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.

Conditionals [*H2010* 3.6]

- For expressions, we can write a conditional using `if` ...`then`...`else`

  $$exp \;\rightarrow\; \texttt{if}\ exp\ \texttt{then}\ exp\ \texttt{else}\ exp$$

- The else-part is compulsory, and cannot be left out (why not?)
- The (boolean-valued) expression after `if` is evaluated:
  If true, the value is of the expression after `then`
  If false, the value is of the expression after `else`

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
```

Lambda abstraction

Since functions are first class entities, we should expect to find some notation in the language to create them from scratch. There are times when it is handy to just write a function "inline". The notation is:

```
\ x -> e
```

where `x` is a variable,
and `e` is an expression that (usually) mentions `x`.
This notation reads as "the function taking `x` as input and returning `e` as a result". We can have nested abstractions

```
\ x -> \ y -> e
```

Read as "the function taking `x` as input and returning a function that takes `y` as input and returns `e` as a result".
There is syntactic sugar for nested abstractions:

```
\ x  y -> e
```

It's just notation!

The following definition groups are equivalent:

```
sqr     =  \ n -> n * n
sqr n   =  n * n


add     =  \ x y -> x+y
add x   =    \ y -> x+y
add x y =           x+y
```

Lambda application

In general, an application of a lambda abstraction to an argument looks like:

```
(\ x -> x + x) a
       ^--e--^
-- Applied:
(a + a)
```

The result is a copy of `e` where any free occurrence of `x` has been replaced by `a`. This is just the β-reduction rule of the lambda calculus.

Factorial: a comparison

A simple definition of factorial, ignoring negative numbers, is the following:

```
fac 0 = 1
fac n = n * fac (n-1)
```

But what is `fac`?

```
fac = ..... ?
```

```
fac = \n -> case n of
             0 -> 1
             m -> m * fac (m-1)
```

Lists: `Haskell` vs. `Prolog`

Mathematically we might write lists as items separated by commas, enclosed in angle-brackets

$$\sigma_0 = \langle\rangle \quad \sigma_1 = \langle 1 \rangle \quad \sigma_2 = \langle 1, 2 \rangle \quad \sigma_3 = \langle 1, 2, 3 \rangle$$

Haskell	Prolog
`s0 = []`	`S0 = []`
`s1 = 1:[]  or  [1]`	`S1 = [1]`
`s2 = 1:2:[]  or  [1,2]`	`S2 = [1,2]`
`s3 = 1:2:3:[] or [1,2,3]`	

`1:2:3:[] is really (1:(2:(3:[])))`

Patterns

Haskell			Prolog		
`[]`	`(x:xs)`	`(x:y:xs)`	`[]`	`[X|Xs]`	`[X,Y|Xs]`
	`[x]`	`[x,y]`		`[X]`	`[X,Y]`

Polymorphism brings great power!

What is the type of `length`?

```
length [] = 0
length (x:xs) = 1 + length xs
```

It's `length :: [a] -> a`
One piece of code can handle all lists, no matter what their contents!

"Polymorphism sets us free" , ...or does it?

`f11 :: a -> a`

```
f11 x = ?
```

We are totally constrained here, and all we can do is reproduce the input:

```
f11 x = x
```

`f11` is in the Prelude, where it is called `id`.

`f12 :: a -> b`

```
f12 x = ?
```

Here all we can do is induce a runtime failure

```
f12 x = undefined
```

This will happen for values/functions with types: `a`, `a->b`, `a->b->c`, `a->b->c->d` and so on ...

`f121 :: a -> b -> a`

```
f121 x y = ?
```

We are totally constrained here, and all we can do is reproduce the first input:

```
f121 x y = x
```

`f121` is in the Prelude, where it is called `const`.

f122 :: a -> b -> b

```
f122 x y = ?
```

We are totally constrained here, and all we can do is reproduce the second input:

```
f122 x y = y
```

`f122` is in the Prelude, where it is called `seq`, but it's strange!

f111 :: a -> a -> a

```
f111 x y = ?
```

We are less constrained here, and have two choices:

```
f111 x y = x
OR
f111 x y = y
```

`f111`, first version, is in the Prelude, where it is called `asTypeOf`.

f321 :: (a -> b -> c) -> (a,b) -> c

```
f321 f (x,y) = ?
```

We are totally constrained here, and all we can do apply `f` to the other inputs

```
f321 f (x,y) = f x y
```

`f321` is in the Prelude, where it is called `uncurry`.

f213 :: ((a,b) -> c) -> a -> b -> c

```
f213 g x y = ?
```

We are totally constrained here, and all we can do apply `g` to a paior built from the other inputs

```
f213 g x y = g (x,y)
```

`f213` is in the Prelude, where it is called `curry`.

`f1221 :: (a -> b -> c) -> b -> a -> c`

```
f1221 f x y = ?
```

We are totally constrained here, and all we can do is apply `f` to `x` and `y` in reversed order.

```
f1221 f x y = f y x
```

`f1221` is in the Prelude, where it is called `flip`.

Polymorphism is a constraint

A polymorphic type in fact drastically reduces the options for coding a function because such code cannot use functions that require specific types (or type classes).