Basic Stuff

- Due to layout rules Haskell syntax is rather elegant and generally eazy to understand. The import thing is to indent consistently, becuase, unlike other languages, indentation matters.

- Like ML Functions can either be defined in a curried form:

  ```
  add x y = x + y
  ```

  Or an un-curried form using turples, which work the same way as they do in ML.

  ```
  add (x,y) = x + y
  ```

  However, unlike ML, functions are generally defined in the un-curried form.

- Functions can also be defined without a name

  ```
  \x y -> x + y
  ```

- Haskell also has infix operators which are really just functions.

  Which can be partly applied just like curied functions using a compact syntax.

  ```
  (+) = \x y -> x + y
  (5+) = \y -> 5 + x
  ```

  It is also possable to define your own infix operators:

  ```
  infixl <? -- infix, left binding

  x <? y | x < y     = x
         | otherwise = y
  ```

  Which is definding the "min" operator. The expression "20 <? 30 <? 10" will then evaluate to 10 as expected.

- Pattens and wildcards behave the same way they do in ML.

  ```
  len []     = 0
  len (_:xs) = 1 + len xs
  ```

  However, Haskell also has pattern guards which are an elegant form of "if then else".

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

  But it is not always convenient to have to define a separate function every time a patern match/guard is needed. For this, haskell provided the case statement.

  ```
  len lst = case lst of
              []     -> 0
              (_:xs) -> 1 + len xs

  abs x = case x of
            x | x >= 0 -> x
              | x <  0 -> -x
  ```

Haskell even has the "if then else" statment, however it is really just a shorthand for:

```
case <exp> of
  true  -> <then clause>
  false -> <else clause>
```

- A let clause can be used to define bindings much like in ML.

```
let y   = a * b
    f x = (x + y) / y
in f c + f d
```

In the contex of functions and case expressions, a where clause can also be used which is similar to let except that the bindings come after the expression.

```
fun x = f c + f d
  where y   = a * b
        f x = (x + y) / y
```

A where cause, unlike the let clause, can also be used to scope bindings over several guarded equations:

```
f x y  |  y > z   =  ...
       |  y == z  =  ...
       |  y < z   =  ...
     where z = x * x
```