Your First Haskell Terminal Application

A Gentle Project-Based Introduction to Haskell

Rebecca Skinner

<2022-11-08 Tue>

Prelude

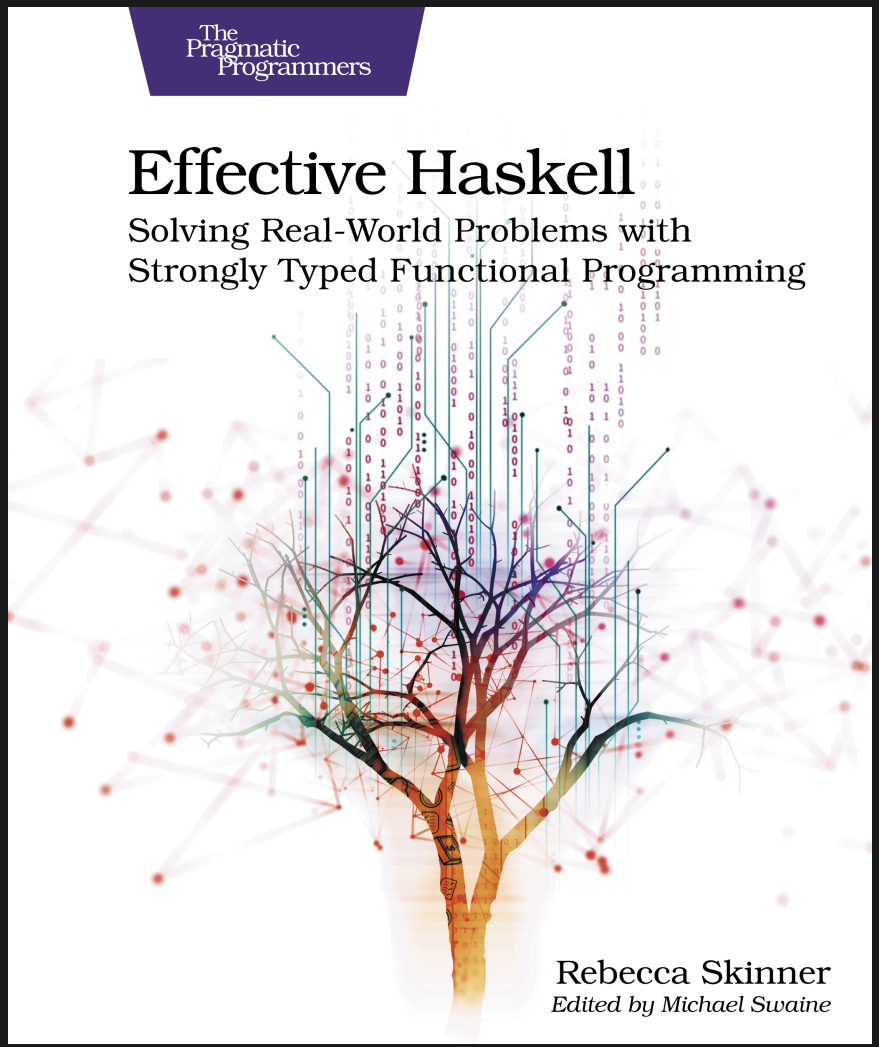
Hello, World

- 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/

About This Talk

- This talk is inspired by the first half of Effective Haskell.
- We'll focus on one happy path, there are tools and libraries that are good but won't be covered.
- This talk will give you a "lay of the land" but we'll leave out some details.

Effective Haskell



https://tinyurl.com/2744kfu7

Now in Beta!

Build Your Tools!

Build Your Tools!

HCat: A Haskell Pager

A pager is a command line tool that lets you view long documents one page at a time. Some examples you might be familiar with incldue:

HCat: A Haskell Pager

A pager is a command line tool that lets you view long documents one page at a time. Some examples you might be familiar with incldue:

- less
- more
- bat

HCat: A Haskell Pager

A pager is a command line tool that lets you view long documents one page at a time. Some examples you might be familiar with incldue:

- less
- more
- bat

We're going to build our own pager in Haskell. I'm calling mine hcat.

HCat: A Screenshot

```
{-# 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
```

Re-Inventing a Thousand Wheels

Why write something boring like a pager?

Re-Inventing a Thousand Wheels

Why write something boring like a pager?

- Teaches you how to do basic IO and work with the local system

Re-Inventing a Thousand Wheels

Why write something boring like a pager?

- Teaches you how to do basic IO and work with the local system
- Implementing word wrap and pagination will help you think about problems in a functional way

Re-Inventing a Thousand Wheels

Why write something boring like a pager?

- Teaches you how to do basic IO and work with the local system
- Implementing word wrap and pagination will help you think about problems in a functional way
- Building something similar to tools you use ever day will help you make good decisions, and give you something you can benefit from

Choosing Haskell for Developer Tooling

Why choose Haskell to build your developer tooling?

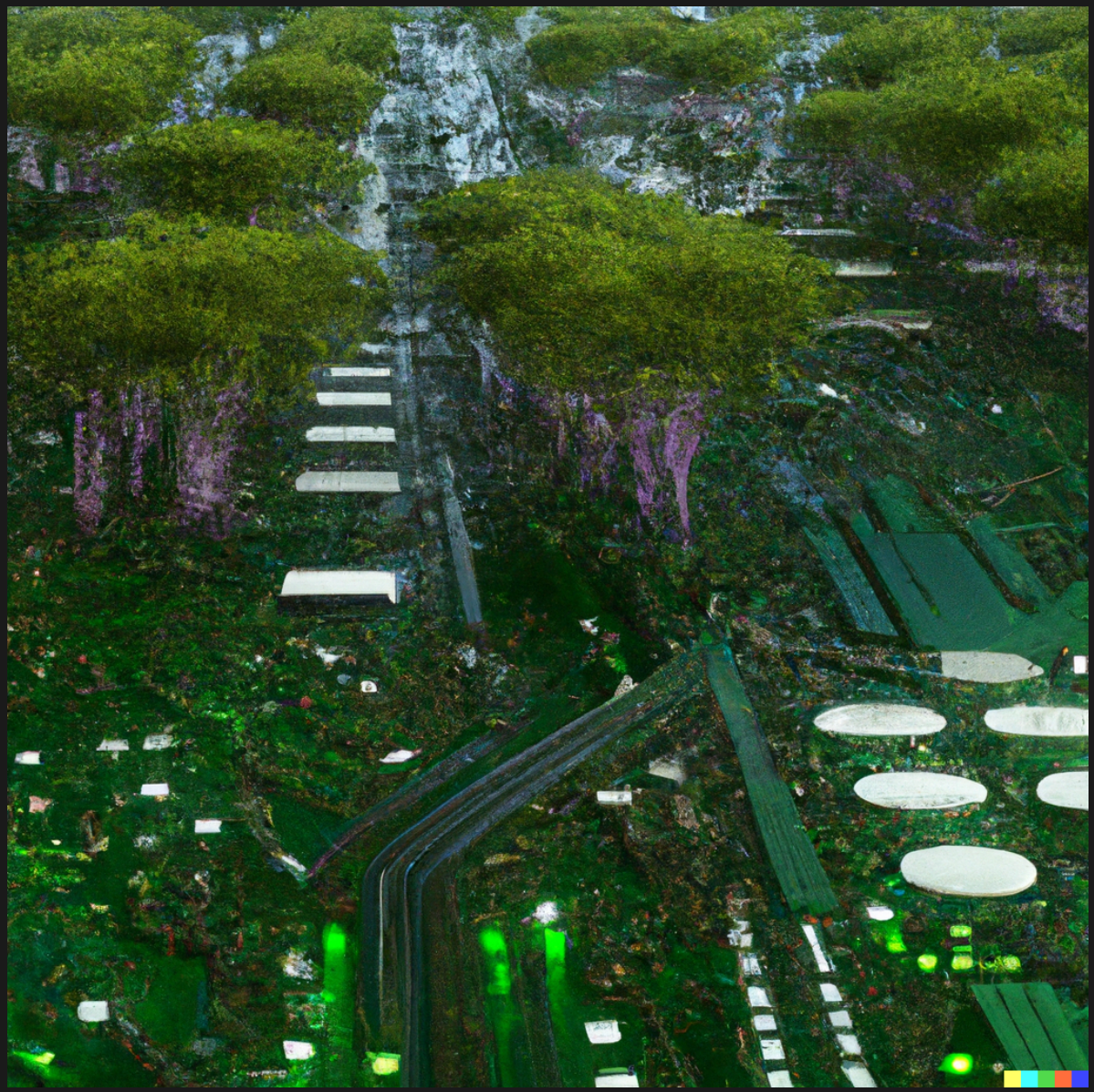
Choosing Haskell for Developer Tooling

Why choose Haskell to build your developer tooling?

- You want to learn Haskell, and building things for yourself is a good way to learn
- Haskell is a compiled language, so you don't have to deal with the startup times of a JIT language like Java
- Haskell is garbage collected (fewer memory errors)
- Haskell is statically typed so you'll have fewer runtime errors
- Haskell has good performance
  - Frequently, but not always slower than C and C++ programs
  - About as fast as Java programs while using a bit less memory, and having no JIT warmup cost
  - Nearly always faster than Python programs

Setting Up Your Environment

Setting Up Your Environment



Operating System

Haskell works on all of the major operating systems.

- Linux
  - x86 and arm. Best on x86
- macOS
  - Intel and Apple hardware are well supported
- Windows
  - x86. Native and WSL supported, but best with WSL.

Operating System

Haskell works on all of the major operating systems.

- Linux
  - x86 and arm. Best on x86
- macOS
  - Intel and Apple hardware are well supported
- Windows
  - x86. Native and WSL supported, but best with WSL.

I use NixOS. Haskell works great with NixOS, but it's a steep learning curve.

Operating System

Installing Haskell

The best way to install Haskell is with ghcup:

https://www.haskell.org/ghcup/

Installing Haskell

The best way to install Haskell is with ghcup:

https://www.haskell.org/ghcup/

- Managing Haskell with nixpkgs works well, but is a steep learning curve.
- Avoid using linux distro packages, homebrew, etc.
- Installing Haskell with Stack is no longer recommended, but you can install stack with ghcup if you like.

Configuring Your Editor

VSCode is the most popular editor for working with Haskell. It will help you set up additional tooling and offers the most IDE-like experience.

Emacs and vim also have good Haskell support, and can be configured with more IDE like features if you want.

Linting

hlint is the most popular Haskell linter by far. You should try to use hlint when you are developing.

- hlint will sometimes tell you things that you might not have learned yet, try not to worry about that too much.
- hlint sometimes makes bad suggestions, but most of the time it's suggestions are good and will help you learn to write better Haskell programs.
- hlint is very configurable, so you can tune it if there are things that you find annoying.

Pretty Printing

There are a lot of different pretty printers for Haskell. None of them do a perfect job, and they'll all make your code worse on occasion, so you should treat them as a tool to help you format your code, rather than something that should be enforced.

Pretty Printing

There are a lot of different pretty printers for Haskell. None of them do a perfect job, and they'll all make your code worse on occasion, so you should treat them as a tool to help you format your code, rather than something that should be enforced.

- Fourmolu is modestly configurable, and tends to work reliably. It's also fairly aggressive and is more likely to uglify your code.
- Stylish Haskell tends to lag a bit in supporting newer GHC versions, and can be a little flaky, but is less intrusive and it's style is a bit more idiomatic to classic haskellers

Pretty Printing

There are a lot of different pretty printers for Haskell. None of them do a perfect job, and they'll all make your code worse on occasion, so you should treat them as a tool to help you format your code, rather than something that should be enforced.

- Fourmolu is modestly configurable, and tends to work reliably. It's also fairly aggressive and is more likely to uglify your code.
- Stylish Haskell tends to lag a bit in supporting newer GHC versions, and can be a little flaky, but is less intrusive and it's style is a bit more idiomatic to classic haskellers

There are a bunch of other options you can explore if you want

Too Much Text

IDEs and Not-IDEs

The Haskell Language Server (HLS) provides the most IDE-like experience for working with Haskell. Other tools you might like are:

ghci	The GHC REPL
ghcid	The ghci repl as a deamon, with some extra features
halfsp	A minimal Haskell language server with good performance
haskell-mode	Emacs major mode with good REPL integration
hasktags	ctags and etags support for navigating files

Picking a GHC Version

- As a rule of thumb, staying one version behind the newest release is a safe strategy for getting new features in a timely manner while avoiding broken packages.
- Tools tend to take longer to update than libraries, so if you are flexible about what tools you want you can typically upgrade sooner.

Getting Help

Browse Documentation on Hackage

https://hackage.haskell.org

Hot Tip When browsing documentaiton the s key to bring up a box that will let you search for functions by name or type

Hot Tip The source links take you to a page where you can browse the source of packages. The source is typically hyperlinked to source in other packages, so you can follow definitions to other code from your browser

Hot Tip In search engines like DuckDuckGo and Kagi you can use !hackage <package> to search for a package.

Browse Documentation on Hackage

img/hackage-demo.webm

Search Libraries with Hoogle

img/hoogle-demo.webm

Starting a New Project

Starting a New Project

Haskell Packaging Overview

You can write, build, and run a Haskell program without any special build tools:

```
ghc Hello.hs && ./Hello
```

Haskell Packaging Overview

You can write, build, and run a Haskell program without any special build tools:

```
ghc Hello.hs && ./Hello
```

Usually you want to use a tool like cabal to make it easier to manage dependencies, build your program, and run tests.

Picking a Build Tool

There are three popular ways to build Haskell programs:

Picking a Build Tool

There are three popular ways to build Haskell programs:

- **cabal** The default tool for building packages with GHC.
- **stack** Another popular tool. It aims to make dependency management easier.
- **nix** Uses cabal under the hood. Provides reproducibility.

Picking a Build Tool

There are three popular ways to build Haskell programs:

- **cabal** The default tool for building packages with GHC.
- **stack** Another popular tool. It aims to make dependency management easier.
- **nix** Uses cabal under the hood. Provides reproducibility.

cabal is a good default choice and we'll stick with it in this talk.

Cabal Basics

`cabal init`	Start a new project
`cabal build`	Compiler your project
`cabal exec`	Run your project
`cabal repl`	Load your code into an interactive environment

The Cabal File

```
cabal-version:       2.4
name:                hcat
version:             0.1.0.0

library
  hs-source-dirs:      src
  exposed-modules:     HCat
  build-depends:       base, bytestring, text
                     , process, directory, time
  default-language:    Haskell2010

executable hcat
  hs-source-dirs:      app
  main-is:             Main.hs
  build-depends:       base, hcat
  default-language:    Haskell2010
```

App, Src, and Test

- Most haskell applications have a very minimal application. Often, the executable is a single file named `Main.hs`
- The application logic is typically all in a library that lives in the same project along with the executable
- Unit tests are in a `test` directory and cover the library code

Managing Dependencies

Don't worry too much about things like version boundaries when you are just getting started. Focus on writing programs using only a few popular libraries, and stick with the defaults where you can. You'll need to be more careful in the future, but for now it's not worth being concerned with.

Putting the M in MVP

Read File, Write File

Haskell can look a lot like other languages you might have seen. Here's a working version of hcat:

```
getFileNameFrom arguments = do {
  when (length arguments <= 0) do {
    fail "empty args";
  };
  return (head arguments);
}

main = do {
  arguments <- getArgs;
  fileName <- getFileNameFrom arguments;
  fileContents <- readFile fileName;
  putStr fileContents;
  return ();
}
```

Read File, Write File

But this is another working version:

```
main = getArgs >>= readFile . head >>= putStr
```

Read File, Write File

But this is another working version:

```
main = getArgs >>= readFile . head >>= putStr
```

Programs you read are more likely to look like this example than the previous one.

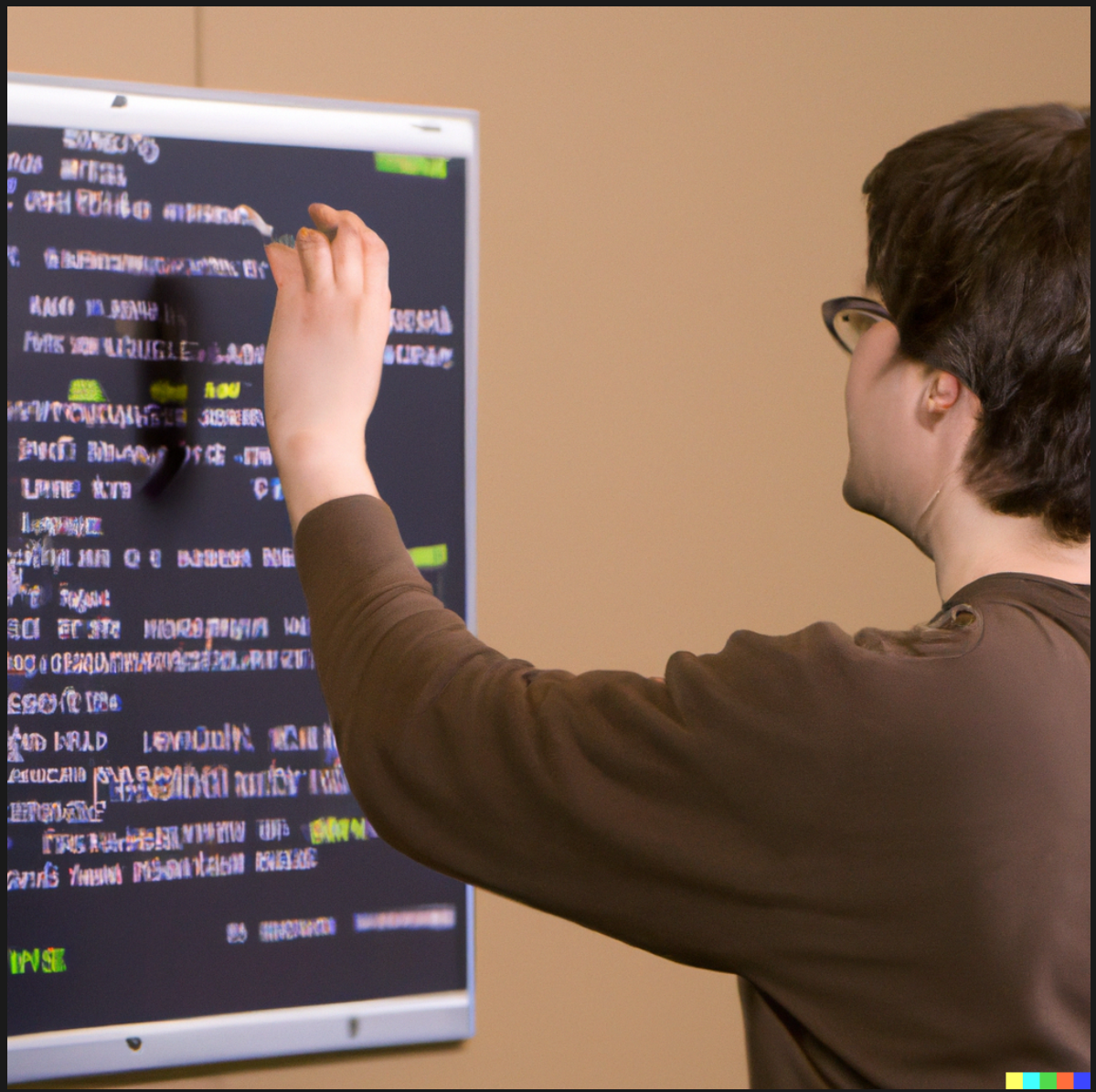
Read File, Write File

The lesson:

Read File, Write File

The lesson:
When you are first getting started, don't worry too much about idiomatic code. Haskell has a lot of ways to do things, and the learning curve is a little steep at first. Focus on getting things working even if they aren't the "right way". Refactor as you learn more.

Zoom, Enhance, Refactor



Zoom, Enhance, Refactor

Let's give this another shot. How might a reasonable version of our program look?

Zoom, Enhance, Refactor

Let's give this another shot. How might a reasonable version of our program look?

```
module HCat where
import System.Environment (getArgs)

filenameArgument :: IO FilePath
filenameArgument = do
  allArguments <- getArgs
  case allArguments of
    [fileName] ->
      pure fileName
    _otherwise ->
      fail "Please provide one file name"

hcat :: IO ()
hcat = do
  fileName <- filenameArgument
  fileContents <- readFile fileName
  putStr fileContents
```

That's No Moon: Working With IO

A Tale of Two Languages

At first glance, our first hcat implementation seems straightforward, but writing your first Haskell program that does substantial IO can be tricky.

A Tale of Two Languages

In this function we're only using let and in

```
reverseSentence sentence =
  let
    wordsInReverse =
      reverse (words sentence)
  in unwords wordsInReverse
```

But in this function, we're also using do and <-

```
printSentenceBackwards = do
  sentence <- getLine
  let
    reversed =
      reverseSentence sentence
  putStrLn reversed
```

A Tale of Two Languages

This is a pure function

```haskell
reverseSentence :: String -> String
reverseSentence sentence =
  let
    wordsInReverse =
      reverse (words sentence)
  in unwords wordsInReverse
```

This is an IO Action

```haskell
printSentenceBackwards :: IO ()
printSentenceBackwards = do
  sentence <- getLine
  let
    reversed =
      reverseSentence sentence
  putStrLn reversed
```

We put code in your code

An IO Action is a program that will be executed by the Haskell runtime when a user launches the program. All executable Haskell programs ultimately work by running a single IO action named main.

We put code in your code

An IO Action is a program that will be executed by the Haskell runtime when a user launches the program. All executable Haskell programs ultimately work by running a single IO action named main.

Haskell is a pure functional language that we can use to write impure effectful programs in the IO Action language:

We put code in your code

An IO Action is a program that will be executed by the Haskell runtime when a user launches the program. All executable Haskell programs ultimately work by running a single IO action named main.

Haskell is a pure functional language that we can use to write impure effectful programs in the IO Action language:

- Writing IO programs that run pure functional code
- Creating new IO programs by running one right after another
- Runing one IO program and passing it's output in as the input to another IO program
- Writing an IO program that executes many other IO programs as it runs

How do you do it?

Haskell's do notation gives us special syntax for working with things like IO actions. You can think of a do block as special syntax to embed a program written in the IO Action language directly into your pure functional program. You can even interleave pure functional Haskell code with code written in the IO Action language.

How do you do it?

Code inside of do blocks can really look a lot like impure procedural programming languages. Let's look at an example from Effective Haskell:

```haskell
editDistance :: Text -> Text -> Int
editDistance stringA stringB = runST $ do
  let
    aLen = T.length stringA
    bLen = T.length stringB
    as = zip [1..] (T.unpack stringA)
    bs = zip [1..] (T.unpack stringB)
    lookupIndex x y = (y * (aLen + 1)) + x
  cache <- MVec.new $ (aLen + 1) * (bLen + 1)
  for_ [0..aLen] $ \idx -> MVec.write cache (lookupIndex idx 0) idx
  for_ [0..bLen] $ \idx -> MVec.write cache (lookupIndex 0 idx) idx
  for_ as $ \(idxA, charA) -> do
    for_ bs $ \(idxB, charB) -> do
      let
        editCost = if charA == charB then 0 else 1
      insertCost <- (1 +) <$> MVec.read cache (lookupIndex (idxA - 1) idxB)
      deleteCost <- (1 +) <$> MVec.read cache (lookupIndex idxA (idxB - 1))
      swapCost   <- (editCost +) <$> MVec.read cache (lookupIndex (idxA - 1) (idxB - 1))
      MVec.write cache (lookupIndex idxA idxB) $ min swapCost (min insertCost deleteCost)
  MVec.read cache $ lookupIndex aLen bLen
```

Slings and Arrows

It's important to remember that things "inside" of an IO Action don't really exist yet when you are writing pure functional code. They won't exist until the IO Action is run. When an IO Action program runs and returns a value, you can't use that value in your pure code because it hasn't been computed yet.

Slings and Arrows

It's important to remember that things "inside" of an IO Action don't really exist yet when you are writing pure functional code. They won't exist until the IO Action is run. When an IO Action program runs and returns a value, you can't use that value in your pure code because it hasn't been computed yet.

But. . . ..

Slings and Arrows

It's important to remember that things "inside" of an IO Action don't really exist yet when you are writing pure functional code. They won't exist until the IO Action is run. When an IO Action program runs and returns a value, you can't use that value in your pure code because it hasn't been computed yet.

But. . . ..

IO Action programs are regular values, and you can return them, and work with them like any other kind of value.

Slings and Arrows

```haskell
aBunchOfPrintStatements :: Int -> [IO ()]
aBunchOfPrintStatements count =
  [print x | x <- [0..count]]

printTenTimes :: IO ()
printTenTimes =
  for_ (aBunchOfPrintStatements 10) $ \statement -> do
    putStrLn "running a statement..."
    statement
```

Putting It All Together

Putting It All Together

Putting It All Together

At the top level of our program we have an IO action that runs several other IO actions.

```haskell
runHCat :: IO ()
runHCat = do
  targetFilePath <- handleArgs
  contents <- TextIO.readFile targetFilePath
  termSize <- getTerminalSize
  hSetBuffering stdout NoBuffering
  finfo <- fileInfo targetFilePath
  let pages = paginate termSize finfo contents
  showPages pages
```

Putting It All Together

Some IO Actions run other programs, like this one that calls the tput command to get the dimensions of the terminal.

```haskell
getTerminalSizeBind :: IO ScreenDimensions
getTerminalSizeBind =
  case System.Info.os of
    "darwin" -> tputScreenDimensions
    "linux" -> tputScreenDimensions
    _other -> pure $ ScreenDimensions 25 80
  where
    tputScreenDimensions = do
      lines <- Process.readProcess "tput" ["lines"] ""
      cols <- Process.readProcess "tput" ["cols"] ""
      let lines' = read $ init lines
          cols'  = read $ init cols
      pure $ ScreenDimensions lines' cols'
```

Putting It All Together

IO Actions can also be run recursively. Here we run actions recursively to scroll through a list of pages:

```haskell
getContinue :: IO ContinueCancel
getContinue = do
  hSetBuffering stdin NoBuffering
  hSetEcho stdin False
  keyInput <- getChar
  case keyInput of
    ' ' -> return Continue
    'q' -> return Cancel
    _   -> getContinue

showPages :: [Text.Text] -> IO ()
showPages [] = return ()
showPages (page:pages) = do
  clearScreen
  TextIO.putStr page
  shouldContinue <- getContinue
  case shouldContinue of
    Continue -> showPages pages
    Cancel   -> return ()
```

There's Still Room For Pure Code

Even though we're doing a lot of IO in our program, there's still plenty of room for pure functions, like this pagination function:

```haskell
paginate :: ScreenDimensions -> FileInfo -> Text.Text -> [Text.Text]
paginate (ScreenDimensions rows cols) finfo text =
  let
    rows' = rows - 1
    wrappedLines = concatMap (wordWrap cols) (Text.lines text)
    pages = map (Text.unlines . padTo rows') $ groupsOf rows' wrappedLines
    pageCount = length pages
    statusLines = map (formatFileInfo finfo cols pageCount) [1..pageCount]
  in zipWith (<>) pages statusLines
  where
    padTo :: Int -> [Text.Text] -> [Text.Text]
    padTo lineCount rowsToPad =
      take lineCount $ rowsToPad <> repeat ""
```

Finding The Patterns

Finding The Patterns

In these examples, notice that the IO actions tended to run a lot of other IO actions. They included a little bit of pure code, but not very much. That's an intentional pattern in many Haskell programs:

- Write small IO actions that do a particular thing
- Keep the bulk of your logic in pure functions
- Have IO actions gather all of their inputs, then pass them to pure functions.
- Avoid mostly pure functions "that can have a little IO, as a treat"

What's Next?

Build Your Own!

We've only looked at a few small parts of hcat but you can try to build your own! If you want more guidance, check out Chapter 8 of Effective Haskell.

Questions?