1. http://zvon.org/other/haskell/Outputprelude/FF_o.html

   (&&) performs the *and* operation. Given two boolean values, it evaluates to True if both the first and the second are True, and to False otherwise.

2. .

```
Prelude> let b=[[3,4], [5,6]]
Prelude> b
[[3,4],[5,6]]
Prelude> b ++ [[1,2]]
[[3,4],[5,6],[1,2]]
Prelude>
```

3. .

A triple (x,y,z) of positive integers is called pythagorean if $x^2 + y^2 = z^2$. Using a list comprehension, define a function

```
pyths :: Int → [(Int,Int,Int)]
```

that maps an integer n to all such triples with components in [1..n]. For example:

```
> pyths 5
[(3,4,5),(4,3,5)]
```

```haskell
-- |List all pythagorean triples that include a given length (either as a leg
-- or hypotenuse).
pythSide :: Integral a => a -> [(a, a, a)]
pythSide s = sort $ pythLeg s ++ pythHyp s

-- |List all pythagorean triples that include a given leg length.
pythLeg :: Integral a => a -> [(a, a, a)]
pythLeg leg = sort [ (k * a, k * b, k * c)
                   | k <- divisors leg
                   , (a, b, c) <- primPythLeg $ leg `quot` k
                   ]
```

4. zipWith :: (a → b → c) → [a] → [b] → [c]
   -- tail xs removes the first element from the list xs and returns the remainder.
   -- Thus, tail [1, 2, 3] equals [2, 3]. tail errors if the list is empty.

zipWith takes two lists and applies a function to each pair of elements, generating a list that is the same length as the shorter of the two

5. no
6.

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

The first pattern specifies that if we try to take a 0 or negative number of elements, we get an empty list. Notice that we're using `_` to match the list because we don't really care what it is in this case. Also notice that we use a guard, but without an `otherwise` part. That means that if `n` turns out to be more than 0, the matching will fall through to the next pattern. The second pattern indicates that if we try to take anything from an empty list, we get an empty list. The third pattern breaks the list into a head and a tail. And then we state that taking `n` elements from a list equals a list that has `x` as the head and then a list that takes `n-1` elements from the tail as a tail. Try using a piece of paper to write down how the evaluation would look like if we try to take, say, 3 from `[4,3,2,1]`.

- The underscore symbol _ is a wildcard pattern that matches any argument value.

7. .

```
-- The scalar product of two lists of integers xs and ys of length n is given by
-- the sum of the products of corresponding integers:

-- In a similar manner to the function chisqr, show how a list comprehension can
-- be used to define a function scalarproduct :: [Int] → [Int] → Int that returns
-- the scalar product of two lists. For example:

-- > scalarproduct [1, 2, 3] [4, 5, 6]
-- 32

scalarproduct       :: [Int] -> [Int] -> Int
scalarproduct xs ys = sum [k * v | (k, v) <- zip xs ys]
```

8. .

```
pairEq :: (a -> a -> Bool) -> (b -> b -> Bool) -> (a,b) -> (a,b) -> Bool
pairEq (==.) (.==) (x1,y1) (x2,y2) = x1 ==. x2 && y1 .== y2

pairOrd :: (a -> a -> Bool) -> (b -> b -> Bool) -> (a,b) -> (a,b) -> Bool
pairOrd (<=.) (.<=) (x1,y1) (x2,y2) = x1 <. x2
                                    || x1 ==. x2 && y1 .<= y2
  where
  x <.  y = x <=. y && not (y <=. x)
  x ==. y = x <=. y &&      y <=. x
```

9. .

The function *halve* splits a list into two equal halves:

$$halve\ xs = (take\ m\ xs, drop\ m\ xs) \quad \textbf{where}\ m = length\ xs\ \text{div}\ 2$$

This definition involves three traversals of the list, one to compute its length, and two more to perform the splitting. No human would split up a list this way. Instead, they would simply deal out the elements alternately into two piles:

$$halve = foldr\ op\ ([],[]) \quad \textbf{where}\ op\ x\ (ys,zs) = (zs, x:ys)$$

This version of *halve* produces a different result, but the two sublists have the same sizes as before and each is a subsequence of the input. For example,

$$halve\ [1..9] = ([2,4,6,8],[1,3,5,7,9])$$

Since the order of the elements in the tree is immaterial, this definition of *halve* is perfectly adequate.

10. .

```
-- Joining two lists

(++) :: [a] -> [a] -> [a]

[]     ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)
```

11..

```
-- 7.1 Pattern matching revisted

mystery,mystery' :: Integer -> Integer -> Integer
mystery x y
  | x==0      = y
  | otherwise = x

mystery' 0 y = y
mystery' x _ = x
```

12..

```
null' :: [a] -> Bool
null' (_:_) = False
null' []    = True
```

still works correctly, because `(_:_)` fails to match the particular case of an empty list.

- https://stackoverflow.com/questions/27941690/implementation-of-null-function
- And (_:_) specifies that the list must have at least one element or the match will fail