FUNC Practical after Lectures 4–6

♠ `pascal` **(Grade II)** The infinite-order *Pascal's triangle* has `1` as the first and last value in each row. Each other value is the sum of the values immediately above-left and above-right in the triangular layout.
Define `pascal :: Tri Integer`. For example:

```
ghci> Tri (take 4 (rows pascal))
   1
  1 1
 1 2 1
1 3 3 1
```

♠ `hamming` **(Grade II)** The ordered infinite sequence of *Hamming numbers* contains all integers greater than one that are products of twos, threes and fives. Define `hamming :: [Integer]`. Expect `hamming` ⇝ `[2,3,4,5,6,8,9,10,12,15,...`
Hints: every Hamming number is the result of doubling, trebling or quintupling either one or a smaller Hamming number.

♠ `primes` **(Grade III)** The ordered infinite sequence of *prime numbers* contains all integers greater than one that are not divisible by any smaller prime number. Define `primes :: [Integer]`. Expect `primes` ⇝ `[2,3,5,7,11,13,17,...`
Hint: if $n > 1$ is not prime, it has a prime factor p with $p^2 \leq n$.

♠ `queens` **(Grade II)** The file `queens.hs` declares the `queens` function discussed in Lecture 4. It is an *inefficient generate-and-test* program. Make it faster by transferring conditions from the test into the generator.
Hint: for example, generalise the fixed list `[1..8]` from which candidate ranks for a new queen are drawn.

♠ `edit` **(Grade II)** The file `Edit.hs` contains the mini-editor program. Add a command `u` to undo the effect of previous commands, like this:

```
ed> i 0 Jack be nimble; Jack be quick;
1 Jack be nimble; Jack be quick;
ed> i 1 Jack fell down and broke his crown,
1 Jack be nimble; Jack be quick;
2 Jack fell down and broke his crown,
ed> u
1 Jack be nimble; Jack be quick;
ed> u
ed>
```

Note: a repeated `u` command travels further back in the editing history; it does not undo the undoing.

About Ha! Here is a grammar for the miniature functional language *Ha!*:

```
prog  -->  eqn+
eqn   -->  name pat* "=" exp
exp   -->  app (":" exp)?
app   -->  name arg*
arg   -->  "[]" | name | "(" exp ")"
pat   -->  "[]" | name | "(" name ":" name ")"
name  -->  alpha+
```

Spaces and line-breaks are permitted between any instances of non-terminals but *all and only names starting a new line also start an equation.*

Here are declarations of algebraic datatypes for the abstract syntax of *Ha!*:

```
data Prog  =  Prog [Eqn]

data Eqn   =  Eqn Name [Pat] Exp

data Pat   =  PNil | PVar Name | PCons Name Name

data Exp   =  Nil | Var Name | App Name [Exp] | Cons Exp Exp

type Name  =  String
```

Note that the triple-layered expressions in the grammar are represented by the single type `Exp`.

Advice: In the following exercises, even though *Ha!* is small, start with a subset.

♠ **parser for *Ha!* (Grade II)** The file `Parse.hs` contains the parsing library module developed in Lecture 5. Using this library, write a parser for *Ha!* of type `Parser Prog`.

♠ **pretty-printer for *Ha!* (Grade II)** The file `Pretty.hs` contains the pretty-printing library module developed in Lecture 6. Using this library, define `prettyProg :: Int -> Prog -> String`.

♠ `PrettyHa` **(Grade II)** Use your solutions to the two previous exercises to write a program `PrettyHa`. Its input should be a *Ha!* program, and its output a nicely formatted version of the input. A command-line argument should specify available width. Check `PrettyHa` works correctly given its own output as input.

♠ **PrettyHa extensions: comments and errors (Grade III)** *Comments* can be tricky for parsers and pretty-printers. Define extensions of *Ha!* and its abstract syntax to allow comments, and extend `PrettyHa` to handle them. *Error-handling* is another important issue. Adapt `ParseWith` to give better than `Nothing` in response to ungrammatical input.