

# 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. It uses a narrative of a physicist attempting to understand how the world works.

## TODO: double check these items

### Note:

If you are attending the workshop, please attempt to have the required items from the 'Resources' section available for your use.

**TODO: Don't use underscores in pattern matching early on**

**TODO: Progressive Enhancement**

---

## Table of Contents

---

<a href="#">Readme</a>	Read this First
<a href="#">Resources</a>	Resources Available
<a href="#">Introduction</a>	Motivation, Overview, and Approach
<a href="#">Setup</a>	Setting up your Haskell environment
<a href="#">Opening Exercises</a>	Introductory Exercises
<a href="#">ADTs</a>	Modelling with data in Haskell
Typeclasses	Code-Reuse and Laws
Wildcard	Music
Monads	DSLs, IO, Do-Notation
Ecosystem	Resources and Community

---

## 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](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.

---

## Introduction

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.

---

## Setup

### ... and Sanity Checking

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

- GHC(i)

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
```

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"
```

## 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 1.16.0.2  
using version 1.16.0 of the Cabal library
```

---



## 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?

Note:

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](#)

Some examples of ADTs in Haskell:

```
data MyBool = MyTrue | MyFalse

should_I_eat_something_bigger_than_my_own_head :: MyBool
should_I_eat_something_bigger_than_my_own_head = MyFalse
```

You can also 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:

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

Ex. 3a.1 - Define your own 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?

---

## Opening Exercises

### Variables

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

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

```
my_variable = 1
```

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

Ex. 1.1 - Define your own variable.

### Type Signatures

You can annotate a definition with a valid type-signature:

## TODO: Decide if we should introduce how to define items in GHCi

```
my_variable :: Int
my_variable = 1
```

Ex. 1.2 - Give your variable a type-signature.

### String Literals

String literals look familiar:

```
my_string :: String
my_string = "hello world"
```

Ex. 1.3 - Define a variable containing a string.

### Tuples

Tuples look like this:

```
my_tuple_pair :: (Int, String)
my_tuple_pair = (1,"hello")

my_tuple_trio :: (Int, String, Int)
my_tuple_trio = (1,"hello",3)
```

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

Ex. 1.4 - Define a variable containing a tuple.

## Functions

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

```
my_function :: ... -> ...
my_function x y ... = ...
```

If a function `f` takes something of type `a` and returns something of type `b` we write the type signature as:

```
f :: a -> b
```

For example:

```
my_add :: Int -> Int -> Int
my_add x y = x + y
```

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

Ex. 1.5 - Define a function ‘`my_multiply`’ that multiplies 3 numbers.

Ex. 1.6 - Give your function a type-signature

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

## TODO: Decide if variables should include underscores

### 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 :: [Int]
list2 :: [Int]
list3 :: [String]

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

Ex. 1.7 - Define a variable containing a list.

Ex. 1.8 - Give your variable a type-signature.

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

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

Ex. 1.9 - 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!
```

Ex. 1.10 - 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

Ex. 1.11 -

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 Functions

For your reading pleasure, here are some definitions 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 find the signatures for these functions.

## TODO: Hide .answers in PDF

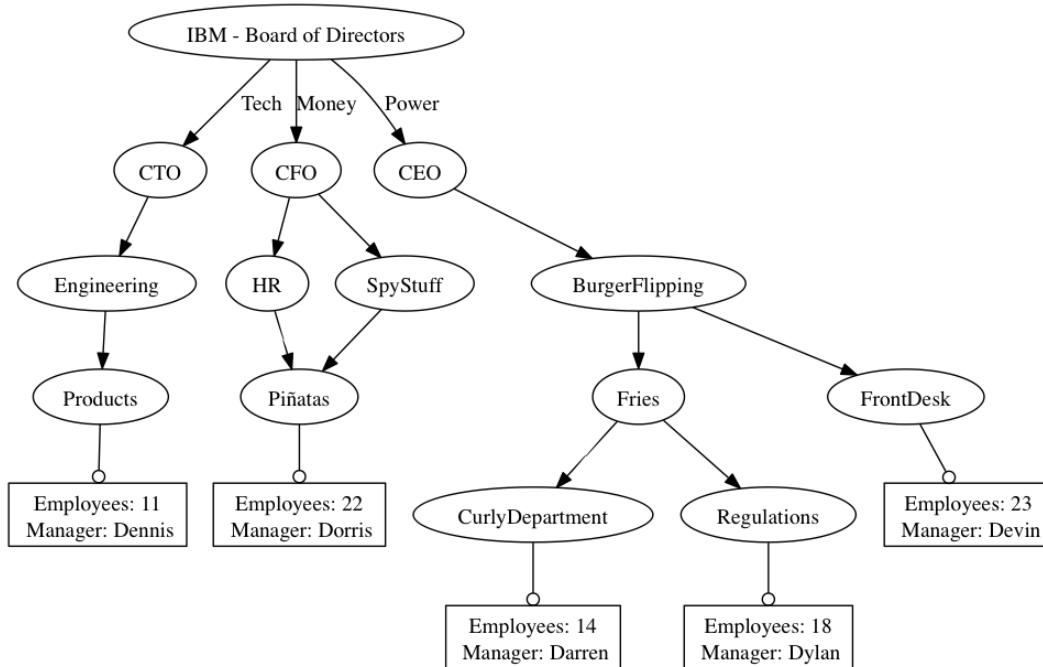
---

## Exercise - Org Chart

You are an employee of International Banking Machines.

The CEO has asked you to model the org-chart with a computer program in order to perform some analysis on the effectiveness of middle-managers.

Before you are able to perform your calculations, you will have to decide how you will model your data. Construct a Haskell data-type to represent the various types of employees and departments shown in the following diagram:



Construct a function that builds up your datastructure from the following plain-text input format:

### Employees

Name	Title	Salary	Manager
Dave	Serf	\$33K	Dorris
Tony	Baker	\$44K	Devin

Your function should have the following signature:

```
process :: String -> OrgChart
```



---

## Exercise - Music

Once you have a model that reaches a sufficient level of complexity you will find that your data-types become recursive.

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 ]
baa_baa = 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 "baa_baa.mid" (make_music 4 4 (Longer 4 baa_baa))
```

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

## Exercise - 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.

In this exercise you should develop a data-type to represent a single-variable polynomial expression.

After you feel confident that you can represent any such expression, write a function using the two rules listed to provide the following function:

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

---

# Appendix

Hutton's Razor