Question 1.1:

1. (a plus (f x)) + ((x times y) * z)
2. ((3 + 4) + 5) + 6
3. 2 ^ (2 ^ (2 ^ 2))

Question 1.2:

The claim is that (f . g) . h = f . (g . h)
LHS:
((f . g) . h) x
= {-> direction of the equation for functional composition}
(f . g)(h x)
= {-> direction of the equation}
f(g(h x))

RHS:
(f . (g . h)) x
= {-> direction of the equation}
f ((g . h) x)
= {-> direction of the equation}
f(g(h x))

Question 1.3:

The (++) operator is associative because it does not matter which concatenation is done first, only the order of the elements matter.
The operator is not commutative because the result of (a ++ b) is first the element a and then the element b, but the result from (b ++ a) is first the element b, then the element a.
The identity element is the empty set as (a ++ []) = a = ([] ++ a).
The zero element is also the empty set.

Question 1.4:

map double [3, 7, 4, 2] = [6, 14, 8, 4]
map (double . double) [3, 7, 4, 2] = [12, 28, 16, 8]
map double [] = []

The first equation is true because it does not matter if each of the elements in the set is doubled and then the sum is taken, or first if the sum is taken and then doubled.
For the second equation the function "concat" just "flattens" a set of sets and then it takes the sum of all the individual elements. The left-hand side "map sum" returns a set in which the elements are the sum of every set and then the sum of those sum is taken, so it is true.
In the third equation it does not matter if a set is ordered or not, the sum is still the same, so it is true.

Question 2.1:

```
factorial :: Int -> Int
factorial n = if n >= 0 then product s else error “The number has to be non-negative”
  where s = [1..n]

choose :: Int -> Int -> Int
choose n m = factorial n `div` (factorial m * factorial (n - m))

check :: Int -> Bool
check n = if sum [choose n r | r <- [1..n]] == 2^n then True else False
```

Question 2.2:

```
not :: Bool -> Bool
not x
 | True = False
 | False = True

not :: Bool -> Bool
not x = x < True

not :: Bool -> Bool
not x = if x == True then True else False

not :: Bool -> Bool
not True = False
not False = True
```

Question 2.3:

When a function of type A -> B is given, the number of such functions are cardinality of B to the power of cardinality of A. The reason for that is that for every A there are B options the function can return.

1. Bool -> Bool

2 ^ 2 = 4

2. Bool -> Bool -> Bool is the same as Bool -> (Bool -> Bool)

For the expression in the brackets there are 4 things. So the total number of functions that has this type is 4 ^ 2 = 16.

3. Bool -> Bool -> Bool -> Bool is the same as Bool -> (Bool -> (Bool -> Bool)) because “->” is right associative.

From the previous question it can be seen that for the expression (Bool -> (Bool -> Bool)) there are 16 possibilities. So the total number of functions that have this type is 16 ^ 2 = 256.

4. For (Bool, Bool) there are 2 options for the first Bool and 2 options for the second Bool => 4

5. (Bool, Bool) -> Bool there are 2 options for Bool and 4 options for (Bool, Bool) => 2 ^ 4 = 16.

6. (Bool, Bool, Bool) there are 2.2.2 = 8 things

7. (Bool, Bool, Bool) -> Bool there are 2 ^ 8 = 256 things

8. (Bool -> Bool) -> Bool there are 2 ^ 4 =16 things

9. (Bool -> Bool -> Bool) -> Bool there are 2 ^ 16 = 65 536

10. ((Bool -> Bool) -> Bool) -> Bool
For the expression ((Bool -> Bool) -> Bool) there are 2 ^ 4 =16 things, so for the whole expression there are 2 ^ 16 = 65 536 things.