Haskell (by Mark Lentczner)

Haskell was developed by Isaac Newton in the 15th Century as a tool to help his investigations into the Alchemic arts. It was rediscovered in the 1980s by three Cambridge undergraduates who were browsing through Newton's laboratory notebooks looking for smutty jokes in the margins, and has since developed into an elaborate joke perpetrated by elite computer scientists who believe that predictable order of execution is contrary to natural law. The current version of Haskell is Haskell 1714, which adds syntactic sugar for Zygohistomorphic Premorphisms[1] *to the original language definition of 1693.*

[1] Thanks to Edward Kmett for making this "real".

Coping with CS3016

From previous experience, some students...

- are surprised by the difficulty of learning a new programming paradigm
- who have not done significant programming before the class may struggle
- assume they can get by without reading over the lecture notes between classes
- assume they can spend all night programming before a deadline and bash out something that would work well enough to get a passing grade.
- the above approach actually works for C, C++, Java, etc., which you have encountered before, but emphatically does not work for Haskell, because a logically incoherent design often results in a program which not only does not work, but **cannot be made to work** short of a complete rethink.

Functional Programming is Different

- *Balance of effort*: It's not only common but *desirable* that you spend much more time thinking about how to structure your code than writing the code - this point really has to be stressed
- *Workflow*: A lot of you will struggle initially with the workflow of writing functional programs, because the approach to programming you've learned from imperative languages is build & fix, and iterate on fixing until the program works
- " in an imperative language, you can slap together a program, and at some point where you realise that your design is incoherent, you can put in arbitrary mutations of global or shared state and (imagining for a moment that the program is a flow-graph) introduce arbitrary back-edges in the flow-graph (spaghetti code). In Haskell this is nearly impossible for a beginner, because you effectively can't mutate anything unless you're in a monadic context or you know enough Haskell to use the unsafe parts properly. "

FP Workflow

- *What do I have?* Identify the parts of the problem that need to be represented with datatypes
- *What do I want?* Model the solution to the problem as an instance of some datatype
- *How do I get there?* Imagine a function mapping the problem to the solution (both represented as an instance of some data type)
- Try to write this function as a sequence of steps
- Use this same process to construct each of the steps
- Eventually a step is some language or library primitive or a one-liner

Programming Toolbox (Secondary Learning)

- Development enviroment (OS / shell / editor)
- Source control - `git`
- Automated testing - `stack test`
- Build systems - `stack`
- Different kinds of safety (tests / types)

Online Resources for Haskell

There are lots of good resources online to help with learning Haskell:

- Searching libraries on Hackage:
  - **Hoogle**: http://www.haskell.org/hoogle/
  - **Hayoo!**: http://hayoo.fh-wedel.de
- Tutorials: (in order of difficulty)
  - **Learn You A Haskell**: http://learnyouahaskell.com/
  - **Haskell on Wikibooks**: https://en.wikibooks.org/wiki/Haskell
  - **FPComplete School of Haskell**: https://www.fpcomplete.com/school
  - **What I Wish I Knew When Learning Haskell**: http://dev.stephendiehl.com/hask/
  - **Haskell the Hard Way**: http://yannesposito.com/Scratch/en/blog/Haskell-the-Hard-Way/
- **Syntax Cheat Sheet**: http://cheatsheet.codeslower.com/

Haskell Language Structure

- Haskell is built on top of a simple functional language (Haskell "Core")
- A lot of syntactic sugar is added (e.g `"ab"` for `['a','b']` for `'a':'b':[]`).
- A large collection of standard types and functions are predefined and automatically loaded (the Haskell "Prelude")
- There are a vast number of libraries that are also available
- See `www.haskell.org`

Function definition

We have seen that functions are defined in Haskell as (sets of) equations:

```
sum []     = 0
sum (n:ns) = n + sum ns
```

Let's look in more detail at how these are written, and how the system selects which equation applies in any given case.

Patterns in Mathematics

In mathematics we often characterise something by laws it obeys, and these laws often look like patterns or templates:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)!, \qquad n > 0 \end{aligned}$$

$$\begin{aligned} \mathit{len}(\langle\rangle) &= 0 \\ \mathit{len}(\ell_1 \frown \ell_2) &= \mathit{len}(\ell_1) + \mathit{len}(\ell_2) \end{aligned}$$

Here $\langle\rangle$ denotes an empty list, and $\frown$ joins two lists together.
Pattern matching is inspired by this
(but with some pragmatic differences).

Cases by pattern matching

When there is more than one equation defining a function there must be some way to select between them.
One particularly convenient way to do this is *pattern matching*:

```
myfun 0 = 0
myfun 1 = 1
myfun n = myfun (n-2)
```

These three equations give a definition for a function. When the function is applied to an integer Haskell selects one of the equations by matching the actual parameter against the patterns.

Using `myfun`

Consider application of `myfun` to `3`:

```
> myfun 3
```

Haskell tries to match `3` against `0`, and fails.
Haskell tries to match `3` against `1`, and fails.
Haskell tries to match `3` against `n`, and succeeds, binding `n` to `3`.
The result, according to the righthand side (rhs) is:

```
myfun (3-2)

myfun 1     -- subtraction gets done
```

Haskell tries to match `1` against `0`, and fails.
Haskell tries to match `1` against `1`, and succeeds.

```
1
```

What does `myfun` do, in simple terms? What is `myfun (-1)`?

Cases by pattern matching

Patterns can be used to give an elegant expression to certain functions, for instance we can define a function over two `Bool` arguments like this:

```
and True True = True
and _    _    = False
```

The special pattern "`_`" will match any value without binding it to a name. It is usually used to indicate that the value is not needed on the right-hand side.

Partial functions

We can select equations using quite complex patterns. This set is non-exhaustive, and so the `sum3` function is *partial*.

```
sum3 [] = 0
sum3 [x] = x
sum3 [x,y] = x+y
sum3 [x,y,z] = x+y+z
```

It is an error to apply a partial function outside the domain it is defined for (division is another example of a partial function). So `sum3 [1,2,3,4]` will give a runtime error.

Pattern matching on structures

Here is a (partial) function from the Prelude:

```
> head [1,2,3]
1
```

It provides the first element of a list. We can define it using pattern matching:

```
head (x:xs) = x
```

The pattern in this case is based on something called a *constructor*.

Pattern matching on structures

The pattern match relies on the fact that lists are built out of individual elements using "`[]`" (empty list) and "`:`" ("cons"[2]). It's only for convenience that we use the comma-list notation.

```
> 1:[]
[1]
> 2:3:[]
[2,3]
> 1:[2,3]
[1,2,3]
```

So in the pattern match for `head`,

```
head (x:xs) = x
```

an argument like `[1,2,3]` would result in `x` being associated with `1` and `xs` being associated with `[2,3]`

[2] pron. "conz"

Order in Pattern Matching

- Patterns are matched in order until a match occurs, or all fail to match (runtime error)
  - we don't re-evaluate arguments from scratch for later patterns
  - an early general match (e.g. `x:xs`) masks a later more specific one (e.g. `1:xs`)
- This "first-come first-served" approach handles overlapping patterns gracefully
- Haskell can warn about overlapping and (some) missing patterns.

Definition by cases

Often we want to have different equations for different cases. Mathematically we sometimes write something like this:

$$signum(x) = \begin{cases} 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

This indicates that there are three different cases in the definition of *signum*, and gives a rule for choosing which case should applied in any particular application.

Cases by guarded alternatives

Another way to make choices is to use *guarded alternatives* (this looks rather more like the mathematical style we saw earlier):

```
signum x | x < 0  = -1
         | x == 0 = 0
         | x > 0  = 1
```

The guards are boolean expressions.
If a guard is true the corresponding right-hand side will be selected as the definition for the function.
We can't use pattern matching for the above function (why not?)

Cases by guarded alternatives

Each guard is tested in turn, and the first one to match selects an alternative. This means that it is OK to have a guard that would always be true, as long as it is the *last* alternative.
So the previous definition could have been written like this:

```
signum x | x < 0   =  -1
         | x == 0  =  0
         | True    =  1
```

For readability the name `otherwise` is allowed as a synonym for `True`:

```
signum x | x < 0      =  -1
         | x == 0     =  0
         | otherwise  =  1
```

Cases by guarded alternatives

We can use guards to select special cases in functions. This function is `True` when the year number is a leap year:

```
leapyear :: Int -> Bool
leapyear y | mod y 400 == 0  =  True   -- 2000 was
           | mod y 100 == 0  =  False  -- 1900 wasn't
           | mod y 4 == 0    =  True   -- 2016 is
           | otherwise       =  False  -- 2017 won't be
```

A more compact form

```
leapyear y | y `mod` 100 == 0 = y `mod` 400 == 0
           | otherwise        = y `mod` 4 == 0
```

> In Haskell any function of two arguments may be written infix if it is surrounded by backquotes, which is why `` `mod` `` is OK.

Cases by guarded alternatives

Guards and patterns can be combined:

```
startswith _ [] = False
startswith c (x:xs) | x == c    = True
                    | otherwise = False
```

First the patterns are matched; when an equation is found the guards are evaluated in order in the usual way.
If no guard matches then we return to the pattern matching stage and try to find another equation.