1. For each of the following Haskell expressions, either give the proper Haskell notation for the expression's type, or explain why the expression does not have a type.
   (a) `['5', 'a']`
   (b) `[5, 'a']`
   (c) `('5', 'a')`
   (d) `(5::Int, 'a')`
   (e) `[('a', 'b')]`
   (f) `(['a'], 'b')`
   (g) `[['a'], 'b']`

2. (a) Explain the difference between a function that has this Haskell type

```
[a] -> [b] -> Int
```

   and a function that has this type.

```
[a] -> [a] -> Int
```

   (b) What are the types of each of the following two Haskell functions? Explain why.

```
f1 x y = length x + length y
f2 x y = length (x ++ y)
```

3. This list comprehension,

```
[ (x,3*x) | x <- [2, 4..18] ]
```

   produces this list.

```
[(2,6),(4,12),(6,18),(8,24),(10,30),(12,36),(14,42),(16,48),(18,54)]
```

   Find another way to write this list as a list comprehension.

4. Define each of the following lists using a list comprehension.
   (a) `[-18, -9, 0, 9, 18, 27, ...]`
   (b) `[1, 10, 100, 1000, 10000, 100000, 1000000, 10000000]`

5. Simplify each arithmetic expression so that it uses only the infix version of each operator.
   (a) `(+) x ((*) y (r - (*) s t)) where x=4; y=5; r=2; s=3; t=6`
   (b) `(+2) ((*) ((*3) x) (y - ((+1) z))) where x=4; y=5; z=6`

6. Suppose that `f` and `g` have the following Haskell types.

```
f :: Int -> Int
g :: Int -> Int -> Int
```

   Which of the following expressions make sense? Explain why.

   (a) `f g 1 2`

   (b) `f (g 3 4)`

   (c) `f (g 5) 6`

   (d) `g (f 7) 8`

   (e) `g f 9 8`

   (f) `g (f 7 6)`

   (g) `g 5 (f 4)`

   (h) `(g 3) (f 2)`

7. Write a function which returns the head and the tail of a list as the first and second elements of a tuple. What would be this function's type?

8. For each list, write a list comprehension that produces the list.

   (a) `[[0,0], [0,0,0,0], [0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0,0,0,0]]`

   (b) `[(1, [1]), (2, [1,2]), (3, [1,2,3]), (4, [1,2,3,4]), (5, [1,2,3,4,5]), (6, [1,2,3,4,5,6])]`

   (c) `[[1], [1,2,1], [1,2,3,2,1], [1,2,3,4,3,2,1], [1,2,3,4,5,4,3,2,1], [1,2,3,4,5,6,5,4,3,2,1]]`

9. For each function, use pattern matching to rewrite the function in a clearer way.

   (a) `g1 x = 1 + fst x`

   (b) `g2 x = 1 + head x`

   (c) `g3 x = (snd x, fst x)`

   (d) `g4 x = (head x, tail x)`

   (e) `g5 x = (head (tail x)) :  (head x) :  (tail (tail x))`

   (f) `g6 x = (head x) + (head (tail x)) :  (tail (tail x))`

10. Write a recursive function that counts the number of times its first argument appears in its list argument.

```
count :: Eq a => a -> [a] -> Int
```

11. Suppose I want to write a function

```
repeats :: Eq a => [a] -> Bool
```

that returns true if and only if the input list has two equal elements next to each other.

(a) What is wrong with this implementation of `repeats`? (Hint: `[1,2,2,3]`)

```
repeats [ ] = False
repeats [x] = False
repeats (x:y:xs) | (x == y) = True
                 | otherwise = repeats xs
```

(b) How do you fix the above definition for `repeats`?

12. (a) Explain what this function does to a list. Note: Do not just say in words what the code says ("This function returns the empty list when given the empty list and it returns an empty list when given a one item list and it ...") *Explain* what the function does to a list at a fairly high level.

```
mystery1 [ ] = [ ]
mystery1 [x] = [ ]
mystery1 (x:y:xs) = y : (mystery1 xs)
```

(b) Explain how this function is different from the previous one.

```
mystery2 [ ] = [ ]
mystery2 [x] = [x]
mystery2 (x:y:xs) = x : (mystery2 xs)
```

(c) If I define `mystery2` the following way, what bug will that introduced into the function? Explain. (Hint: This function will work correctly for some lists but not for all lists.)

```
mystery2 [ ] = [ ]
mystery2 (x:y:xs) = x : (mystery2 xs)
```

13. What does this expression evaluate to. Briefly explain why.

```
map (\(x,y)->x) [(1,2),(3,4),(5,6),(7,8)]
```

14. What does each of the following expresions evaluate to? Briefly explain how each expression is evaluated. In what way are they similar? In what way do they differ?

```
filter even [1,2,3,4,5,6,7]
   map even [1,2,3,4,5,6,7]
```

15. Use `map` to write a one line function

```
pairList :: [a] -> [(a,a)]
```

that puts each element from the input list into a pair that duplicates that element. Here is an example.

```
pairList [1,2,3,4]  ==> [(1,1),(2,2),(3,3),(4,4)]
```

16. What does the following expression evaluate to? Also, explain how Haskell evaluates this expression. (What is being plugged in where?)

```
(\x -> (\y -> x y)) (\w -> 3*w) 4
```

17. Suppose I define this function

```
lin m b x = m*x + b
```

(a) Suppose I now define the function

```
f = lin 2 3
```

What is the type of function `f`? What does `f` do?

(b) Suppose I now define the functions

```
g = lin 1
h1 = g 3
h2 = g 5
```

What is the type of function `g`? What do `h1` and `h2` do?

18. In each of the following Haskell expressions, determine which letter is a function and wich letter is a parameter to a function.

(a) `a b c d`

(b) `(((a b) c) d)`

(c) `(a (b (c d)))`

(d) `a (b c d)`

(e) `a b (c d)`

(f) `a (b c) d`

(g) `(a b) c d`

(h) `(a b) (c d)`

19. Let

```
f x y z = 2*x + 3*y + 4*z
```

Describe each of the functions `g1` through `g14` by giving a lambda expression that each function is equivalent to.

(a) `g1 = f 5`

(b) `g2 u v = f 5 u v`

(c) `g3 u v = f 5 v u`

(d) `g4 u v = f u 5 v`

(e) `g5 u v = f u v 5`

(f) `g6 = f 5 4`

(g) `g7 u = f 5 4 u`

(h) `g8 u = f u 5 4`

(i) `g9 u = f 5 u 4`

(j) `g10 = g5 5`

(k) `g11 = g3 5`

(l) `g12 = g2 5`

(m) `g13 = g4 5`

(n) `g14 u v = g2 v u`

20. Write a one line function

```
positives :: [Integer] -> [Integer]
```

that copies each number from the input list to the output list unless the number is negative in which case the input number is replaced by zero in the output list.

21. Write a one line function

```
incList :: Num a => a -> [a] -> [a]
```

that increments each element of the input list by the numeric parameter.

22. Write a one line function

```
sumMaxs :: [(Integer,Integer)] -> Integer
```

which consumes a list of pairs of integers and returns the sum of the maximum elements from each pair. Here is an example.

```
sumMaxs [(1,2),(-1,1),(3,0),(4,4),(5,4)]  ==> 15
```

23. Write a one line function

```
member :: Eq a => a -> [a] -> Bool
```

that returns true if and only if the first parameter is an element of the list parameter.

24. (a) Write a recursive function

    ```
    indexPairs :: [a] -> [(Integer,a)]
    ```

    that consumes a list and produces a list of pairs where every element from the input list is paired up with its index from the input list. Here are two examples.

    ```
    indexPairs "hello"  ==>  [(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
    indexPairs (take 4 (repeat 5))  ==>  [(0,5),(1,5),(2,5),(3,5)]
    ```

    (b) Notice that this function consumes a list and produces another list the same length as the input list. Quite often this is a sign that the function can be written using the `map` function. Explain why `indexPairs` is NOT a good candidate for being implemented using `map`.

    (c) Implement `indexPairs` using `foldr` or `foldl`.

    (d) Implement `indexPairs` using a list comprehension.

25. Write a one line definition for

```
zip3' :: [a] -> [b] -> [c] -> [(a,b,c)]
```

using `zip`, `map`, and a lambda expression.

26. Each of the following functions consumes a list of numbers. Give a verbal description of what each function does. (Don't just read the definition of each function from left to right. Give an explanation of what the function does to an input list.)

    (a) `f1 = (map (\x -> x*x)) . (map (max 0))`

    (b) `f2 = map ((\x -> x*x) . (max 0))`

    (c) `f3 = map ((\x -> x*x) . (min 0))`

    (d) `f4 = (map (\x -> x*x)) . (filter (0<))`

    (e) `f5 = (map (max (-10)) . (map (min 10))`

    (f) `f6 = map ((min 10) . (max (-10)))`

    (g) `f7 = ((filter ((-10)<)) . (filter (<10)))`