Solution 1.1

- A function `is_even` such that `is_even n` will be true when `n` is divisible by 2, and false otherwise.
- A number is even if its value modulo 2 is zero (`n \mod 2 = 0`)
- Guarded Solution

\[
\text{is\_even } n | n \mod 2 == 0 \Rightarrow \text{True} \\
| \text{otherwise} \Rightarrow \text{False}
\]

- Direct solution

\[
is\_even n = (n \mod 2) == 0
\]

Solution 1.2

- A function `unixname` which removes all vowels from a string (so `unixname "the house"` would be "th hs").
- How about a predicate checking for “vowel-hood” first?

\[
is\text{vowel } 'a' = \text{True} \\
is\text{vowel } 'e' = \text{True} \\
is\text{vowel } 'i' = \text{True} \\
is\text{vowel } 'o' = \text{True} \\
is\text{vowel } 'u' = \text{True} \\
is\text{vowel } _ = \text{False}
\]

- The empty string stays empty

\[
\text{unixname } [] = []
\]

- For the non-empty case, discard a vowel

\[
\text{unixname } (c:cs) | \text{is\_vowel } c = \text{unixname } cs \\
| \text{otherwise} = c:(\text{unixname } cs)
\]
Solution I.3

- A function `censor` which will replace all vowels in a string with the letter `x`
  (so, `censor "the house"` would be `thx hxxsx`).
- The empty string stays empty
  
      censor [] = []

- For the non-empty case, change a vowel into an 'x'
  
      censor (c:cs) | isvowel c  = 'x':(censor cs)
                 | otherwise = c:(censor cs)

Solution I.4

- A function `copies` such that `copies 3 "hello"` will compute `"hello", "hello", "hello"`
- Zero copies produces an empty list
  
      copies 0 x = []

- `n` copies is one copy followed by `n-1`:
  
      copies n x = x:(copies (n-1) x)

Solution II

- Implement the Calculator based on a Expression syntax as described at the end of lecture 3.
- Possible transcript:

```
_GAIN> let x = Val 3
_GAIN> let y = Val 4
_GAIN> eval x
3.0
_GAIN> eval y
4.0
_GAIN> let xandy = Add x y
_GAIN> xandy
Add (Val 3.0) (Val 4.0)
_GAIN> eval xandy
7.0
_GAIN> Divide xandy x
Divide (Add (Val 3.0) (Val 4.0)) (Val 3.0)
_GAIN> eval it
2.333333333
```

Error Message

- What it is thinking?
  - It thinks the author meant `(is_even n)-2`
  - Subtraction requires both arguments be numeric (Num)
  - It knows that `is_even n` is Bool
  - It suggests adding type Bool to the Num class
    (this is in fact silly)
- What is actually wrong?
  - Function application binds tighter than infix operators
  - so `is_even n - 2` is parsed as `(is_even n) - 2`
  - What should have been written was `is_even (n-2)`
Higher Order Functions (HOFs) are a key aspect of Haskell

Lab 2 Part I concentrates on using HOFs
Lab 2 Part II looks at writing HOFs

Submission:

- Create file `⟨username⟩-lab02.hs`
- Make sure lab no. and your name is in end-of-line comment at start
  
  ```haskell
  -- Lab2 : Forename Surname
  ```
- Attach source to email message and send
  
  To: Andrew.Butterfield@cs.tcd.ie
  Subject: CS4011 Lab2

Without using recursion or list comprehensions (i.e. using `map` and `foldl`), and given the following Prelude functions:

- `const x _ = x`
- `f . g = x -> f (g x)`
- `flip f x y = f y x`
- `f $ x = f x`

Code up the following functions:

1. `sumsq` — computes sum of squares of a numeric list
   
   ```haskell
   sumsq [1,2,3] = 1 + 4 + 9 = 14
   ```

2. `len` — computes length of a list
   
   ```haskell
   len [1,2,3] = 3
   ```

3. `rev` — reverses its list argument
   
   ```haskell
   rev [5,6,7,8] = [8,7,6,5]
   ```

4. `cat` — concatenates two lists together
   
   ```haskell
   cat [1,2,3] [4,5,6] = [1,2,3,4,5,6]
   ```

Lab 2, Part II

Given a simple binary tree datatype:

```haskell
data Tree a = Nil | Node (Tree a) a (Tree a)
```

Use recursion to:

5. Write a function `tmap` taking a function `f` and tree `t` as arguments,
   that returns the result of applying `f` to every value in the tree `t`
   
   ```haskell
   tmap (+1) (Node (Node Nil 1 Nil) 2 (Node Nil 3 Nil))
   = (Node (Node Nil 2 Nil) 3 (Node Nil 4 Nil))
   ```

6. Write a function `tfold` taking a function `f`, value `v` and tree `t` as arguments,
   that returns the result of:
   - using `v` when dealing with an empty tree
   - using `f` to combine the results of left and right trees with the
     (middle) node value to get an overall result
   
   ```haskell
   myfun left middle right = left + right + 2 * middle
   tfold myfun 0 (Node (Node Nil 1 Nil) 2 (Node Nil 3 Nil))
   = 12
   ```

Mapping and Folding over arbitrary datatypes

- Mapping replaces “leaf-values” by result of applying function argument
  - overall data-structure shape is unchanged
- Folding replaces base-case values by supplied value
- Folding replaces recursive data-constructors by function argument.
  - data structure is reduced down to a single value.