Basic Stuff

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

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

  \[ \text{add } x \ y = x + y \]

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

  \[ \text{add } (x,y) = x + y \]

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

- Functions can also be defined without a name

  \[ \backslash x \ y \to x + y \]

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

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

  \[ (+) = \backslash x \ y \to x + y \]

  \[ (5+) = \backslash y \to 5 + x \]

  It is also possible to define your own infix operators:

  \[ \text{infixl} \ <? \ \text{-- infix, left binding} \]

  \[ x <? y \mid x < y \quad = x \]

  \[ \mid \text{otherwise} \quad = y \]

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

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

  \[ \text{len } [] = 0 \]

  \[ \text{len } (_\colon xs) = 1 + \text{len } xs \]

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

  \[ \text{sign } x \mid x > 0 \quad = \quad 1 \]

  \[ \mid x == 0 \quad = \quad 0 \]

  \[ \mid x < 0 \quad = \quad -1 \]

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

  \[ \text{len } \text{lst} = \text{case } \text{lst of} \]

  \[ \quad [] \to 0 \]

  \[ \quad (_\colon xs) \to 1 + \text{len } xs \]

  \[ \text{abs } x = \text{case } x \text{ of} \]

  \[ \quad x \mid x >= 0 \to x \]

  \[ \quad \mid x < 0 \to -x \]
Haskell even has the "if then else" statement, however it is really just a shorthand for:

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

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

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

In the context 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.

```haskell
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:

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