hardwrappedLine
          in nextLine : wordWrap lineLength (remainder <> rest)
  where
    softWrap hardWrapped =
      let (rest, wrappedText) = break isSpace $ reverse hardWrapped
          in (reverse wrappedText, reverse rest)

main :: IO ()
main = do
  contents <- readFile <<< targetFileName
  termSize <- getTerminalSize
  let wrapped = wordWrap (screenColumns termSize) contents
  putStrLn $ unlines wrapped
```

---

# Stepping Back

# Stepping Back



Let's talk about Architecture

# A Tale of Two Word Wraps

In our earlier example, we built an IO action to fetch the terminal size, and passed the width into a **pure function** that handled word wrapping. Let's consider the alternative:

# A Tale of Two Word Wraps

In our earlier example, we built an IO action to fetch the terminal size, and passed the width into a **pure function** that handled word wrapping. Let's consider the alternative:

---

```
wordWrap :: String -> IO [String]
wordWrap lineText = do
    lineLength <- tput TerminalCols
    case splitAt lineLength lineText of
        (fullLine, "") ->
            pure [fullLine]
        (hardwrappedLine, rest) -> do
            let (nextLine, remainder) = softWrap hardwrappedLine
            wrappedRemainder <- wordWrap (remainder <>> rest)
            pure (nextLine : wrappedRemainder)
    where
        softWrap hardWrapped =
            let (rest, wrappedText) = break isSpace $ reverse hardWrapped
            in (reverse wrappedText, reverse rest)
```

---

# A Tale of Two Word Wraps

In our earlier example, we built an IO action to fetch the terminal size, and passed the width into a **pure function** that handled word wrapping. Let's consider the alternative:

---

```
wordWrap :: String -> IO [String]
wordWrap lineText = do
    lineLength <- tput TerminalCols
    case splitAt lineLength lineText of
        (fullLine, "") ->
            pure [fullLine]
        (hardwrappedLine, rest) -> do
            let (nextLine, remainder) = softWrap hardwrappedLine
            wrappedRemainder <- wordWrap (remainder <>> rest)
            pure (nextLine : wrappedRemainder)
    where
        softWrap hardWrapped =
            let (rest, wrappedText) = break isSpace $ reverse hardWrapped
            in (reverse wrappedText, reverse rest)
```

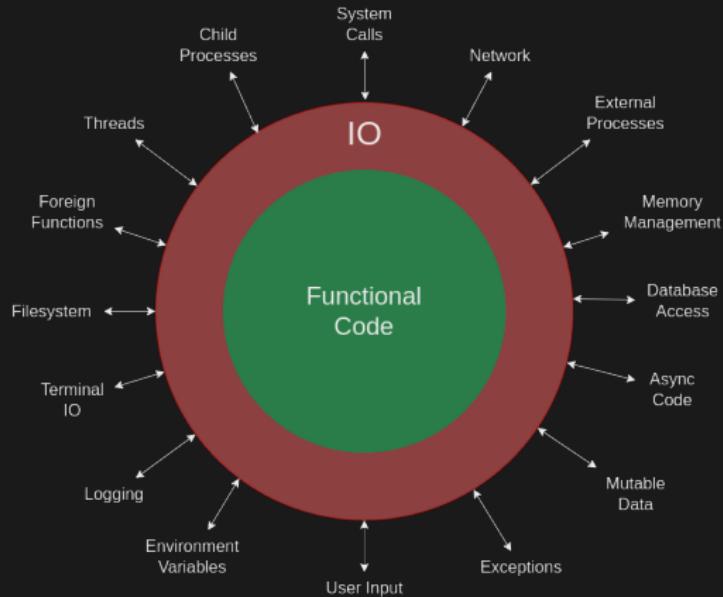
---

This might look **easier** at first. It hides details from the caller behind a smaller interface, but now it can't be used from any pure functions.

# The Lesson

As much as possible, have IO actions gather data then pass it into pure functions for computation.

# Procedural Shell, Functional Core



The "procedural shell, functional core" model is an over-simplification of a good guideline

# IO Actions are Like Layers



**IO Actions and pure functions more closely resemble a tree**

# Back to HCat

# Back to HCat



Back to our regularly scheduled HCat Presentation

# Pagination

Our pager has one big problem right now: It doesn't paginate.

# Pagination

Our pager has one big problem right now: It doesn't paginate.

---

```
paginate :: ScreenDimensions -> String -> [String]
paginate dimensions text = pages
  where
    rows = screenRows dimensions
    cols = screenColumns dimensions
    wrappedLines = concatMap (wordWrap cols) (lines text)
    pages = map (unlines . padTo rows) $ groupsOf rows wrappedLines
    padTo lineCount rowsToPad =
      take lineCount $ rowsToPad <> repeat ""
    groupsOf n elems
      | null elems = []
      | otherwise =
        let (hd, tl) = splitAt n elems
        in hd : groupsOf n tl
```

---

# Questions?