Haskell Language Structure

- Haskell is built on top of a simple functional language (Haskell "Core")
- A lot of syntactic sugar is added
  (e.g `"ab"` for `['a','b']` for `'a':'b':[]`).
- A large collection of standard types and functions are predefined and automatically loaded (the Haskell "Prelude")
- There are a vast number of libraries that are also available
- See `www.haskell.org`

Patterns in Mathematics

In mathematics we often characterise something by laws it obeys, and these laws often look like patterns or templates:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)!, \qquad n > 0 \end{aligned}$$

$$\begin{aligned} len(\langle\rangle) &= 0 \\ len(\ell_1 \frown \ell_2) &= len(\ell_1) + len(\ell_2) \end{aligned}$$

Here $\langle\rangle$ denotes an empty list, and $\frown$ joins two lists together.
Pattern matching is inspired by this
(but with some pragmatic differences).

Factorial! as Patterns

Math:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)!, \qquad n > 0 \end{aligned}$$

Haskell:

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

Length with Patterns

Math:

$$\begin{aligned} len(\langle\rangle) &= 0 \\ len(\ell_1 \frown \ell_2) &= len(\ell_1) + len(\ell_2) \\ len(\langle\_\rangle \frown \ell) &= 1 + len(\ell) \\ len(\langle\_\rangle) &= 1 \end{aligned}$$

Haskell:

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

The special pattern "_" will match any value without binding it to a name. It is usually used to indicate that the value is not needed on the right-hand side.

Compact "Truth Tables"

Patterns can be used to give an elegant expression to certain functions, for instance we can define a function over two `Bool` arguments like this:

```
and True True = True
and _    _    = False
```

Compact "Truth Tables"

Patterns can be used to give an elegant expression to certain functions, for instance we can define a function over two `Bool` arguments like this:

```
and True True = True
and _    _    = False
```

Pattern Matching

- Inputs: a pattern, and a Haskell expression
- Output: either a "fail",
  or a "success",
  with a binding from each variable to the part of the expression it matched.

Example:
pattern `('c':zs)`,
matches expression `'c':('a':('t':[]))`,
with `zs` bound to `'a':('t':[])`

Patterns in Haskell

- We can build patterns from atomic values, variables, and certain kinds of "constructions".
- An atomic value, such as `3`, `'a'` or `"abc"` can only match itself
- A variable, or the wildcard `_`, will match anything.
- A "construction" is either:
  1. a tuple such as `(a,b)` or `(a,b,c)`, etc.,
  2. a list built using `[]` or `:`,
  3. or a user-defined datatype.

  A construction pattern matches if all its sub-components match.

Pattern Sequences

- Function definition equations may have a sequence of patterns (e.g., the `and` function example.)
- Each pattern is matched against the corresponding expression, and all such matches must succeed.
  One binding is returned for all of the matches
- Any given variable may only occur once in any pattern sequence
  (it can be reused sequence in a different equation.).

Pattern Examples

- Expect three arbitrary arguments (of the appropriate type!)
  ```
  myfun x y z
  ```
- Illegal — if we want first two arguments to be the same then we need to use a conditional (somehow).
  ```
  myfun x x z
  ```
- First argument must be zero, second is arbitrary, and third is a non-empty list.
  ```
  myfun 0 y (z:zs)
  ```
- First argument must be zero, second is arbitrary, and third is a non-empty list, whose first element is character 'c'
  ```
  myfun 0 y ('c':zs)
  ```
- First argument must be zero, second is arbitrary, and third is a non-empty list, whose tail is a singleton.
  ```
  myfun 0 y (z:[z'])
  ```

Pattern Matching (summary)

- Pattern-matching can *succeed* or *fail*.
- If successful, a pattern match returns a (possibly empty)*binding*.
- A binding is a mapping from (pattern) variables to values.
- Examples:

Patterns	Values	Outcome
x (y:ys) 3	99 [] 3	Fail
x (y:ys) 3	99 [1,2,3] 3	Ok, $x \mapsto 99, y \mapsto 1, ys \mapsto [2,3]$
x (1:ys) 3	99 [1,2,3] 3	Ok, $x \mapsto 99, ys \mapsto [2,3]$

- Binding $x \mapsto 99, y \mapsto 1, ys \mapsto [2,3]$
  can also be written as substitution $[99, 1, [2,3]/x, y, ys]$

Definition by cases

Often we want to have different equations for different cases.
Mathematically we sometimes write something like this:

$$signum(x) = \begin{cases} 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Signum as Cases

Math:

$$signum(x) = \begin{cases} 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Haskell:

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

If all else fails...

Each guard is tested in turn, and the first one to match selects an alternative. This means that it is OK to have a guard that would always be true, as long as it is the *last* alternative.
So the previous definition could have been written like this:

```
signum x | x < 0   =  -1
         | x == 0  =   0
         | True    =   1
```

For readability the name `otherwise` is allowed as a synonym for `True`:

```
signum x | x < 0      =  -1
         | x == 0     =   0
         | otherwise  =   1
```

Cases by guarded alternatives

We can use guards to select special cases in functions. This function is `True` when the year number is a leap year:

```
leapyear :: Int -> Bool
leapyear y | mod y 400 == 0  =  True   -- 2000 was
           | mod y 100 == 0  =  False  -- 1900 wasn't
           | mod y 4 == 0    =  True   -- 2016 was
           | otherwise       =  False  -- 2018 wasn't
                                       -- 2019 won't be
```

Cases by guarded alternatives

Guards and patterns can be combined:

```
startswith _ [] = False
startswith c (x:xs) | x == c    = True
                    | otherwise = False
```

First the patterns are matched; when an equation is found the guards are evaluated in order in the usual way.
If no guard matches then we return to the pattern matching stage and try to find another equation.

Factorial! as Patterns (again!)

Math:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)!, \qquad n > 0 \end{aligned}$$

Haskell:

```
factorial 0          =  1
factorial n | n > 0  =  n * factorial (n-1)
```