Haskell With Chris Allen

My name is Chris. I teach Haskell to people that are new to programming and as well as long-time coders. Haskell is a general purpose programming language that is most useful to mere mortals.

I'm going to show you how to write a package in Haskell and interact with the code inside of it.

Installing tools for writing Haskell code

The most popular compiler for Haskell is `GHC` and you use `Cabal` alongside `GHC` to manage projects and their dependencies. Packaging itself is part of `GHC` via `ghc-pkg` .

To get GHC and Cabal installed, use one of the following links:

- Windows: https://www.haskell.org/downloads/windows (https://www.haskell.org/downloads/windows)
- OS X: https://www.haskell.org/downloads/osx (https://www.haskell.org/downloads/osx)
- Linux: https://www.haskell.org/downloads/linux (https://www.haskell.org/downloads/linux)

After you've finished the install instructions, `ghc` , `cabal` , and `ghci` should all be in your path. `ghci` is the REPL (read-eval-print loop) for Haskell, though as often as not, you'll use `cabal repl` to invoke a REPL that is aware of your project and its dependencies.

What we're going to make

We're going to write a little csv parser for some baseball data. I don't care a whit about baseball, but it was the best example of free data I could find.

Project layout

There's not a prescribed project layout, but there are a few guidelines I would advise following.

One is that Edward Kmett's lens library (https://github.com/ekmett/lens) is not only a fantastic library in its own right, but is also a great resource for people wanting to see how to structure a Haskell project, write and generate `Haddock` documentation, and organize your namespaces. Kmett's library follows Hackage guidelines (http://hackage.haskell.org/packages/) on what namespaces and categories to use for his libraries.

There is an alternative namespacing pattern demonstrated by Pipes, a streaming library (http://hackage.haskell.org/package/pipes). It uses a top-level eponymous namespace. For an example of another popular project you could also look at Pandoc (https://github.com/jgm/pandoc/) for examples of how to organize non-trivial Haskell projects.

Once we've finished laying out our project, it's going to look like this:

```
$ tree
.
├── LICENSE
├── Setup.hs
├── cabal.sandbox.config
├── bassbull.cabal
├── src
│   ├── Main.hs
4 directories, 7 files
```

Here's where each of those files are going to come from.

- We'll use `cabal init` to generate `LICENSE`, `Setup.hs`, and `bassbull.cabal`.
- When we initialize our cabal sandbox, later it will generate `cabal.sandbox.config`.
- You'll need to `mkdir src` and `touch src/Main.hs` for your `src/Main.hs` file to exist and that's where we'll be putting our code after we've made our project directory.

Ordinarily I'd structure things a little more, but there isn't a lot to this project. Onward!

Getting your project started

First we're going to make our directory for our project wherever we tend to stash our work. If we're on a Unix-alike, that'll look something like:

```
$ mkdir bassbull
$ cd bassbull
```

Now we're going to download our test data now that we're inside the directory of our `bassbull` project.

You can download the data from here (https://raw.githubusercontent.com/bitemyapp/csvtest/master/batting.csv). If you want to download it via the terminal on a Unix-alike (Mac, Linux, BSD, etc) you can do so via:

```
$ curl -0 https://raw.githubusercontent.com/bitemyapp/csvtest/master/batting.csv
```

It should be about 2.3 MB when it's all said and done.

Having done that, we're now going to use `Cabal`, our GHC Haskell dependency manager and build tool, to create some initial files for us. You have a couple options here. You can use the interactive helper or you can define everything non-interactively in one go.

To do it interactively:

```
$ cabal init
```

And the command I used to do it non-interactively (edit as appropriate for your project):

```
$ cabal init -n -l BSD3 --is-executable --language=Haskell2010 -u bitemyapp.com \
  -a 'Chris Allen' -c Data -s 'Processing some CSV data' -p bassbull
```

Before we start making changes, I'm going to init my version control (git, for me) so I can track my changes and not lose any work.

```
$ git init
$ git add .
$ git commit -am "Initial commit"
```

I'm also going to add the gitignore from Github's gitignore repository plus some additions for Haskell so we don't accidentally check in unnecessary build artifacts or other things inessential to the project.

This should go into a file named `.gitignore` at the top level of your bassbull project.

```
dist
cabal-dev
*.o
*.hi
*.chi
*.chs.h
.virtualenv
.hpc
.hsenv
.cabal-sandbox/
cabal.sandbox.config
cabal.config
*.prof
*.hp
*.aux
```

You might be wondering why we're telling `git` to ignore something called a "cabal sandbox". Cabal, unlike the package managers in other language ecosystems, requires direct and transitive dependencies to have compatible versions. For contrast, Maven will use the "closest" version. To avoid packages having conflicts, Cabal introduced sandboxes which let you do builds of your projects in a way that doesn't use your user package-db. Your user package-db is global to all your builds on your user account and this is almost never what you want. This is not dissimilar from `virtualenv` in the Python community. The `.cabal-sandbox` directory is where our build artifacts will go when we build our project or test cases. We don't want to version control that as it would bloat the git repository and doesn't need to be version controlled.

Editing the Cabal file

First we need to fix up our `cabal` file a bit. Mine is named `bassbull.cabal` and is in the top level directory of the project.

Here's what I changed my `cabal` file to:

```
name:                bassbull
version:             0.1.0.0
synopsis:            Processing some csv data
description:         Baseball data analysis
homepage:            bitemyapp.com
license:             BSD3
license-file:        LICENSE
author:              Chris Allen
maintainer:          cma@bitemyapp.com
copyright:           2014, Chris Allen
category:            Data
build-type:          Simple
cabal-version:       >=1.10

executable bassbull
  ghc-options:         -Wall
  hs-source-dirs:      src
  main-is:             Main.hs
  build-depends:       base >= 4.7 && <5,
                       bytestring,
                       vector,
                       cassava
  default-language:    Haskell2010
```

A few notable changes:

- Set the description so Cabal would stop squawking about it.
- Set `hs-source-dirs` to `src` so Cabal knows where my modules are.
- Added a named executable stanza to the Cabal file so I can build a binary by that name and run it.
- Set `main-is` to `Main.hs` in the executable stanza so the compiler knows what main function to use for that binary.
- Set `ghc-options` to `-Wall` so we get the *rather* handy warnings GHC offers on top of the usual type checking.
- Added the libraries our project will use to `build-depends` .

Building and interacting with your program

The contents of `src/Main.hs` :

```haskell
module Main where

main = putStrLn "hello"
```

One thing to note is that for a module to work as a `main-is` target for GHC, it must have a function named `main` and itself be named `Main` . Most people make little wrapper `Main` modules to satisfy this, sometimes with argument parsing and handling done via libraries like optparse-applicative (https://github.com/pcapriotti/optparse-applicative).

For now, we've left Main very simple, making it just a `putStrLn` of the string `"Hello"` . To validate that everything is working, let's build and run this program.

We're going to create a Cabal sandbox so that our dependencies are isolated to this project.

```
$ cabal sandbox init
```

Then we install our dependencies. These should get installed into the Cabal sandbox package-db now that our sandbox has been created. Otherwise they'd get installed into our user package-db located in our home directory, which would be global to all the projects on our current user account. This can take some time on first run.

```
$ cabal install --only-dependencies
```

Now we're going to build our project.

```
$ cabal build
```

If this succeeds, we should get a binary named `bassbull` in `dist/build/bassbull`. To run this, do the following.

```
$ ./dist/build/bassbull/bassbull
hello
$
```

If everything is in place, let's move onto writing a little csv processor.

Writing a program to process csv data

One thing to note before we begin is that you can fire up a project-aware Haskell REPL using `cabal repl`. The benefit of doing so is that you can write and type-check code interactively as you explore new and unfamiliar libraries or just to refresh your memory about existing code.

You can do so by running it in your shell like so:

```
$ cabal repl
```

If you do, you should see a bunch of stuff about loading packages installed for the project and then a `Prelude Main>` prompt.

```
[1 of 1] Compiling Main             ( Main.hs, interpreted )
Ok, modules loaded: Main.
Prelude Main>
```

Now we can load our `src/Main.hs` in the REPL.

```
$ cabal repl
Preprocessing executable 'bassbull' for bassbull-0.1.0.0...
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package array-0.5.0.0 ... linking ... done.
Loading package deepseq-1.3.0.2 ... linking ... done.
Loading package bytestring-0.10.4.0 ... linking ... done.
Loading package containers-0.5.5.1 ... linking ... done.
Loading package text-1.2.0.0 ... linking ... done.
Loading package hashable-1.2.2.0 ... linking ... done.
Loading package scientific-0.3.3.2 ... linking ... done.
Loading package attoparsec-0.12.1.2 ... linking ... done.
Loading package blaze-builder-0.3.3.4 ... linking ... done.
Loading package unordered-containers-0.2.5.1 ... linking ... done.
Loading package primitive-0.5.4.0 ... linking ... done.
Loading package vector-0.10.12.1 ... linking ... done.
Loading package cassava-0.4.2.0 ... linking ... done.
[1 of 1] Compiling Main             ( src/Main.hs, interpreted )

src/Main.hs:3:1: Warning:
    Top-level binding with no type signature: main :: IO ()
Ok, modules loaded: Main.
*Main> :load src/Main.hs
[1 of 1] Compiling Main             ( src/Main.hs, interpreted )

src/Main.hs:3:1: Warning:
    Top-level binding with no type signature: main :: IO ()
Ok, modules loaded: Main.
*Main>
```

Becoming comfortable with the REPL can be a serious boon to productivity. There is editor integration for those that want it as well.

Now we're going to update our `src/Main.hs`. Our goal is to read a CSV file into a `ByteString` (basically a byte vector), parse the `ByteString` into a `Vector` of tuples, and sum up the "at bats" column.

```haskell
module Main where

import qualified Data.ByteString.Lazy as BL
import qualified Data.Vector as V
-- from cassava
import Data.Csv

-- a simple type alias for data
type BaseballStats = (BL.ByteString, Int, BL.ByteString, Int)

main :: IO ()
main = do
  csvData <- BL.readFile "batting.csv"
  let v = decode NoHeader csvData :: Either String (V.Vector BaseballStats)
  let summed = fmap (V.foldr summer 0) v
  putStrLn $ "Total atBats was: " ++ (show summed)
  where summer (name, year, team, atBats) n = n + atBats
```

Let's break down this code.

```haskell
import qualified Data.ByteString.Lazy as BL
import qualified Data.Vector as V
-- from cassava
import Data.Csv
```

First, we're importing our dependencies. Qualified imports let us give names to the namespaces we're importing and use those names as a prefix, such as `BL.ByteString`. This is used to refer to values and type constructors alike. In the case of `import Data.Csv` where we didn't qualify the import (with `qualified`), we're bringing everything from that module into scope. This should be done only with modules that have names of things that won't conflict with anything else. Other modules like `Data.ByteString` and `Data.Vector` have a bunch of functions that are named identically to functions in the `Prelude` and should be qualified.

```haskell
-- a simple type alias for data
type BaseballStats = (BL.ByteString, Int, BL.ByteString, Int)
```

Here we're creating a type alias for `BaseballStats`. I made it a type alias for a few reasons. One is so I could put off talking about algebraic data types! I made it a type alias of the 4-tuple specifically because the Cassava library already understands how to translate CSV rows into tuples and our type here will "just work" as long as the columns that we say are `Int` actually are parseable as integral numbers. Haskell tuples are allowed to have heterogenous types and are defined primarily by their length. The parentheses and commas are used to signify them. For example, `(a, b)` would be both a valid value and type constructor for referring to 2-tuples, `(a, b, c)` for 3-tuples, and so forth.

```haskell
main :: IO ()
main = do
  csvData <- BL.readFile "batting.csv"
```

We need to read in a file so we can parse our CSV data. We called the lazy `ByteString` namespace

`BL` using the `qualified` keyword in the import. From that namespace we used `BL.readFile` which has type `FilePath -> IO ByteString`. You can read this in English as
`I take a FilePath as an argument and I return a ByteString after performing some side eff`

You can see the type of `BL.readFile` here (http://hackage.haskell.org/package/bytestring-0.10.4.0/docs/Data-ByteString-Lazy.html#v:readFile).

We're binding over the `IO ByteString` that `BL.readFile "batting.csv"` returns. `csvData` has type `ByteString` due to binding over `IO`. Remember our tuples that we signified with parentheses earlier? Well, `()` is a sort of tuple too, but it's the 0-tuple! In Haskell we usually call it unit. It can't contain anything; it's a type that has a single value - `()`, that's it. It's often used to signify we don't return anything. Since there's usually no point in executing functions that don't return anything, `()` is often wrapped in `IO`. Printing strings are a good example of the result type `IO ()` as they do their work and return nothing. In Haskell you can't actually "return nothing;" the concept doesn't even make sense. Thus we use `()` as the idiomatic "I got nothin' for ya" type and value. Usually if something returns `()` you won't even bother to bind to a name, you'll just ignore it.

```
let v = decode NoHeader csvData :: Either String (V.Vector BaseballStats)
```

`v` has the type you see at the right with the type assignment operator `::` I'm assigning the type to dispatch the typeclass that `decode` uses to parse csv data. See more about the typeclass cassava uses for parsing csv data here (http://hackage.haskell.org/package/cassava-0.4.2.0/docs/Data-Csv.html#t:FromRecord).

In this case, because I defined a `type` alias of a tuple for my record, I get my parsing code for free (already defined for tuples, `bytestring`, and `Int`).

```
let summed = fmap (V.foldr summer 0) v
```

Here we're using a `let` expression to bind the expression `fmap (V.foldr summer 0) v` to the name `summed` so that the expressions that follow it can refer to `summed` without repeating all the same code.

First we fmap over the `Either String (V.Vector BaseballStats)`. This lets us apply `(V.foldr summer 0)` to `V.Vector BaseballStats`. We partially applied the `Vector` folding function `foldr` to the summing function and the number `0`. The number `0` here is our "start" value for the fold. Generally in Haskell we don't use recursion directly. Instead in Haskell we use higher order functions and abstractions, giving names to common things programmers do in a way that lets us be more productive. One of those very common things is folding data. You're going to see examples of folding and the use `fmap` from `Functor` in a bit.

We say `V.foldr` is partially applied because we haven't applied all of the arguments yet. Haskell has something called currying built into all functions by default which lets us avoid some tedious work that would require a "Builder" pattern in languages like Java. Unlike previous code samples, these examples are using my interactive `ghci` REPL.

```
-- Person is a product/record, if that
-- is confusing think "struct" but better.
Prelude> data Person = Person String Int String deriving Show

Prelude> :type Person
Person :: String -> Int -> String -> Person

Prelude> :t Person "Chris" 415
Person "Chris" 415 :: String -> Person

Prelude> :t Person "Chris" 415 "Allen"
Person "Chris" 415 "Allen" :: Person

Prelude> let namedChris = Person "Chris"
Prelude> namedChris 415 "Allen"
Person "Chris" 415 "Allen"

Prelude> Person "Chris" 415 "Allen"
Person "Chris" 415 "Allen"
```

This lets us apply some, but not all, of the arguments to a function and pass around the result as a function expecting the rest of the arguments.

Fully explaining the `fmap` in `let summed = fmap (V.foldr summer 0) v` would require explaining `Functor`. I don't want to belabor specific concepts *too* much, but I think a quick demonstration of `fmap` and `foldr` would help here. This is also a transcript from my interactive `ghci` REPL. I'll explain Either, Right, and Left after the REPL sample. The `:type` or `:t` command is a command to my `ghci` REPL, not part of the Haskell language. It's a way to request the type of an expression.

```
Prelude> let v = Right 1 :: Either String Int
Prelude> let x = Left "blah" :: Either String Int

Prelude> :t v
v :: Either String Int
Prelude> :t x
x :: Either String Int

Prelude> let addOne x = x + 1
<interactive>:4:12: Warning:
    This binding for ‘x’ shadows the existing binding
      defined at <interactive>:3:5
Prelude> addOne 2
<interactive>:5:1: Warning:
    Defaulting the following constraint(s) to type ‘Integer’
      (Show a0) arising from a use of ‘print’ at <interactive>:5:1-8
      (Num a0) arising from a use of ‘it’ at <interactive>:5:1-8
    In a stmt of an interactive GHCi command: print it
3

Prelude> fmap addOne v
Right 2
Prelude> fmap addOne x
Left "blah"
```

`Either` in Haskell is used to signify cases where we might get values of one of two possible types. `Either String Int` is a way of saying, "you'll get either a `String` or an `Int` ". This is an example of sum types. You can think of them as a way to say `or` in your type, where a `struct` or `class` would let you say `and` . `Either` has two constructors, `Right` and `Left` . Culturally in Haskell `Left` signifies an "error" case. This is partly why the `Functor` instance for `Either` maps over the `Right` constructor but not the `Left` . If you have an error value, you can't keep applying your happy path functions. In the case of `Either String Int` , `String` would be our error value in a `Left` constructor and `Int` would be the happy-path "yep, we're good" value in the `Right` constructor. Also, Haskell has type inference. You don't have to declare types explicitly like I did in the example from my REPL transcript - I did so for the sake of explicitness.

`Either` isn't the only type we can map over.

```
Prelude> let myList = [1, 2, 3] :: [Int]
Prelude> fmap addOne myList
[2,3,4]
Prelude> let multTwo x = x * 2
Prelude> fmap multTwo myList
[2,4,6]
```

Here we have the list type, signified using the `[]` brackets and whatever type is inside in our list, in this case `Int` . With `Either` we have two possible types and `Functor` only lets us map over one of them, so the `Functor` instance for `Either` only applies our function over the happy path values. With the type `[a]` there's only one type inside of it, so it'll get applied regardless...or will it? What if I have an empty list?

```
Prelude> fmap multTwo []
[]
Prelude> fmap addOne []
[]
```

Conveniently not only does `fmap` let us avoid manually pattern matching the `Left` and `Right` cases of `Either` , but it lets us not bother to manually recurse our list or pattern-match the empty list case. This helps us prevent mistakes as well as clean up and abstract our code. In a less happy alternate universe, we would've had to write the following code, written in typical code file style rather than for the REPL this time:

```
addOne :: Int -> Int
addOne x = x + 1 -- at least we can abstract this out

incrementEither :: Either e Int -> Either e Int
incrementEither (Right numberWeWanted) = Right (addOne numberWeWanted)
incrementEither (Left errorString) = Left errorString
```

We use parens on the left-hand side here to pattern match at the function declaration level on whether our `Either e Int` is `Right` or `Left` . Parentheses wrap `(addOne numberWeWanted)` so we don't try to erroneously pass two arguments to `Right` when we mean to pass the result of applying `addOne` to `numberWeWanted` , to `Right` . If our value is `Right 1` this is returning `Right (addOne 1)` which reduces to `Right 2` .

As we process the CSV data we're going to be doing so by *folding* the data. This is a general model for understanding how you process data that extends beyond specific programming languages. You might have seen `fold` called `reduce` . Here are some examples of folds and list/string concatenation in Haskell. We're switching back to REPL demonstration again.

```
Prelude> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b

Prelude> foldr (+) 0 [1, 2, 3]
6
Prelude> foldr (+) 1 [1, 2, 3]
7
Prelude> foldr (+) 2 [1, 2, 3]
8
Prelude> foldr (+) 2 [1, 2, 3, 4]
12

Prelude> :t (++)
(++) :: [a] -> [a] -> [a]

Prelude> [1, 2, 3] ++ [4, 5, 6]
[1,2,3,4,5,6]
Prelude> "hello, " ++ "world!"
"hello, world!"
```

Okay, enough of the REPL jazz session.

Now back to the CSV processing code!

```
  putStrLn $ "Total atBats was: " ++ (show summed)
```

Last, we stringify the summed up count using `show` , then concatenate that with a string to describe what we're printing, then print the whole shebang using `putStrLn` . The `$` is just so everything to the right of the `$` gets evaluated before whatever is to the left. To see why I did that remove the `$` and build the code. Alternatively, I could've used parentheses in the usual fashion. That would look like the following.

```
  putStrLn ("Total atBats was: " ++ (show summed))
```

`show` is a function from the typeclass `Show` . Here's how you can find out about it in your REPL:

```
Prelude> :type show
show :: Show a => a -> String

Prelude> :info Show
class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
-- Defined in ‘GHC.Show’
instance (Show a, Show b) => Show (Either a b)
-- Defined in ‘Data.Either’
instance Show a => Show [a] -- Defined in ‘GHC.Show’
instance Show Ordering -- Defined in ‘GHC.Show’
instance Show a => Show (Maybe a) -- Defined in ‘GHC.Show’
instance Show Integer -- Defined in ‘GHC.Show’
instance Show Int -- Defined in ‘GHC.Show’
instance Show Char -- Defined in ‘GHC.Show’
instance Show Bool -- Defined in ‘GHC.Show’
...
```

What `instance Show Integer` is telling us is that `Integer` has implemented `Show` . This means we should be able to use `show` on something with that type. We can specialize the type of `show` to `Integer` in a few passes.

```
show :: Show a => a -> String
show :: Show Integer => Integer -> String
-- you can just drop Show Integer =>, the typeclass
-- instances associated with a specific type are
-- a given.
show :: Integer -> String
```

In fact, we can even make a pointless version of show pre-specialized to `Integer` . Here's an example from my REPL:

```
Prelude> :t show
show :: Show a => a -> String
Prelude> :t show myInteger
show myInteger :: String
Prelude> let integerShow = show :: Integer -> String
Prelude> integerShow 1
"1"
Prelude> integerShow ("blah", ())

<interactive>:11:13:
    Couldn't match expected type ‘Integer’
                with actual type ‘([Char], ())’
    In the first argument of ‘integerShow’, namely ‘("blah", ())’
    In the expression: integerShow ("blah", ())
    In an equation for ‘it’: it = integerShow ("blah", ())
Prelude> show ("blah", ())
"(\"blah\",())"
```

Next we'll look at `summer`. `summer` is the function we are folding our `Vector` with. You can hang `where` clauses off of functions which are a bit like `let` but they come last. `where` clauses are more common in Haskell than `let` clauses, but there's nothing wrong with using both.

Our folding function here takes two arguments: the tuple record (we'll have many of those in the vector of records), and the sum of our data so far.

Here `n` is the sum we're carrying along as fold the `Vector` of `BaseballStats`.

```
  where summer (name, year, team, atBats) n = n + atBats
```

Building and running our csv parsing program

First we're going to rebuild the project.

```
$ cabal build
```

Then, assuming we have the `batting.csv` I mentioned earlier in our current directory, we can run our program and get the results.

```
$ ./dist/build/bassbull/bassbull
Total atBats was: Right 4858210
$
```

Refactoring our code a bit

Splitting out logic into independent functions is a common method for making Haskell code more composable and easy to read.

To that end, we'll clean up our example a bit.

First, we don't care about `name` , `year` , and `team` for our folding code.

So we're going to use the Haskell idiom of bindings things we don't care about to `_` .

This changes our fold from this:

```haskell
where summer (name, year, team, atBats) sum = sum + atBats
```

To this:

```haskell
where summer (_, _, _, atBats) sum = sum + atBats
```

Next we'll make our extraction of the 'at bats' from the tuple more compositional. If you'd like to play with this further, consider rewriting our example program at the end of this article into using a Haskell record instead of a tuple. I used a tuple here because Cassava already understands how to parse them, sparing me having to write that code.

First we'll add `fourth` :

```haskell
fourth :: (a, b, c, d) -> d
fourth (_, _, _, d) = d
```

Then we'll rewrite our folding function again from:

```haskell
where summer (_, _, _, atBats) n = n + atBats
```

Into:

```haskell
where summer r n = n + fourth r
```

Here we can use something called *eta reduction* to remove the explicit record and sum values to make it point-free. Since our function is really just about composing the extraction of the fourth value from the tuple and summing that value with the summed up `atBat` values so far, this makes the code quite concise.

You can read more about this in the article on pointfree programming in Haskell (https://www.haskell.org/haskellwiki/Pointfree).

To that end, we go from:

```haskell
where summer r n = n + fourth r
```

to:

```haskell
where summer = (+) . fourth
```

`.` is how we compose functions in Haskell. The entire definition of `.` is:

```haskell
(f . g) x = f (g x)
```

So, for example, if we `multiplyByTwo . addOne` we're adding one, then passing that result to the `multiplyByTwo` function. In the csv parser code, first `fourth` gets applied to the `r` argument, then

`(+)` is composed so that it is applied to the result of `fourth r` and the value `n`.

We should also split out our decoding of `BaseballStats` from CSV data.

We're going to move this code:

```
let v = decode NoHeader csvData :: Either String (V.Vector BaseballStats)
```

Into an independent function:

```
baseballStats :: BL.ByteString -> Either String (V.Vector BaseballStats)
baseballStats = decode NoHeader
```

Then `summed` becomes:

```
let summed = fmap (V.foldr summer 0) (baseballStats csvData)
```

With that bit of tidying done, we should have:

```
module Main where

import qualified Data.ByteString.Lazy as BL
import qualified Data.Vector as V
-- cassava
import Data.Csv

type BaseballStats = (BL.ByteString, Int, BL.ByteString, Int)

fourth :: (a, b, c, d) -> d
fourth (_, _, _, d) = d

baseballStats :: BL.ByteString -> Either String (V.Vector BaseballStats)
baseballStats = decode NoHeader

main :: IO ()
main = do
  csvData <- BL.readFile "batting.csv"
  let summed = fmap (V.foldr summer 0) (baseballStats csvData)
  putStrLn $ "Total atBats was: " ++ (show summed)
  where summer = (+) . fourth
```

Now we're going to double-check that our code is working:

```
$ cabal build
...(stuff happens)...
$ ./dist/build/bassbull/bassbull
Total atBats was: Right 4858210
```

Streaming

We can improve upon what we have here. Currently we're going to use as much memory as it takes to store the entirety of the csv file in memory, but we don't really have to do that to sum up the records!

Since we're just adding the current records' "at bats" with the sum we've accumulated so far, we only really need to read one record into memory at a time. By default Cassava will load the csv into a `Vector` for convenience, but fortunately it has a streaming module so we can stream the data incrementally and fold our result without loading the entire dataset at once.

First, we're going to drop Cassava's default module for the streaming module.

Changing from this:

```haskell
-- cassava
import Data.Csv
```

To this:

```haskell
-- cassava
import Data.Csv.Streaming
```

Next, since we won't have a `Vector` anymore (we're streaming, not using in-memory collections), we can drop:

```haskell
import qualified Data.Vector as V
```

In favor using the `Foldable` typeclass Cassava offers for use with its streaming API:

```haskell
import qualified Data.Foldable as F
```

Then in order to use the streaming API we just change the definition of our `summed` from:

```haskell
let summed = fmap (V.foldr summer 0) (baseballStats csvData)
```

To:

```haskell
let summed = F.foldr summer 0 (baseballStats csvData)
```

We are incrementally processing the results, not loading the entire dataset into a Vector.

The final result should look like:

```haskell
module Main where

import qualified Data.ByteString.Lazy as BL
import qualified Data.Foldable as F
-- cassava
import Data.Csv.Streaming

type BaseballStats = (BL.ByteString, Int, BL.ByteString, Int)

fourth :: (a, b, c, d) -> d
fourth (_, _, _, d) = d

baseballStats :: BL.ByteString -> Records BaseballStats
baseballStats = decode NoHeader

main :: IO ()
main = do
  csvData <- BL.readFile "batting.csv"
  let summed = F.foldr summer 0 (baseballStats csvData)
  putStrLn $ "Total atBats was: " ++ (show summed)
  where summer = (+) . fourth
```

The core here is the `Records` datatype `Cassava` gives us via the Streaming module. You can read more about the `Records` datatype on hackage (http://hackage.haskell.org/package/cassava-0.4.2.0/docs/Data-Csv-Streaming.html#t:Records). `Records` is a sum type, you could read out in English like so:

- `data Records a` -> Records is a datatype that takes a type variable `a`
- `Cons (...) | Nil (...)` -> It is a sum type of two possible constructors, `Cons` or `Nil` (note the list-like nomenclature). This is way of saying a `Record a` is always either `Cons` or `Nil`.
- `Cons (Either String a) (Record a)` -> the `Cons` data constructor is a product of `Either String a` and `Record a`. We're saying `Cons` is always `Either String a` *and* `Record a`. Also, this `Cons` resembles the cons-cells in Lisp, Haskell, ML, etc. The library has the following comment about it: "A record or an error message, followed by more records."
- `Nil (Maybe String) BL.ByteString` -> the `Nil` data constructor is a product of `Maybe String` and `BL.ByteString`. The library has the following comment: "End of stream, potentially due to a parse error. If a parse error occured, the first field contains the error message. The second field contains any unconsumed input."

What the Records type is doing for us is letting us process the records like a lazy list, but with a little extra context in the `Nil` case.

Because Haskell has abstractions like the `Foldable` typeclass, we can talk about folding a dataset without caring about the underlying implementation! We could've used the `foldr` from `Foldable` on our `Vector`, a `List`, a `Tree`, a `Map` - not just Cassava's streaming API. `foldr` from `Foldable` has the type: `Foldable t => (a -> b -> b) -> b -> t a -> b`. Note the similarity with the `foldr` for the list type, `(a -> b -> b) -> b -> [a] -> b`. What we've done is abstracted the specific type out and made it into a generic interface.

In case you're wondering what the `Foldable` instance is doing under the hood:

```
-- | Skips records that failed to convert.
instance Foldable Records where
    foldr = foldrRecords

foldrRecords :: (a -> b -> b) -> b -> Records a -> b
foldrRecords f = go
  where
    go z (Cons (Right x) rs) = f x (go z rs)
    go z _ = z
{-# INLINE foldrRecords #-}
```

Adding tests

Now we're going to add tests to our package. First we are going to add a test suite to our `bassbull.cabal` file. The name of our test suite will just be `tests`.

```
test-suite tests
  ghc-options:         -Wall
  type:                exitcode-stdio-1.0
  main-is:             Tests.hs
  hs-source-dirs:      tests
  build-depends:       base,
                       bassbull,
                       hspec >= 2.0 && < 2.1
  default-language:    Haskell2010
```

We're also going to add a library and shift over some code so that our package is exposed as a proper library rather than only working as an executable. We're exposing a single module named `Bassbull`. With an `hs-source-dirs` of `src` and an exposed module named `Bassbull`, Cabal will expect a file to exist at `src/Bassbull.hs`.

```
library
  ghc-options:         -Wall
  exposed-modules:     Bassbull
  build-depends:       base >= 4.7 && <5,
                       bytestring,
                       vector,
                       cassava
  hs-source-dirs:      src
  default-language:    Haskell2010
```

We need to change our executable in the Cabal file so that it depends on our library. No point duplicating the code!

```
executable bassbull
  main-is:             Main.hs
  ghc-options:         -rtsopts -O2
  build-depends:       base,
                       bassbull,
                       bytestring,
                       cassava
  hs-source-dirs:      src
  default-language:    Haskell2010
```

Next we're going to create a file named `src/Bassbull.hs` and shift code from `src/Main.hs` over to it. Note we've also refactored our `main` function so it takes an argument of what csv file to process.

```haskell
-- src/Bassbull.hs

module Bassbull where

import qualified Data.ByteString.Lazy as BL
import qualified Data.Foldable as F
import Data.Csv.Streaming

type BaseballStats = (BL.ByteString, Int, BL.ByteString, Int)

baseballStats :: BL.ByteString -> Records BaseballStats
baseballStats = decode NoHeader

fourth :: (a, b, c, d) -> d
fourth (_, _, _, d) = d

summer :: (a, b, c, Int) -> Int -> Int
summer = (+) . fourth

-- FilePath is just an alias for String
getAtBatsSum :: FilePath -> IO Int
getAtBatsSum battingCsv = do
  csvData <- BL.readFile battingCsv
  return $ F.foldr summer 0 (baseballStats csvData)
```

And here's our defrocked `src/Main.hs` which is now only responsible for fronting the executable.

```haskell
module Main where

import Bassbull

main :: IO ()
main = do
  summed <- getAtBatsSum "batting.csv"
  putStrLn $ "Total atBats was: " ++ (show summed)
```

Next we'll create a directory named `tests` and add a file named `Tests.hs` to it.

For our tests, we're going to use HSpec (http://hspec.github.io/) because the library is easy to use, the syntax is clean, and the author Simon Hengel (https://github.com/sol) is one of the most responsive and helpful I've run into in open source.

Here's our `tests/Tests.hs` file

```
module Main where

import Bassbull
import Test.Hspec

main :: IO ()
main = hspec $ do
  describe "Verify that bassbull outputs the correct data" $ do
    it "equals zero" $ do
      theSum <- getAtBatsSum "batting.csv"
      theSum `shouldBe` 4858210
```

There's not too much here. We're importing `Bassbull`, which is the library module we've exposed. This is also a `Main` module with its own `main` file because we execute our test suite as a binary just like we do with executables.

With all that in place, we'll install the dependencies our tests need.

```
$ cabal install --enable-tests
```

Then to run the actual tests, we'll run `cabal test`.

```
$ cabal test
```

Incidentally, `cabal test` is just a shortcut for building `tests` specifically, then running the executable produced to see test output.

You aren't limited to building the `tests` binary and running your tests in that manner. You can also pass `cabal repl` an argument to make it load your tests. This can be faster as the REPL uses an interpreter and can reload your code very quickly - much more quickly than doing a full build & execution run.

```
$ cabal repl tests
```

The above will then give you a REPL which can see anything the build in your Cabal named `tests` can see. You can then run the `main` function or individual test suites - if you bother to split them out.

Tests are useful and important in Haskell, although I often find I need *much* fewer of them. Often my process for working on an existing Haskell project will involve working on the code I'm changing with Emacs and a REPL instantiated via `cabal repl`. As my code starts passing the type-checker, I start running the tests as another layer of assurance that I'm doing the right thing.

I like having a lot of feedback and help from my computer when writing code!

Making your Haskell packages available to

the Haskell community

Hackage (https://hackage.haskell.org/) is the main community repository of Haskell packages and will usually be where you look to find libraries you need.

Mostly you'll find libraries and the occasional executable utility, but utilities should *also* be exposing library APIs that make their functionality accessible via Haskell code. This is not only more useful to other people but enforces good practices and more modular projects.

Haskell users are accustomed to documentation that is accessible via the Hackage website directly such as you might find for the base library that comes with GHC (http://hackage.haskell.org/package/base-4.7.0.1/docs/Data-Functor.html). The tool that builds this documentation is called Haddock (https://www.haskell.org/haddock/).

I strongly recommend you look at well-established libraries like lens (github.com/ekmett/lens) for examples of how to build your documentation (https://github.com/ekmett/lens/blob/master/scripts/hackage-docs.sh) and use continuous integration (https://github.com/ekmett/lens/blob/master/.travis.yml) with your Haskell projects.

To learn more and for more information on building a package for uploading to Hackage see this tutorial (https://www.haskell.org/haskellwiki/How_to_write_a_Haskell_program).

How I work

When I'm working with Haskell code, I interact with my code in a few ways. One is that I'm writing the code itself in Emacs. I'll also have a terminal with a REPL open, usually via `cabal repl` as I am almost always working on a specific project.

My Emacs config is pretty sundry, it's just `haskell-mode` and `ghc-mod`. `ghc-mod` is where the real magic comes up. This is how I get things like "tell me the type of that sub-expression" via a keyboard shortcut without having to type the code into my REPL. One nice thing about ghc-mod is that it is just a little service that runs independently of your editor, so it works with vim and SublimeText as well.

For a quick demonstration of how I work in Haskell (I don't show off everything in ghc-mod), take a look at this video I made (https://www.youtube.com/watch?v=Li6oaO8x2VY).

My basic happy-path event-loop for writing Haskell is:

1. Import module I'm working on in the REPL before I've changed anything
2. Change/add/delete code
3. `:reload` in the REPL. `ghc-mod` will give me type errors, but I sometimes like to see them in the REPL too.
4. Sometimes I'll use eta-reduction to refactor code. You can see an example of this in this code review on StackExchange (http://codereview.stackexchange.com/questions/57843/update-map-in-haskell/57850#57850). Making code point-free makes the most sense when it's primarily about *composing* functions rather than about applying them.
5. If code still type-checks after some cleaning, I'll run the tests. If tests pass, I move on unless I'm suspicious about test coverage. If tests break or I want more coverage, I write more tests until I'm satisfied. When that's done, I return to step #1 in this loop for the next unit of work I want to perform.

My diagnosis process when something *isn't* working:

1. If I can't get something to type-check, I'll break down sub-expression, query the types of those sub-expressions and make certain they were what I expected.
2. If have expressions I am trying to combine and I trying to make the types thereof make sense, but I haven't implemented them yet I will use `undefined` and work with only application, composition, and monadic variations thereof to figure out how I need to get to where I'm going before I've implemented anything. You can see a good example of this in this Github gist (https://gist.github.com/ifesdjeen/4be994aea5846aa1c2fe). I wrote the solution @ifesdjeen displays in his final comment.
3. If I have a function expecting arguments I can't figure out how to satisfy, I will sometimes use typed holes (https://www.haskell.org/haskellwiki/GHC/Typed_holes) or a similar trick with implicit parameters to see what type I need to provide.
4. Since Haskell functions are pure and lazy, I can replace references to functions with their contents with a high degree of confidence that it will not change the semantics of my program. To that end, sometimes it's easier to understand what's going on by inlining the code by hand and seeing what your code turns into.
5. If something type-checks but doesn't work, I'll run the tests. If the coverage isn't catching it, I add it. This is less common for me in Haskell than you'd think. If I can frame the test as an assertion about some *property* the code should satisfy like with QuickCheck (http://hackage.haskell.org/package/QuickCheck) I will do so. You can learn more about using QuickCheck in Real World Haskell (http://book.realworldhaskell.org/read/testing-and-quality-assurance.html).

One common mistake people make is not having `ghc-mod` and `ghc-modi` in a path that Emacs knows about. I recommend doing the following to install ghc-mod itself:

```
# We update so cabal-install knows about the latest versions of ghc-mod.
$ cabal update
$ cabal unpack ghc-mod
# At time of writing, this was the current version. Change as necessary.
$ cd ./ghc-mod-5.2.1.1
$ cabal sandbox init
$ cabal install
$ sudo ln -s `pwd`/.cabal-sandbox/bin/ghc-mod /usr/bin/ghc-mod
$ sudo ln -s `pwd`/.cabal-sandbox/bin/ghc-modi /usr/bin/ghc-modi
```

I found this to be easier than fussing around with adding the sandbox binary directories to Emacs' search path.

ghc-mod

- Hackage page (https://hackage.haskell.org/package/ghc-mod)
- Github repository (https://github.com/kazu-yamamoto/ghc-mod)

Emacs

- Haskell wiki section on Emacs and ghc-mod (https://www.haskell.org/haskellwiki

/Emacs#ghc-mod)

- Installing ghc.el (ghc-mod integration) for Emacs (http://www.mew.org/~kazu/proj/ghc-mod/en/preparation.html)
- Using ghc-mod in Emacs (http://www.mew.org/~kazu/proj/ghc-mod/en/emacs.html)

vim

- vim plugin Github repository (https://github.com/eagletmt/ghcmod-vim)

Sublime Text 2/3

- The SublimeHaskell plugin uses ghc-mod (https://github.com/SublimeHaskell/SublimeHaskell)

My personal dotfiles

- My dotfiles on Github (https://github.com/bitemyapp/dotfiles)

Wrapping up

This is the end of our little journey in playing around with Haskell to process CSV data. Learning how to use abstractions like `Foldable` , `Functor` or use techniques like *eta reduction* takes practice! I have a guide (https://github.com/bitemyapp/learnhaskell) for learning Haskell which has been compiled based on my experiences learning and teaching Haskell with many people over the last year or so.

If you are curious and want to learn more, I strongly recommend you do a course of basic exercises first and then explore the way Haskell enables you think about your programs in terms of abstractions. Once you have the basics down, this can be done in a variety of ways. Some people like to attack practical problems, some like to follow along with white papers, some like to hammer out abstractions from scratch in focused exercises & examples.

Things to do after finishing this article:

- Check out the Haskell community website (https://haskell.org)
- Learn about (unit|spec|property) testing Haskell software with Kazu Yamamoto's tutorial (https://github.com/kazu-yamamoto/unit-test-example/blob/master/markdown/en/tutorial.md)
- Search for code by *type* structurally with Hoogle (http://haskell.org/hoogle)
- Learn about Haddock, the Haskell source documentation tool (https://www.haskell.org/haddock/) and look at the many examples (http://hackage.haskell.org/package/base-4.7.0.1/docs/Data-Functor.html) of Haskell package documentation (http://hackage.haskell.org/package/pipes-4.1.3/docs/Pipes-Tutorial.html).

More than anything else, my greatest wish would be that you develop a richer and more rewarding relationship with learning. Haskell has been a big part of this in my life.

Special thanks to Daniel Compton (https://twitter.com/danielwithmusic) and Julie Moronuki (https://twitter.com/argumatronic) for helping me test & edit this article. I couldn't have gotten it together without their help.

Chris Allen

Haskell

Coder, Teacher, Author

Chris (http://bitemyapp.com) is a long time FP and Lisp user who discovered a love of learning and types when he found Haskell (http://www.haskell.org). Aside from releasing multiple Haskell project, such as Bloodhound (https://github.com/bitemyapp/bloodhound) and Blacktip (https://github.com/bitemyapp/blacktip) he took to teaching Haskell to spread the love and creating the Learn Haskell (https://github.com/bitemyapp/learnhaskell) guide which he is turning into a book (http://haskellbook.com/).