Lesson #1: Getting Acquainted with Haskell

What's It All About?

1. The first thing you will need to do in order that you can do some Haskell programming is to download a Haskell interpreter. The first part of this lesson points you in the right direction to do just that.
2. Haskell is a sign of functional programming, as well as a powerful language in which to do functional programming. Due to its nature as a sign of functional programming, it makes sense to talk a bit about functional programming.
3. The characteristics of Haskell that are associated with functional programming are enumerated.
4. Interactions that feature values and expressions of values are presented as a first step in thinking thoughts about the language.
5. Simple definitions involving numbers are presented.
6. Words of warning. Appeal for patience. Words of encouragement.

The software

You will need a Haskell in order to do some Haskell programming. The Glasgow Haskell Compiler, GHC, is what most people use, and is **highly recommended**! You can find the software here: `https://www.haskell.org/` Like Racket and Prolog, I have also asked that it be installed on the CS machines, so you can work on those machines if you like.

You will also need a good text editor with which to create your Haskell source code. I like emacs, but you might like something else.

A few words on functional programming

> "When the limestone of imperative programming has worn away, the granite of functional programming will be revealed underneath!" – Simon Peyton Jones

`https://www.techrepublic.com/article/whats-the-future-of-programming-the-answer-lies-in-functional-languages/`

What is functional programming? **Functional programming** is a style of programming in which the basic method of computation is the application of functions (honest functions) to arguments.

What is a functional programming language? A **functional programming language** is a programming language that strongly supports and encourages the functional style of programming.

Most imperative languages provides some sort of support for programming with functions. But they may, for example, not allow functions to take functions as values or return functions as arguments. Or they may not allow you

to store functions in data structures, such as lists. Or they may allow a "function" to obtain data from somewhere other than its parameters. This would be considered weak support for the functional style, at best.

That said, it is reassuring that most programming languages seem to be scaling up their support for functional programming in recent years!

History of Functional Programming Languages

((Taken largely from Graham Hutton.))

1. In the 1930s, Alonzo Church developed the lambda calculus, a simple but powerful mathematical theory of functions.
2. In the 1950s, inspired by the lambda calculus, John McCarthy developed Lisp ("LISt Processor"), generally regarded as the first functional programming language, even though it incorporated the concept of variable assignment.
3. In the 1960s, Peter Landin developed ISWIM ("If you Se What I Mean"), the first pure functional programming language, based strongly on the lambda calculus and having no variable assignments.
4. In the 1970s, John Backus developed FP ("Functional Programming"), a functional programming language that emphasized higher order functions ad reasoning about programs.
5. In the 1970s, Robin Milner introduced ML ("Meta-Language"), the first of the modern functional programming languages, which introduced polymorphic types and type inference.
6. In the 1970s and 1980s, David Turner introduced a number of functional programming languages which featured lazy evaluation, most notably Miranda.
7. Beginning in 1987, a group of programming language researchers started work on Haskell. Just 15 years later a stable version was released. Work on development of the language continues.

Some Characteristics of Haskell and Functional Languages

((Taken largely from Graham Hutton.))

1. Concise programs.
2. Recursive functions.
3. Powerful type system.
4. List comprehensions (power operators for list processing).
5. Higher-order functions.
6. Artful integration of nonfunctional programming with the purely functional programming.
7. Lazy Evaluation (evaluation strategy in which evaluation is delated until a value is needed).

First Haskell Session – Values and Expressions

```
bash-3.2$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> 496
496
Prelude> 0.123456789
0.123456789
Prelude> pi
3.141592653589793
Prelude> 5 + 4
9
Prelude> ( pi * ( 10 ^ 2 ) )
314.1592653589793
Prelude> True
True
Prelude> False
False
Prelude> "Desde El Alma"
"Desde El Alma"
Prelude> 'X'
'X'
Prelude> 'X' == 'x'
False
Prelude> ( pi > 3 ) && ( pi < 4 )
True
Prelude> "Desde El Alma" < "Esta Noche de Luna"
True
Prelude> length "Learn You a Haskell for Great Good!"
35
Prelude> :quit
Leaving GHCi.
bash-3.2$
```

First Haskell Functions – Numeric Processing

Code

```
--------------------------------------------------------------------------
--------------------------------------------------------------------------
--- first_functions.hs contains some simple numeric function
--- definitions

--------------------------------------------------------------------------
-- Thing 1

square x = x * x

distance (x1,y1) (x2,y2) = sqrt ( ds1 + ds2 )
   where ds1 = square ( x1 - x2 )
         ds2 = square ( y1 - y2 )

perimeter (x1,y1) (x2,y2) (x3,y3) = a + b + c
   where a = distance (x1,y1) (x2,y2)
         b = distance (x2,y2) (x3,y3)
         c = distance (x3,y3) (x1,y1)

--------------------------------------------------------------------------
-- Thing 2

total_word_length w1 w2 = ( l1 + l2 )
   where l1 = length w1
         l2 = length w2

average_word_length w1 w2 = fromIntegral ( l1 + l2 ) / 2.0
   where l1 = length w1
         l2 = length w2

--------------------------------------------------------------------------
-- Thing 3

factorial :: Integer -> Integer
factorial x = if ( x == 1 ) then 1 else x * factorial ( x - 1 )
```

Demo for Thing 1 Code

```
bash-3.2$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help

Prelude> :set prompt ">>> "
```

```
>>> :load "first_functions.hs"
[1 of 1] Compiling Main             ( first_functions.hs, interpreted )
Ok, one module loaded.

>>> square 5
25
>>> square 7.5
56.25
>>> :t square
square :: Num a => a -> a

>>> distance (0,0) (1,1)
1.4142135623730951
>>> distance (1.5,1.5) (3.5,3.5)
2.8284271247461903
>>> :t distance
distance :: Floating a => (a, a) -> (a, a) -> a

>>> perimeter (0,0) (2,1) (4,0)
8.47213595499958
>>> perimeter (0,0) (1.5,1.5) (0,-3)
9.864736833812211
>>> :t perimeter
perimeter :: Floating a => (a, a) -> (a, a) -> (a, a) -> a

>>> :q
Leaving GHCi.
bash-3.2$
```

Demo for Thing 2 Code

```
bash-3.2$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> :set prompt ">>> "
>>> :load "first_functions.hs"
[1 of 1] Compiling Main             ( first_functions.hs, interpreted )
Ok, one module loaded.
>>> total_word_length "lion" "tiger"
9
>>> :t total_word_length
total_word_length
  :: (Foldable t1, Foldable t2) => t2 a1 -> t1 a2 -> Int
>>> average_word_length "lion" "tiger"
4.5
>>> :t average_word_length
average_word_length
  :: (Fractional a1, Foldable t1, Foldable t2) =>
     t2 a2 -> t1 a3 -> a1
>>> :quit
Leaving GHCi.
bash-3.2$
```

Demo for Erroneous Thing 2 Code

Suppose we had left out the `fromIntegral` function call in the `average_word_length` function definition, which seems like a reasonable thing to do if you do not know Haskell pretty well. What would happen? We can delete it and see!

```
bash-3.2$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> :set prompt ">>> "
>>> :load "first_functions.hs"
[1 of 1] Compiling Main             ( first_functions.hs, interpreted )

first_functions.hs:26:29: error:
     Could not deduce (Fractional Int) arising from a use of /
      from the context: (Foldable t, Foldable t1)
        bound by the inferred type of
                   average_word_length :: (Foldable t, Foldable t1) =>
                                          t1 a -> t a1 -> Int
        at first_functions.hs:(26,1)-(28,23)
     In the expression: (l1 + l2) / 2.0
      In an equation for average_word_length:
          average_word_length w1 w2
            = (l1 + l2) / 2.0
            where
                l1 = length w1
                l2 = length w2
   |
26 | average_word_length w1 w2 = ( l1 + l2 ) / 2.0
   |                             ^^^^^^^^^^^^^^^^^
Failed, no modules loaded.
>>> --
>>> -- Just added back the call to fromIntegral
>>> --
>>> :r
[1 of 1] Compiling Main             ( first_functions.hs, interpreted )
Ok, one module loaded.
>>> average_word_length "good" "grief"
4.5
>>> :quit
Leaving GHCi.
bash-3.2$
```

The morals of this story are:

1. You want to get used to Haskell spewing out seemingly unintelligible error messages at you as you are getting acquainted with the language. Those who understand the type system quite well tend not to receive many of these.
2. You will quite often want to explicitly convert a numeric type to another numeric type in Haskell prior to calling a function.

Demo for Thing 3 Code

```
bash-3.2$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> :set prompt ">>> "
>>> :load "first_functions.hs"
[1 of 1] Compiling Main             ( first_functions.hs, interpreted )
Ok, one module loaded.
>>> factorial 1
1
>>> factorial 10
3628800
>>> factorial 100
933262154439441526816992388562667004907159682643816214685929638
952175999932299156089414639761565182862536979208272237582511852
109168640000000000000000000000000
>>> :type factorial
factorial :: Integer -> Integer
>>> :quit
Leaving GHCi.
bash-3.2$
```

Tip on a Haskell Resource