Melbourne Haskell Workshop 2013

DRAFT COPY

- 6 Hours (Plus Lunch)
- Expected attendance - 50 people

(HTML)

A 6-hour workshop intended to introduce and provide resources for working with Haskell.

This workshop is based around a central concept of domain modeling with Haskell.

Outcomes include...

- Creating, editing, running and interacting with Haskell programs
- Building the confidence to solve problems in the wild with Haskell
- Developing an understanding of the Haskell ecosystem
- Interacting with others in a collaborative environment

Table of Contents

Resources	Resources Available and Required	1m
Welcome	Motivation, Overview, and Approac	15m
Setup	Setting up your Haskell environme	15m
Introduction	Introductory Exercises	30m
Types	The Haskell Type System	30m
ADTs	Modelling with data in Haskell	1h
~ Lunch Break ~	Pizza -> Nomnomnomnomnom	1h
Laziness	Evaluation when required	1h
Type-Classes	Polymorphism, FP style	30m
Wildcard	Music	30m
Monads	IO Monad, Do-Notation	1h
Ecosystem	Resources and Community	30m
~ MOAR!! ~	Awesome Extras!	?
Testing	Testing with QuickCheck	1h
Algebra	Symbolic Mathematics	30m
Web-Site	Making a Web-Site with Scotty	30m
Compression	Compression	1h
Appendix	Appendix	~

Required Resources

Before you begin you will require the following...

A Text-Editor

We are assuming previous programming experience, however, if you would like a recommendation, have a look at Notepad++, Sublime Text, or the ever-popular Emacs and Vim. Just make sure that you are fluent enough to embark on exercises as they appear in the workshop.

The Haskell Platform

In order to run the programs written during this workshop you will need a Haskell installation. The easiest way to get up and running is to install the latest Haskell Platform. This is a "batteries included" installation of GHC and Cabal that includes many of the most useful packages available in the Hackage ecosystem.

A Copy of the Workshop Scaffold Project

https://github.com/sordina/haskell_workshop

Either clone with git:

```
git clone https://github.com/sordina/haskell_workshop.git
```

... or download the zip from GitHub.

Useful Resources

These resources are available to help you with any issues you face when learning Haskell:

#haskell on Freenode

An IRC channel dedicated to discussion of Haskell. This is often the easiest place to fire off a one-off question that is simple enough not to warrant a permanent listing on the internet.

Hackage

Hackage is the primary repository for Haskell packages. It is public, searchable, versioned, and uses Cabal package metadata for classification. Furthermore, this can be used to easily browse package contents, documentation and source-code.

For example, browse the Shake package and look at some of the Modules.

Hoogle

Hoogle is a Haskell module and function search-engine. Hoogle allows you to take advantage of the granular type-system used by Haskell to search not just for function-names, but for function type-signatures.

For example, have a look for the function with signature Text -> ByteString.

MFUG

MFUG is the Melbourne Functional Programmer's User Group. This group discusses many topics, including Haskell.

/r/haskell

For Reddit users, /r/haskell is a very useful resource with a great deal of information regarding recent developments in the Haskell ecosystem and community. This is a good place to ask some more advanced questions or start a flame-war.

Haskell News

Haskell News is a firehose-style haskell news aggregator taking information from sources as varied as academic-journals, and GitHub accounts.

HLint

HLint is a linting tool for Haskell source - It can often provide some useful hints about refactoring avenues for your code.

Learn You a Haskell (For Great Good)

Learn You a Haskell (For Great Good) is a wonderful introductory text on Haskell.

Real World Haskell

Real World Haskell aims to focus on solving real problems with Haskell. Several chapters of the book are somewhat dated in terms of the libraries used, but it still serves as a useful example of getting real work done with Haskell.

Welcome

Welcome to the Melbourne Haskell Workshop 2013.

This intent of this workshop is to provide a working introduction to Haskell for programmers in Melbourne who have not yet used the language in anger.

The workshop is split into chapters. The chapters will start with a few trivial introductory exercises to get your fingers warmed up, then the meat - exercises that should be able to be solved using the material introduced up to that point (with help and hints available if needed).

Each chapter will conclude with an open-question. This question should provide inspiration for further consideration of how harder problems could be solved using Haskell, and for more advanced attendees, can be attacked instead of twiddling your thumbs after finishing the main exercise.

The workshop will be split into pre, and post lunch parts, with additional exercises available for busy beavers following the MOAR section of the table of contents.

Why run a free haskell workshop?

There are a few kinds of benefits that we anticipate will result as an outcome of the workshop:

- Benefits to Individuals
- Benefits to the Community
- Benefits to the Volunteers
- Benefits to the Organisers
- Benefits to Industry
- Benefits to Relationships

Benefits to Individuals

- New Tools
- Experience writing Haskell
- An Introduction to the Wider Community

What do we hope that an individual attendee will gain by coming to the workshop?

Since this workshop is targeted towards beginners, we want an attendee to come away from the workshop with new tools at their disposal. Yes, a new language to reimplement existing knowledge with various platform benefits, but more importantly, a new tool for dissecting problems.

Learning a new problem-solving approach will almost certainly be the most challenging part of the workshop, but through the consistent delivery of exercises that benefit from domain modelling, we hope that the attendee will walk away with the confidence to attack new problems from the “Haskell Mindset”.

Although there won't be time to work on any large-scale problems, we hope that the approaches we demonstrate will be able to be applied with confidence to problems of any size.

Aside from the technical benefits, the workshop should provide a great opportunity for attendees to meet many other members of the Melbourne Haskell using community.

Benefits to the Community

- Haskell as a Tool for Communication
- Local Network Effects
- Consolidation

Often, Haskell (especially the type-system) can be an extremely effective means of communication of certain kinds of behaviors, requirements, and problems. If this workshop can help advance Haskell as a common language for inter-personal discussion of technical topics, then the barriers of communication will be lowered and the community as a whole should become more productive.

Often, it isn't until a particular community reaches a critical mass that it comes into its own. Currently the Melbourne Haskell community is reasonably disparate and unconnected (especially outside of MFUG). If the various groups of people using Haskell can be consolidated, then we would consider starting a regular Melbourne Haskell meetup. Just having the ability to ask questions in person can go a long way to advancing confidence in a particular topic, therefore, as the number of Haskell users increases locally, so to will the overall skill of individuals and the community as a whole.

Benefits to the Volunteers

- Experience
- Networking

We hope that for the volunteers, providing administrative and in-person assistance will be a rewarding experience. We are very open to people of any skill level helping out and this could be a great opportunity for newcomers to the community to get their fingers dirty by providing assistance at the workshop.

The workshop should also be a good networking opportunity for volunteers, as a stratified group of Haskellers of this size has not yet convened in Melbourne.

Benefits to Industry

It has been said that once you are comfortable programming Haskell, you become spoiled, and won't want to go back to your previous languages. Rather than quit your day-job, if Haskell's presence can be advanced in industry, then the Joy of programming haskell will become a possibility in your day-to-day work-life. A longer-term goal to be sure.

This benefit may be the most tenuous outcome of the workshop, but as the mindshare of Haskell grows, the chances of it being used for projects within industry increases.

- An understanding of the language's strengths and weaknesses emerges empowers decision makers to weigh up if the language is a suitable candidate to solve the technical problems at hand.
- Confidence is gained that there are enough people using the language that hiring a programmer will not be a problem if this is required.

The technical benefits to industry will have the most impact where a rigorous solution to a problem is necessary. In the financial sector, Haskell is beginning to become more popular as the cost of bugs can be very high. Technical benefits, however, are a moot point if Haskell does not see adoption.

If this workshop can give project-leaders in industry, making programming-language choices, the confidence to consider Haskell as a real candidate, then it will have gone a long way towards making Melbourne a more attractive place to be for people who wish to use Haskell in day-to-day work.

Benefits to the Organisers

Last-but-not-least -

For us, the organisers, running this workshop will be a great opportunity to see the Melbourne Haskell community in action. We're looking forward to working with you all and expect that, above all else, we will have a lot of...

Fun!

Setup

Ensure that you have the following programs installed and functioning correctly:

GHCi

At a command prompt, enter the following command:

```
ghci
```

This should launch the GHC Haskell REPL.

Type the following to ensure that you have a functioning REPL:

```
1 + 1
```

This should yield the result:

```
2
```

```
You can use GHCi to perform calculations other than just "1 + 1".

Here is an example session:

> ghci
GHCi, version 7.6.2: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[Prelude] > 1 + 2 + 3
6
[Prelude] > 100 / 2
50.0
[Prelude] > 6 ^ 7
279936
[Prelude] > ^D
Leaving GHCi.

Using GHCi...

Calculate the price of 42-bakers-dozens of eggs at $3 per-egg.

[Prelude] 42 * 13 * 3
1638
```

Loading files in GHCi

There are many ways to load and execute Haskell code. For the purposes of this workshop, if you do not already have a workflow you are comfortable with, then we suggest the following steps:

- Write and edit your programs in files ending in the extension ".hs"
- When you are ready to test your program, load it into GHCi
- After making modifications to your program, reload the program in GHCi

Say you had written the following program `test.hs` in your home directory:

```
main = print "hello world"
```

Load the file in GHCi to see it in action:

```
> ghci test.hs
GHCi, version 7.6.2: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, modules loaded: Main.
[*Main] > main
"hello world"
```

... Unfortunately there is a bug in the program, so in your editor you make the change required to print "hello, world" with the mandated comma:

```
main = print "hello, world"
```

Now, back in GHCi, you can reload the program without exiting the REPL:

```
[*Main] > :reload
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, modules loaded: Main.
[*Main] > main
"hello, world"
```

Much better!

```
You can inspect a value (of function) in ghci with the ‘:info‘ command
in order to find out a little about its type and definition:

ghci> :info main
```

```
main :: IO ()   -- Defined at test.hs:1:1

If you just wish to see the type of an expresison, you can use
the ‘:type‘ command:

ghci> :type main
main :: IO ()



* Define a new numeric function in a source file
* Load it in GHCi
* Test your function in GHCi
* Make a modification
* Reload your chages without exiting GHCi
* Test your changes
```

GHC

Create the following source file (program.hs):

```
main = print "hello world"
```

Compile the program as follows:

```
ghc --make program.hs
```

Run the program with the following command:

```
./program
```

The output should look as follows:

```
"hello world"

Compiled programs are almost always significantly faster than instructions
run inside GHCi. Even greater speed-ups are possible by using the "-O"
optimisation settings for GHC.

Using GHC...

Compile and run hello-world.

> echo 'main = print "hello homies"' > main.hs
> ghc --make main.hs
[1 of 1] Compiling Main             ( main.hs, main.o )
Linking main ...
> ./main
"hello friends"
```

Cabal

You should have access to a Cabal installation if you have installed the Haskell Platform.

Check that you have cabal by running:

```
cabal --version
```

This should output something similar to:

```
cabal-install version <VERSION>
using version <VERSION> of the Cabal library
```

```
Install the QuickCheck package using cabal.
```

```
> cabal update # not required if done recently
Downloading the latest package list from hackage.haskell.org
> cabal install quickcheck
...
```

```
An open-ended question:
```

```
Given that GHC is largely written in Haskell, how was GHC first compiled?
```

```
An open-ended question:
```

```
What are some of the current issues with the Haskell ecosystem?
```

Introduction

The following exercises are intended to be used to warm up your fingers, rather than your brain. These should be run through quickly to get you used to using your development environment.

Variables

In Haskell you can define a variable with the `=` sign.

Variables can be defined at the top-level (no-indentation):

```
myVariable = 2
```

Variable names should start with a lowercase letter and contain no spaces, or special characters, besides underscores, numbers, and `'`.

Some examples of variable names are:

- `a`
- `my_name`
- `data43'`

```
Define your own variable.
```

```
x = "hello"
```

```
What is an example of an invalid variable name?
```

```
invalid-variable = 123
```

String literals look familiar:

```
myString = "hello world"
```

```
Define a variable containing a string.
```

Tuples

Tuples look like this:

```
myTuplePair = (1,"hello")
```

```
myTupleTrio = (1,"hello",3)
```

They can be used to group multiple, differently-typed (heterogeneous) values.

```
Define a variable containing a tuple.
```

Functions

Functions are a core part of Haskell. Function definitions look like this:

```
myFunction x y ... = ...
```

For example:

```
myAdd x y = x + y
```

`myAdd` takes two numbers and returns the result of the addition of those two numbers.

```
Define a function ‘myMultiply‘ that multiplies 3 numbers.
```

```
myMultiply x y z = x * y * z
```

Lists

List are a commonly used data-structure in Haskell. Everything in a list has the same type (they are homogeneous).

Lists are built using the infix data-constructor `(:)` (pronounced “cons”). They also have a compact notation using `[...]`.

List literals look like:

```
list1 = [1,2,3]
list2 = 1 : 2 : []
list3 = "hello" : "world" : []
```

More information about why lists can be used the way that they are is contained in the ADTs chapter.

```
Define a variable containing a list.
```

You can deconstruct a list by pattern matching the head and tail like so:

```
f (x:xs) = ...
```

```
Define a function to get the first element of a list.
```

Note: In `Prelude` this function is called `head`.

```
myHead (x:xs) = x -- This is a partial function, Beware!
```

```
Define a variable containing the first element of your list.
```

```
myFirstElement = myHead myList
```

Define Length

`Define a function that takes a list and returns the length.`

Your solution should have the form of:

```
myLength []     = ...
myLength (x:xs) = ...
```

Things to consider:

- What is the length of an empty list? (the base case)
- What is the length of xs?

```
mylength []     = 0
mylength (x:xs) = 1 + mylength xs
```

Define `myMap`

```
Define a function that takes a function from a to b "a -> b",
and a list of as "[a]", and returns a list of bs "[b]".
```

Things to consider:

- What is the base-case of myMap?
- What is the inductive-case of myMap?

```
myMap f [] = []
myMap f (x:xs) = f x : myMap f xs
```

Fun List Functions

For your reading pleasure, here are some definintions of common functions:

```
myFilter f []     = []
myFilter f (x:xs) = if f x then x : myFilter f xs
                           else     myFilter f xs

myFold f z []     = z
myFold f z (x:xs) = f x (myFold f z xs)

myReverse []     = []
```

```
myReverse (x:xs) = myReverse xs ++ [x]

myElem e []     = False
myElem e (x:xs) = if e == x then True
                            else myElem e xs
```

```
An open-ended question:

What is a good balance between safety and expressiveness in a
programming-language?
```

Types

```
Question: How do you create a great program?
Answer:   You type it!
```

In this chapter, we will go over the exercises from the introduction and add types to the examples.

In Haskell, type signatures can be provided inline, or above definitions.

For example:

```
x :: Int
x = 3
```

or

```
x = (3 :: Int)
```

It is far more common to place the type-signature above the definition, with inline types only used in situations where ambiguities need to be resolved.

```
You are defining a floating point variable:

myFloat = 1.1

Give your variable a type-signature.

myFloat :: Float
myFloat = 1.1
```

Type Synonyms

In Haskell, we can give type-expressions an alias (or synonym) by using the `type` keyword. This allows you to cut down the verbosity and chance of errors in your code when you have type expressions that would otherwise be repeated frequently.

An example of this is the `String` type-synonym, which is defined as follows:

```
type String = [Char]



Give your string variable from the previous chapter a type-signature.

myString :: String
myString = "Hello Haskell"
```

Tuples

Tuple type signatures look the same as the tuples themselves, with types in place of the data.

For example, if you had a tuple of a String and an Int, the type would look as follows:

```
myTuple :: (String, Int)
myTuple = ("The meaning of life", 42)

Give your previous tuple definition a type signature.
```

Functions

The type signatures of functions in Haskell are a little different from how they look in the more familiar C family languages, but the syntax is very elegant, and will allow a higher-level of reasoning than less consistant forms.

The syntax for a function type-signarure is of the form:

```
{functionName} :: {argument} -> {result}
```

The main idea is that functions in Haskell only ever take one argument. If you wish to define a function that takes more than one argument, then you should, infact, define a function that takes one argument, then returns another function.

Luckily the syntax for doing this in Haskell looks identical to defining a multi-argument function:

```
myMultiply x y z = x * y * z
```

However, the distinction becomes clear with the type-signature:

```
myMultiply :: Int -> (Int -> (Int -> Int))
myMultiply x y z = x * y * z
```

It is clear, that the function only takes one argument, then returns a function (that only takes one argument, and returns a function (that only takes one argument, that returns an Int.))

Fortunately, Haskell's function syntax is right-associative, allowing us to drop the parentheses:

```
myMultiply :: Int -> Int -> Int -> Int
myMultiply x y z = x * y * z
```

```
Define a function ‘myMultiply‘ that multiplies 4 numbers.
Give your function a type-signature
```

```
myMultiply :: Int -> Int -> Int -> Int -> Int
myMultiply w x y z = w * x * y * z
```

Lists

List type-signatures look like:

```
list1 :: [Int]
list2 :: [Int]
list3 :: [String]

list1 = [1,2,3]
list2 = 1 : 2 : []
list3 = "hello" : "world" : []
```

More information about why lists can be used the way that they are is contained in the ADTs chapter.

```
Define a variable containing a list.

Give your variable a type-signature.
```

You can deconstruct a list by pattern matching the head and tail like so:

```
f (x:xs) = ...

Define a function to get the first element of a list.
```

Note: In `Prelude` this function is called `head`.

```
myHead (x:xs) = x -- This is a partial function, Beware!

Define a variable containing the first element of your list.

myFirstElement = myHead myList
```

Define Length

```
Define a function that takes a list and returns the length.

-- In Haskell type-signature syntax, this is written as:
myLength :: [a] -> Int
myLength = undefined
```

Your solution should have the form of:

```
myLength []     = ...
myLength (x:xs) = ...
```

Things to consider:

- What is the length of an empty list? (the base case)
- What is the length of xs?

```
mylength []     = 0
mylength (x:xs) = 1 + mylength xs
```

Define `myMap`

```
Define a function that takes a function from a to b "a -> b",
and a list of as "[a]", and returns a list of bs "[b]".
```

Things to consider:

- What is the type-signature of myMap?
- What is the base-case of myMap?

```
myMap :: (a -> b) -> [a] -> [b]
myMap f [] = []
myMap f (x:xs) = f x : myMap f xs
```

Fun List Functions

For your reading pleasure, here are some definintions of common functions:

```
myFilter f []     = []
myFilter f (x:xs) = if f x then x : myFilter f xs
                           else     myFilter f xs

myFold f z []     = z
myFold f z (x:xs) = f x (myFold f z xs)

myReverse []     = []
myReverse (x:xs) = myReverse xs ++ [x]

myElem e []     = False
myElem e (x:xs) = if e == x then True
                            else myElem e xs
```

See if you can determine the type-signatures for these functions.

```
An open-ended question:

What is a good balance between safety and expressiveness in a
programming-language?
```

ADTs (Algebraic Data Types)

Haskell Wiki Link

Algebraic Data Types are THE bread and butter of Haskell programs.

- Functions evaluate data by pattern-matching against ADTs
- Domains are modeled using ADTs
- Laziness is linked to ADTs
- Types can be derived from ADT definitions

But how does that help me?

An example of an ADT in Haskell:

```
data MyBool = MyTrue | MyFalse | MyNotSure

should_I_eat_something_bigger_than_my_own_head :: MyBool
should_I_eat_something_bigger_than_my_own_head = MyFalse


With this functionality, you are able to introduce your own "Enum"
values.

The MyBool example is somewhat equivalent to the following C++ code:

enum MyBool { MyTrue, MyFalse, MyNotSure };

With the added bonus of not having out-of-bounds casting ruin your day.

If your problem space can be moddeled using various discrete values,
then this form of ADT will allow you to create a program that mirrors
your problem!
```

You can add parameters to the data constructors:

```
data MyNullString = Nada | MyString String

stringy :: MyNullString
stringy = MyString "Whatever, It's just a string"

blanky :: MyNullString
blanky = Nada
```

Constructors can take multiple parameters:

```
data SomeJunk = Rubbish String | TrashPile String Int Bool

discards :: SomeJunk
discards = TrashPile "Junk Yard" 42 True
```

Furthermore, ADTs can be recursive:

```
data MyNumberList = Nada | SomeNumbers Int MyNumberList

numbers :: MyNumberList
numbers =  SomeNumbers 1 (SomeNumbers 2 Nada)
```

Finally, ADTs can be parametrised by other types:

```
data MyContainer x = Jar x

contained :: MyContainer Int
contained = Jar 1

pun :: MyContainer (MyContainer String)
pun = Jar (Jar "Binks")
```

In general, the syntax of an ADT looks similar to the following:

```
ADT          := data <TypeName> <Variables> = <Constructors>
TypeName     := [A-Z] + [a-z'_]*
Parameters   := <ConcreteType> + (" " + <ConcreteType>)*
Constructors := <Constructor> + (" | " + <Constructor>)*
Constructor  := <TypeName> + <Parameters>
Variables    := <Variable> + (" " + <Variable>)*
Variable     := [a-z] + [a-z'_]*
```

ConcreteType can't be defined syntactically, but it means that your type is "Fully Applied" (in Haskell terms, has a kind of `*`). An example of some concrete types are:

- `String`
- `Int`
- `Maybe String`
- `[Int]`

Examples of some non-concrete types are:

- `Maybe`
- `IO`
- `(->)`

Deriving

One final thing to note is that in order to be able to print values of your data types at the GHCi REPL, you will need your data to be a member of the `Show` type-class.

Type-classes are covered in depth in the type-classes chapter.

This is achieved by appending the following text after your data definition:

```
data MyData = SuperGreat deriving (Show)
```

Similar classes are needed for ordering and equality. If you get stuck on a missing class, just add the following deriving triple for now:

```
data MyData = SuperGreat deriving (Eq, Ord, Show)
```

Exercises

With all of this power at your disposal, it's time to define a list ADT yourself.

```
Define your own generic list ADT.
```

Things to consider:

- Should this ADT be parametrised?
- Should this ADT be recursive?
- Should this ADT have multiple constructors?
- Should the constructors be parametrised?

```
data MyList a = Empty | Items a (MyList a)
```

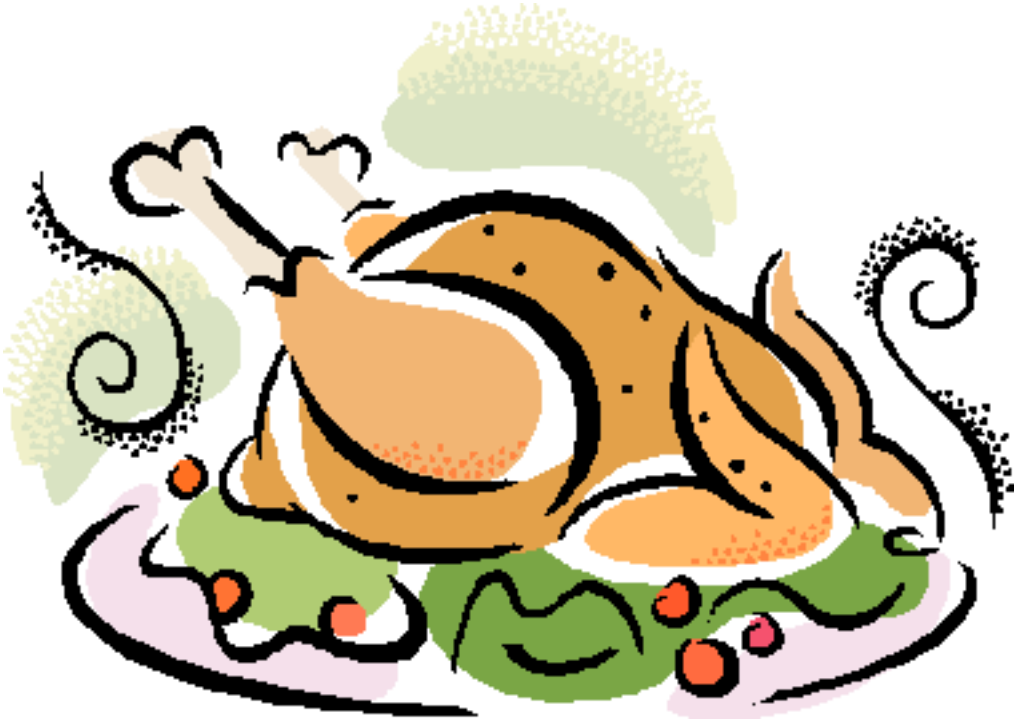
```
An open-ended question:

What would the ADT for a Lisp-like language look like?
```

If you wish to learn about why ADTs are "Algebraic", then have a look at:

- The Algebra of Algebraic Data Types, Part 1
- The Algebra of Algebraic Data Types, Part 2
- The Algebra of Algebraic Data Types, Part 3

Lunch Break

Lunch is provided courtesy of ThoughtWorks



Laziness

Haskell's evaluation strategies ~ Exploring the magic.

Often when looking at examples of Haskell functions you will come across examples of laziness in action. The canonical example used to demonstrate the lazy recursive approach of Haskell is the definition of the list of Fibonacci numbers -

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

This solution has a reputation for being somewhat mind-bending for beginners... What is going on here?

```
In order to show the start of an infinite list, use the ‘take‘ function -

For example:

[Prelude] > let fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
[Prelude] > take 10 fibs
[1,1,2,3,5,8,13,21,34,55]
```

Here is a Fibonacci solution in Javascript:

```
function fib(n) {
    if(n < 2) return 1
    return fib(n-1) + fib(n-2)
}
```

Most programmers won't have any problem with the usual recursive definition of `fibs`, so what makes the Haskell one special? Let's see if we can implement the Haskell version in Javascript:

```
// The Fibonacci Sequence in all its glory~
//
function fibs() { return [1, cons(1, zipWith(add, fibs, tail( fibs )))] }

exports.main = function() { runList(fibs, 13) }

// Helpers
//
function cons(h,t)        { return function() { return [h,t] }}
function head(list)       { return list()[0] }
function add(x,y)         { return x + y }
function tail(list)       { return function() { return list()[1]() }}

function zipWith(f, x, y) { return function() {
```

```
                            return [ f(head(x), head(y)),
                                     zipWith(f, tail(x), tail(y))] } }

function runList(list, n) { while(n --> 0) {
                            console.log(head(list)); list = tail(list)}}
```

The main difference between a normal recursive definition and this one is that the recursion happens at the value level, as opposed to the function-call location.

```
Implement an infinite list of ascending numbers using lazy,
value-based recursion.

ascending = 1 : zipWith (+) ascending ascending

An open-ended question:

Can all numeric-lists be defined by using value-based recursion?
```

Type Classes

A big part of writing clean reusable code is controlled polymorphism. We are like a carny at the roller-coaster, anybody can ride the roller coaster, but you must be at least this tall.

In the object oriented world we use the inheritance heirachy, we know that if something is a subclass, it has at least all of the features of its parent. Of course this is all intertwined with the sharing of data and state, and that's bad.

In the functional world we get type classes, which is just controlled polymorphism without the baggage. They basically say, that I don't need to know exactly what type you'll call me with but you need to provide a bunch of functions for me to use.

A function tells you what type classes it needs with a "context", The context is the bit to the left of the double arrow "=>"

```
(+) :: Num a => a -> a -> a

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

In the above example, we have `(+)`. It can work for any type that is a number. There are built in types for numbers and you can also define your own.

`show` can turn any "Showable" type into a string. this is analogous to the `toString()` method in Java.

```
Define a function that adds one to everything in a list.
What is the type of this function?
```

```
read :: Read a => String -> a

incrementAndshow :: (Num a, Show a) => a -> String
```

Unlike most other languages, with some kind of type driven function dispatch (e.g. methods, overloading). Haskell can also use the *return* type of the function to choose functionality, this can take a little getting used to, but it is powerful.

`read` turns a string into a value of another type, although we don't know what this is yet. It depends, for example on the type of the function that you are passing the result of read into.

`incrementAndShow` demonstrates a function that uses two type classes in its context.

```
In ghci, convert a string to an integer using read, then covert
a string into a list of integers using read.

(Hint: use (::) to add a type to an expression)

If you just type 'read "1"' in ghci you get an
error, why is this?

Define `incrementAndShow` which adds one to a number and
coverts it to a string.
What is they type of this function?
How did haskell infer your context?
```

Defining Your Own Type Classes

Let's define a type class of things that can be rated, as in 1 to 5 stars or awesome to lame.

```
data Rating =
  SoAwesomeICried |
  PrettyCool      |
  Meh             |
  ForTheLoveOfGodMakeItEnd
  deriving Show

class Rateable r where
  rating :: r -> Rating


data Beer = Coopers | Fosters | CarltonDraught
  deriving Show

instance Rateable Beer where
```

```
  rating Coopers = PrettyCool
  rating Fosters = ForTheLoveOfGodMakeItEnd
  rating CarltonDraught = Meh


data Movie = Movie String
  deriving Show

instance Rateable Movie where
  rating (Movie "Bladerunner") = SoAwesomeICried
  rating (Movie "Tron") = PrettyCool
  rating _ = Meh
```

When we define a type class we write out the name and the types of the functions that we want associated with this type class. We also put a type variable in our definition which refers to whatever type may instance this type class later. We can use this type variable in the definitions below. At this point we can also define default definitions for the functions.

We then define two new data types, `Beer` and `Movie` for which we add an "instance declaration" this is where we write the definitions of the type class functions specialized for beers of movies.

We now know how to manually add type class definitions and we could add custom `Show` instances for each of our new datatypes. However this would be a bunch of boilerplate nonsense, so we can use the handy deriving directive to automatically add a `Show` instance for us.

```
Create your own Rateable type class and wacky Rating datatype.
Then create datatypes and instances for something you have a
strong opinion about, like cars or a political party.


Add a review function to your type class that returns
a short textual review.


Add a Rateable instance for a native type, like String.
```

There are a few other cool things you can do with typeclasses that are not covered here. So if you want to know more, check out some of these other articles:

http://www.haskell.org/tutorial/classes.html

Music

When you are modeling a problem of a sufficient level of complexity you will find that your data-types often become recursive. One domain that sits roughly on this threshold is music.

Here is a small musical target-language:

```
data Music
  = A_ -- Underscore Represents a Flat
  | A
  | A' -- Prime represents a Sharp
       -- ... B, C, D, E, F, G ...
  | Sequence [Music]
  | Sharp Music
  | Longer Integer Music
  | Higher Integer Music
```

Certainly not comprehensive enough to write Beethoven's 5th Symphony, but good enough for Baa Baa Black-Sheep.

```
intro  = Sequence [ C, C, G, G ]
run    = Sequence [ A, B, C, A ]
middle = Sequence [ F, F, E, E ]
run2   = Sequence [ D, C, D, E ]
baaBaa = Sequence [ Longer 2  intro
                  , Higher 12 run
                  , Longer 4  G
                  , Longer 2  middle
                  , run2
                  , Longer 4  C ]
```

In order to listen to this music, create a file with the following code:

```
import Midi
import qualified Data.ByteString as BS

main = BS.writeFile "baaBaa.mid" (make_music 4 4 (Longer 4 baaBaa))
```

```
Write a melody using this language. Try to write and compose
functions, rather than declaring one big data-structure.

Why not try writing Beethoven's 5th Symphony after all?

An open question:

What kinds of music is this language incapable of modeling?
How would you modify this language to be more inclusive?
```

Monads

No seriously, what are Monads?

A monad is just a bunch of rules. There are many, many analogies for monads out there. Each of these analogies are useful, but can be obscuring on their own, they are just one view point. To effectively wield monads we must use many different view points together, each one revealing a little piece of the underlying structure of monads. Each view point becomes another tool in our mental toolbox.

So there is no one magic-bullet analogy for monads, only many complementary ones.

Haskell uses monads to represent side-effects. The simplist and most practical analogy is the "tainted value" analogy. In haskell the function that reads a file looks like this:

```
readFile :: FilePath -> IO String
```

This function can't return a normal string, because the return value has been tainted by side effects. So the IO monad is acting as a tag saying that the returned string must be treated specially.

But an `IO String` is not very useful to us, because we want to actually do things with it. So Haskell, in its normal paternalistic style, allows us access the `String` inside an `IO String` only in a very careful way. We use an operation call bind `(>>=)` to access the string.

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

-- Here specialized for IO and String

(>>=) :: IO String -> (String -> IO b) -> IO b
```

This says the only way we can "look inside" an `IO String` is by providing a function that takes a `String` and returns some other new value that has also been tainted by the outside world. In other words we can only look at a value "inside" the `IO` monad if we promise to make sure our result will also be tainted.

This means that if some code uses a value from the outside world, even deep inside, it cannot be hidden, it must be made explicit in the type. Tainting is a one way street, once you are tainted you can't be untainted. There is no function `untaint :: IO a -> a`. One can't get an untainted value from a tainted one.

In fact, in haskell, the very way we construct a useful program is by ultimately creating value of type `IO ()` that we assign to special variable called `main`.

```
Why can't one write untaint?
If you could what problems would this cause?
```

One thing that can be a little strange is the type of `getLine`. In imperative languages, functions of zero arguments make some sense. They can be thought of recipies or to-do lists. In haskell a total function of type `() -> Foo` is isomorphic to a constant `Foo`. Because the function only has one input value and therefore only one possible output value.

So let us return to `getLine`. In an imperative language it would look like `getLine :: () -> String`. Our first problem is that the return value of this function is tainted by the outside world, so we need to use the `IO` monad, `getLine :: () -> IO String`. Now because of the isomorphism between unit domian functions and constants we can just write `getLine :: IO String`. We call a constant of type `IO a` an "IO action". Because it stands for a bunch of side effects that will be performed together.

This will seem strange at first, because getLine isn't really a function – it's just a constant value! But that's okay because while the `IO String` is a constant (i.e. there is only one distinct IO action that reads a line from the console) the value *inside* the monad is not constant. It can be different each time we look inside.

```
> getLine
hello
"hello"

> getLine
monad
"monad"

> getLine >>= (\name -> putStrLn ("Hello " ++ name))
andy
Hello andy
```

One thing leads to another

Often when doing IO one wants to make sure one thing happens after another, We can use `(>>=)` and just ignore the unwrapped value:

```
putStr "Hello " >>= (\_ -> putStrLn "World")

putStrLn "One" >>= (\_ -> putStrLn "Two") >>= (\_ -> putStrLn "Three")
```

This pattern can be easily abstracted, it has been standardized as `(>>) :: IO a -> IO b -> IO b`. This can be read as "and then".

```
putStr "Hello " >> putStrLn "World"

putStrLn "One" >> putStrLn "Two" >> putStrLn "Three"

Write a program that prints something stupid, funny or rude.
Make sure to use (>>) somewhere.
```

Monad Wars III: Return of the Value

We mentioned before that there is no way to untaint a value, once it has been tainted, we can make new tainted values from it, but never untainted ones. But that begs the question, can we choose to taint a value? Indeed we can, in fact, this is a fundamental operation of a Monad. In haskell it is confusingly called `return :: a -> IO a`.

A common pattern is to "open up" an `IO` with bind `(>>=)`, mess around with the contents then wrap it back up again with `return`. We have a function to help us with this called `lift`. Specialized for `IO` it has type `lift :: (a -> b) -> (IO a -> IO b)`. We can use this to take a vanilla function and "lift" it into the IO monad. It works by unwrapping the IO monad calling our function and then wrapping the result back up again.

```
use return to write 'myLiftIO :: (a -> b) -> IO a -> IO b'
```

Do it, do it good.

Sometimes when you are doing a lot of ad-hoc interleaved IO, using bind and return all over the place can become a pain in the fingers. So haskell provides special syntax for using monads, called "do notation".

To use do notation, we start a "do block" with the keyword `do`. Inside a do block, statements can be written line by line and they are automatically joined using "and then" `(>>)`. This causes them to be run one after the other like in an imperative programming language. One can also easily unwrap monad values and bind them to a variable with using a left arrow "<-" syntax.

```
main = do
  putStrLn "Hello World"
  putStrLn "Enter your name: "
  name <- getLine
  putStrLn ("Hello: " ++ name)
```

The do-notation is like a DSL, under the hood it is just expanded to a chain of bind `(>>=)` calls. Here is what the above code would look like without do-notation:

```
main  =
  putStrLn "Hello World" >>
  putStrLn "Enter your name: " >>
  getLine >>= (\ name ->
    putStrLn ("Hello: " ++ name))
```

```
Write a program that asks for someone's first and second name,
then complements them (or makes fun of them).

For extra points ask for their age, and customize the complement
(insult) depending on how old they are.

Do it with do-notation first, then "desugar" it into raw bind (>>=) calls


My father once told me about an "amazing AI" that his
magician/programmer friend build, which could answer any yes/no
question as well as a human.

Of course it only worked if his friend was the one typing
the question! The trick being that it just counted the
number of spaces in the question. If there was an even number
it would output true, if there was an odd number, false.
You just fiddled with the wording, or snuck in an extra space,
to get the answer that you wanted...

Looks like it's time to write your super-dooper
human-level AI and see if your friends can figure
out how it works.
```

Stay Functional, San Diego

Even when we are programming with side effects, we still want our programs to follow functional principals. To stop our code ending up like C written in do-notation, there are some principals we can try to follow:

1. Try to do most of the work inside pure functions.
2. Don't create an IO action until the last possible moment.
3. Be declarative, think of program as declaring a pipeline or specifying an interaction, instead of being a to-do list.
4. Avoid mixing control, state and calculation. Instead abstract them into small composable pieces. For inspiration see the monad combinators in Control.Monad (e.g. `sequence`, `forever`, `mapM`).

These principals are not specific to monads. They are applicable to all side-effect heavy programing problems. These principles can also be applied to imperative programs for great justice.

A lot of patterns like reactive programming, dataflow programming and CQRS are consequences of following principals such as these.

The pithy take-away is don't "update, execute, update". Instead "represent, transform, compose".

```
An open-ended question:

Why is it a good idea to make side effects explicit?
```

Ecosystem

The Haskell ecosystem is large and interesting, it is held together more by convention than by dictation, with the current convention being that open source packages are made available through `cabal` on Hackage.

Cabal

Cabal is a library package, but "cabal" is often used to refer to the cabal executable provided by the "cabal-install" package. These packages are both provided by the Haskell Platform.

The premiere online Cabal package repository is Hackage.

Cabal provides mechanisms for download and installing packages from Hackage onto your system. It also provides a consistent way to structure, export, test, and package your own programs.

```
Install the hlint package from hackage.

Use the ‘hlint‘ command-line program to to check for stylistic issues in one
of your previous solutions.

Information about the hlint package can be found at

* hackage.haskell.org/package/hlint
* http://www.haskell.org/hoogle/?hoogle=hlint

etc.

The command used to install the hlint package is

> cabal install hlint

Although you may have to run

> cabal update

to ensure that your list of packages is up to date.

Once you have the hlint tool installed you can run it like so:

> hlint dragon.hs

    dragon.hs:35:17: Warning: Redundant $
    Found:
```

```
      flip zip (map toLower text) $ dragon_points
    Why not:
      flip zip (map toLower text) dragon_points
```

```
An open-ended question:

How would you go about publishing your own package to Hackage?
```

MOAR PLZ!

Hungry for more? Here are some extra sections that cover more advanced haskell awesomeness.

Testing

Haskell's type-system allows you to eliminate many kinds of errors that are usually combated using testing in more dynamic languages.

However, just because the type system can prevent some classes of error, doesn't mean that it can catch them all.

There are a couple of really useful approaches to testing used commonly by Haskell programmers:

- QuickCheck
- HUnit

QuickCheck focuses primarily on testing pure code, while HUnit provides means to test code that lives in the IO monad.

In the world of unit-testing, generating data can be a pain. Coming up with concise examples that test edge-cases and don't bother repeating useless properties is a real art-form that can take longer than writing the code to solve the problem at hand.

In many languages, frameworks and libraries have sprung up to aid with this problem by providing ways to generate various forms of valid and invalid test-data on demand.

An example of this form of library in Ruby is Factory Girl. You specify the structure and properties of the data you want generated, and using the helpers provided by the framework you will be able to generate many unit tests to ensure that your application's code behaves as expected!

But hang-on... Why would we go about telling a Haskell library what the structure of our data looks like when in a great-deal of cases - Haskell already knows?

This is where the `Arbitrary` type-class from QuickCheck steps up to bat for us!

Let's say that we have a function `hyphenate` that takes a sentence and turns the spaces into hyphens - There should be no trailing or leading hyphens unless they were already there to begin with, and there should be no instance of multiple hyphens in a row in our output - For example:

```
"hello world"      => "hello-world"
"say     whaaaaa?" => "say-whaaaat?"
```

We get pretty excited and code up an enterprise solution ~

```
hyphenate :: String -> String
hyphenate s = map (replace ' ' '-') s

replace :: Char -> Char -> Char -> Char
replace x y c | c == x    = y
              | otherwise = c
```

Great!

Those examples we looked at earlier kind of look like some test-data. Let's write some unit-tests.

```
test1 :: Bool
test1 = hyphenate "hello world" == "hello-world"

test2 :: Bool
test2 = hyphenate "say     whaaaaa?" == "say-whaaaat?"
```

Awwwwww yis.

. . . Except test2 fails spectacularly. . . Hmm. . .

So we fix our test-data, we fix our function (hopefully?) and code up a few more tests, only to find we missed another edge-case when our client comes back with some trailing hyphens and mad as a raging bull. . . There has to be a better way!

Enter QuickCheck -

Rather than writing tests for individual applications, we write properties that cover the various specifications of the desired behavior:

```
import Data.List

prop_no_spaces :: String -> Bool
prop_no_spaces s = not (elem ' ' (hyphenate s))

prop_no_leading :: String -> Bool
prop_no_leading s = case hyphenate s of (x:xs) -> x /= '-'
                                        _      -> True

prop_no_trailing :: String -> Bool
prop_no_trailing s = prop_no_leading (reverse s)

prop_no_multiple :: String -> Bool
prop_no_multiple s = not (elem "--" (concatMap tails (inits s)))
```

That's all well and good, but how does that help us?

Check this out!

```
> quickCheck +names hyphen-solution-and-tests.hs
[Prelude] > [1 of 1] Compiling Main ( hy.hs, interpreted )
Ok, modules loaded: Main.
[*Main] > prop_no_spaces: +++ OK, passed 100 tests.
[*Main] > prop_no_leading: *** Failed! (after 44 tests and 7 shrinks):
" "
[*Main] > prop_no_trailing: +++ OK, passed 100 tests.
[*Main] > prop_no_multiple: +++ OK, passed 100 tests.
[*Main] > Leaving GHCi.
```

ZOMG - So much free test data!

This found an edge-case for us without too much trouble, and what's more, narrowed it down to the minimal failing example!

```
If you want to be able to take advantage of QuickCheck in the
fashion performed in the example you will need to install
QuickCheck and quickcheck-script:

> cabal install quickcheck quickcheck-script
```

In order to take advantage of QuickCheck's data-generation for custom data-types - take a look at the Arbitrary type-class.

Given that the Fibbonacci sequence is defined as follows:

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)



Write a property checking that successive Fibbonacci term
pairs sum to the next term.
```

Symbolic Differentiation

Your Highschool math teacher has decided to introduce differentiation of single-variable polynomials...

This is in super easy once you grasped the concept:

Polynomial term differentiation:

$$a.x^n.\frac{d}{dx} = n.a.x^{n-1}$$

Linearity of differentiation:

$$(a+b)\frac{d}{dx} = a\frac{d}{dx} + b\frac{d}{dx}$$

But you don't wish to perform the undoubtedly large number of rote-examples required after your moment of epiphany.

```
differentiate :: Polynomial -> Polynomial
differentiate = ...
```

The data-types used to represent a `Polynomial` are as follows:

```
data Constant   = C Int             deriving (Show, Eq)
data Power      = U Int             deriving (Show, Eq)
data Term       = T Constant Power  deriving (Show, Eq)
data Polynomial = P [ Term ]        deriving (Show, Eq)
```

The `x` variable is implicit in the Term data-type.

```
Create a function "differentiate" that takes a Polynomial, and returns a new
Polynomial that has been differentiated with respect to x.
```

```
An open ended question:

What would an ADT look like that could model partially-differentiable
multi-varable equations?
```

Figure 1: Beam me Up

Making a Web-Site with Scotty

The Haskell library Scotty is similar to the ruby web-library Sinatra.

Scotty can be installed by using the following Cabal command:

```
> cabal install scotty
```

Scotty's behaviour is based around REST verbs and routes.

For example - A simple Hello-World website:

```
{-# LANGUAGE OverloadedStrings #-}

import Web.Scotty
import Data.Monoid (mconcat)

main = scotty 3000 $ do
  get "/:word" $ do
    beam <- param "word"
    html $ mconcat ["<h1>Scotty, ", beam, " me up!</h1>"]
```

If we inspect the type of `get` in GHCi we see this:

```
> import Web.Scotty
> :info get
get :: Action action => RoutePattern -> action -> ScottyM ()
    -- Defined in ‘scotty-0.4.6:Web.Scotty.Route’
```

The ActionM Action type-class instance allows us to perform any IO action we desire, such as printing to the console, reading files, etc - through the use of the liftIO function.

```
{-# LANGUAGE OverloadedStrings #-}

import Web.Scotty
import Control.Monad.IO.Class
import Data.Monoid


myRoute = get "/hello" $ do
  liftIO $ putStrLn "What is your name?"
  myName <- liftIO readLn
  html $ mconcat ["Your name is ", myName, "... Thanks!"]
```

Make a simple website using Scotty that shows the current time.

Check out the website project in the scaffold folder for an
example of how to put together a Scotty based web-project.

An open question:

What features do the more advanced Haskell web-frameworks include
that Scotty lacks?

Compression

In the scaffold folder `compression` there is a Cabal project that implements a compression strategy based on Huffman Coding.

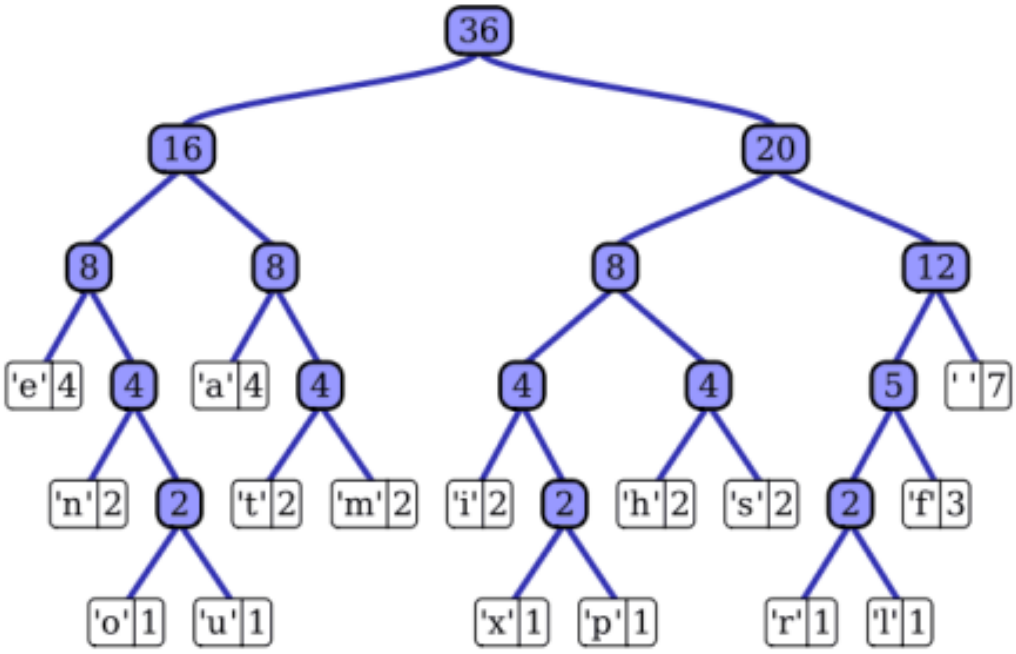


Figure 2: Huffman Tree

A Huffman Tree is built recursively by recursively constructing a tree from the two lowest-scored trees in a list, then adding the new tree to the list, until there is only one tree left.

```
data FingerTree k v = L k (Maybe v) | B k (FingerTree k v) (FingerTree k v)

type HuffmanTree = FingerTree (Sum Int) Char

instance Monoid k => Monoid (FingerTree k v)
  where x `mappend` y = B (getKey x `mappend` getKey y) x y
        mempty        = L mempty Nothing

getKey :: FingerTree t t1 -> t
getKey (L k _  ) = k
getKey (B k _ _) = k

step :: [HuffmanTree] -> [HuffmanTree]
```

```
step []    = []
step [x]   = [x]
step [x,y] = [ x <> y ]
step xs    = step (take 2 sorted) ++ drop 2 sorted
  where
    sorted = sortBy (comparing getKey) xs

build :: [HuffmanTree] -> HuffmanTree
build = head . fromJust . find ((<= 1) . length) . iterate step
```

The basic idea is that in a non-random string, some combinations of tokens will be more frequent than others, and as-such, common substrings can be represented with a short-hand encoding to save space.

The project contains two executables: `compression` and `decompression`. These accept data on STDIN and output data on STDOUT. `compression` accepts text on STDIN and outputs binary data. Decompression does the reverse.

```
The Haskell package "binary" exposes the module "Data.Binary" that
allows for very easy binary serialisation and parsing, with
most existing types already containing a marshelling implementation.
```

If you install these executables by using `cabal install`, then you will be able to use them in the following manner:

```
> # Compress
> compression < lyrics.txt > lyrics.txt.compressed
> # Decompress
> decompression < lyrics.txt.compressed
```

Top down, the compression algorithm works as follows:

```
* Determine the frequencies of the characters in the input text from STDIN
* Build a Huffman Tree from the frequency table
* Traverse the tree, outputting Bit-String encodings for each character
* Map a lookup of the encoding table over the input text
* Pack the resulting bits into a ByteString
* Attach a header (containing length, and frequency table)
* Output the result to STDOUT
```

With the decompression algorithm performing the following:

```
* Read a ByteString
* Unpack into header (length and frequency table) and body
* Translate frequency-table into encoding table as per compression
* For each encoding match against the start of the compressed body
* Output the character if there is a match
* Process the left-over body
```

```
Unfortunately, the compression algorithm contains a bug.

In the compression scaffolding project, locate and fix the bug
that causes the test-suite to fail.

You can run the tests by calling "cabal test" from within
the project folder.

-- Line 131 should be:

treeToBits t = map (second reverse) result
```

Although the algorithm itself is very basic, and the use of strings instead of binary-data is not a common way to perform compression, the structure itself is typical of a real-world Haskell project. The project structures itself around the Cabal toolkit, and uses several libraries. It exposes a library, several executables, and a test-suite.

```
An open question:

How would you implement a more efficient compression algorithm in Haskell?
```

Appendix

- Hutton's Razor
- Implementing Functional Languages: A Tutorial
- QuickCheck Wikipedia Page
- QuickCheck on Hackage
- 99 Haskell Problems
- Hackage
- BZip
- Scotty
- Sinatra
- REST
- HaskSymb - Symbolic Algebra
- Type-Classes

Contributors

Lyndon Maydwell	GitHub
Andy Kitchen	GitHub, Twitter
Alan Hawkins	GitHub

Volunteers - TODO