Haskell Layout Rule [*H2010* 2.7]

- Some Haskell syntax specifies lists of declarations or actions as follows: $\{item_1 ; item_2 ; item_3 ; \ldots ; item_n\}$
- In some cases (after keywords `where`, `let`, `do`, `of`), we can drop `{`, `}` and `;`.
- The layout (or "off-side") rule takes effect whenever the open brace is omitted.
  - When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments).
  - For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted);
  - if it is indented the same amount, then a new item begins (a semicolon is inserted);
  - and if it is indented less, then the layout list ends (a close brace is inserted).

Layout Example

Offside rule (silly) example: consider

let $x = y + 3 \wedge z = 10 \wedge f(a) = a + 2z$ **in** $f(x)$

- Full syntax:

```
let {  x = y + 3 ; z = 10; f a = a + 2 * z } in f x
```

- Using Layout:

```
let x = y + 3
    z = 10
    f a = a + 2 * z
in f x
```

- Using Layout (alternative):

```
let
  x =  y + 3
  z = 10
  f a
   = a + 2 * z
in f x
```

Local Declarations [*H2010* 3.12]

- A let-expression has the form:

$$\texttt{let}\ \{d_1 ; \ldots ; d_n\}\texttt{in}\ e$$

d_i are declarations, e is an expression.
The offside-rule applies.

- Scope of each d_i is e and righthand side of all the d_is (mutual recursion)
- Example: $ax^2 + bx + c = 0$ means $x = \frac{-b \pm (\sqrt{b^2 - 4ac})}{2a}$
-
```
solve a b c
 = let twoa = 2 * a
       discr = b*b - 2 * twoa * c
       droot = sqrt discr
   in ((droot-b)/twoa , negate ((droot+b)/twoa))
```

Local Declarations [*H2010* 3.12]

- A where-expression has the form:

$$\texttt{where}\ \{d_1 ; \ldots ; d_n\}$$

d_i are declarations.
The offside-rule applies.

- Scope of each d_i is the expression that *precedes* `where` and righthand side of all the d_is (mutual recursion)
-
```
solve a b c
 = ((droot-b)/twoa , negate ((droot+b)/twoa))
 where
   twoa = 2 * a
   discr = b*b - 2 * twoa * c
   droot = sqrt discr
```

`let` ([*H2010* 3.12]) vs. `where` [*H2010* 4.?]

- What is the difference between `let` and `where` ?
- The `let ...in ...` is a full expression and can occur anywhere an expression is expected.
- The `where` keyword occurs at certain places in declarations

$$\ldots\texttt{where}\{d_1\,;\ldots;d_n\}$$

  of
  - case-expressions [*H2010* 3.13]
  - `modules` [*H2010* 4]
  - `classes` [*H2010* 4.3.1]
  - `instances` [*H2010* 4.3.2]
  - function and pattern righthand sides (rhs) [*H2010* 4.4.3]
- Both allow mutual recursion among the declarations.

Case Expression [*H98* 3.13]

- A case-expression has the form:

$$\texttt{case}\ e\ \texttt{of}\ \{p_1\ \texttt{->}\ e_1\,;\ldots;p_n\ \texttt{->}\ e_n\}$$

  p_i are patterns, e_i are expressions.
  The offside rule applies.

```
odd x =
  case (x `mod` 2) of
    0 -> False
    1 -> True
```

```
empty x =
  case x of
  [] -> True
  _  -> False
```

```
vowel x =
  case x of
    'a' -> True
    'e' -> True
    'i' -> True
    'o' -> True
    'u' -> True
    _   -> False
```

Implementing `splitAt` recursively

- `splitAt ::  Int -> [a] -> ([a],[a])`
  Let `(xs1,xs2) = splitAt n xs` below.
  Then `xs1` is the first `n` elements of `xs`.
  Then `xs2` is `xs` with the first `n` elements removed.
  If `n >= length xs` then `(xs1,xs2) = (xs,[])`.
  If `n <= 0` then `(xs1,xs2) = ([],xs)`.
-
```
splitAt n xs | n <= 0  =  ([],xs)
splitAt _ []           =  ([],[])
splitAt n (x:xs)
  = let (xs1,xs2) = splitAt (n-1) xs
    in  (x:xs1,xs2)
```
- How long does `splitAt n xs` take to run?
- It takes time proportional to `n` or `length xs`, whichever is shorter, which is twice as fast as the version using `take` and `drop` explicitly!

Switcheroo!

- Can we implement `take` and `drop` in terms of `splitAt`?
- Hint: the Prelude provides the following:
```
fst :: (a,b) -> a
snd :: (a,b) -> b
```
- Solution:
```
take n xs = fst (splitAt n xs)
drop n xs = snd (splitAt n xs)
```
- How does the runtime of these definitions compare to the direct recursive ones?

Higher Order Functions

What is the difference between these two functions?

```
add x y = x + y
add2 (x, y) = x + y
```

We can see it in the types; `add` takes one argument at a time, returning a function that looks for the next argument.
This concept is known as "Currying" after the logician Haskell B. Curry.

```
add :: Integer -> (Integer -> Integer)
add2 :: (Integer, Integer) -> Integer
```

Remember, any type `a -> (a -> a)` can also be written `a -> a -> a`. The function type arrow associates to the right.

In Haskell functions are *first class citizens*. In other words, they occupy the same status in the language as values: you can pass them as arguments, make them part of data structures, compute them as the result of functions...

```
add3 :: (Integer -> (Integer -> Integer))
add3 = add
```

```
> add3 1 2
3

(add3) 1 2
  ==> add 1 2
  ==> 1 + 2
```

Notice that there are no parameters in the definition of `add3`.

A function with multiple arguments can be viewed as a function of one argument, which computes a new function.

```
add 3 4
  ==> (add 3) 4
  ==> ((+) 3) 4
```

The first place you might encounter this is the notion of *partial application*:

```
increment :: Integer -> Integer
increment = add 1
```

If the type of `add` is `Integer -> Integer -> Integer`, and the type of `add 1 2` is `Integer`, then the type of `add 1` is?
It is `Integer -> Integer`

Some more examples of partial application:

```
second :: [a] -> a
second = head . tail
```

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

An infix operator can be partially applied by taking a *section*:

```
increment = (1 +) -- or (+ 1)

addnewline = (++"\n")

double :: Integer -> Integer
double = (*2)
```

```
> [ double x | x <- [1..10] ]
[2,4,6,8,10,12,14,16,18,20]
```

Functions can be taken as parameters as well.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)

addtwo = twice increment
```

Here we see functions being defined as functions of other functions!

Composition

In fact, we can define composition using this technique:

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g x = f (g x)

twice f = f `compose` f
```

Haskell permits the definition of infix functions:

```
(f ! g) x = f (g x)
twice f = f!f
```

Function composition is in fact part of the Haskell Prelude:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```