1.1

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

1.2 Proving that function composition is associative follows as a very simple application of the definition:
Let f :: c -> d, g :: b -> c, h :: a -> b, x :: a. One can write:
((f . g) . h) x = (f . g) (h x) = f (g (h x)) = f ((g . h) x) = (f . (g . h)) x

---

1.3 Let as ++ bs = concat [as, bs], then:
++ IS associative since for all as, bs, cs :: [a] we have that (as ++ bs) ++ cs == concat [as, bs] ++ cs == concat [concat [as, bs], cs] == concat [as, bs, cs] == concat [as, concat [bs, cs]] == as ++ concat [bs, cs] == as ++ (bs ++ cs).

++ is NOT commutative since [1] ++ [2] == [1, 2] whereas [2] ++ [1] == [2, 1].

++ HAS GOT the trivial unit / identity element e = []. One can easily see that x ++ e == concat [x, e] == concat [x, []] == x == concat [[], x] == concat [e, x] == e ++ x.

++ DOESN'T have a zero element. To see this start from the equation z ++ x == z and substitute in x = 2, thus z ++ x == concat [z, 2] == z. But the lenght of a list can not decrease under concatenation, so there is no such z.

(Note: as a result, (The set of all lists of a given type, ++) is a monoid)

1.4
```
double :: Integer -> Integer
double x = 2 * x
```

Then:
```
map double [3, 7, 4, 2] == [6, 14, 8, 4]

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

map double [] = []
```

Also let sum :: [Integer] -> Integer, then:
sum . map double == double . sum, by distributivity of multiplication over addition (adding then summing up versus summing up and then doubling)

sum . map sum == sum . concat, by associativity of addition (adding up the elements of each list and then summing over the results versus concatenating all lists and then summing up all the numbers from the resulting list)

sum . sort == sum, by commutativity of addition (sorting the numbers first doesn't change the outcome of adding up all the numbers)

2.1 Here are, actually, 5 ways to do it:

```
not, not', not'', not''', not'''' :: Bool -> Bool

not False = True
not True  = False

not' x | x == False = True
       | otherwise  = False

not'' x = invertedBools !! fromEnum x
          where invertedBools = [True, False]

not''' x = if x == False then True else False

not'''' x = case x of False -> True
                      True  -> False
```

We can check that these are indeed correct implementations of not by running the following test and seeing that it outputs True, meaning that our implemeareions were all correct:

```
filter (\x -> x) [f x /= Prelude.not x | f <- [Main.not, not', not'', not''', not''''], x <- [False, True]] == []
```

2.2
Let nr(f) denote the number of different haskell functions of type f. Then, nr satisfies the following:

i. nr(Bool) = 2, obvious from definition
ii. nr((f, g)) = nr(f) * nr(g), clear from cardinality of cartesian products (Note: this can be further extended for arbitrarily sized tuples).
iii. nr(f -> g) = nr(g) ^ nr(f), clear from the number of functions one can mathematically define with domain the set of all f and codomain the set of all g

These being said:
1. nr(Bool -> Bool) = nr(Bool) ^ nr(Bool) = 2 ^ 2 = 4
2. nr(Bool -> Bool -> Bool) = nr(Bool -> Bool) ^ nr(Bool) = 4 ^ 2 = 16
3. nr(Bool -> Bool -> Bool -> Bool) = nr(Bool -> Bool -> Bool) ^ nr(Bool) = 16 ^ 2 = 256
4. nr((Bool, Bool)) = nr(Bool) ^ 2 = 2 ^ 2 = 4
5. nr((Bool, Bool) -> Bool) = nr(Bool) ^ nr((Bool, Bool)) = 2 ^ 4 = 16

6. nr((Bool, Bool, Bool)) = nr(Bool) ^ 3 = 2 ^ 3 = 8
7. nr((Bool, Bool, Bool) -> Bool) = nr(Bool) ^ nr((Bool, Bool, Bool)) = 2 ^ 8 = 256
8. nr((Bool -> Bool) -> Bool) = nr(Bool) ^ nr(Bool -> Bool) = 2 ^ 4 = 16
9. nr((Bool -> Bool -> Bool) -> Bool) = nr(Bool) ^ nr(Bool -> Bool -> Bool) = 2 ^ 16 = 65536
10. nr(((Bool -> Bool) -> Bool) -> Bool) = nr(Bool) ^ nr((Bool -> Bool) -> Bool) = 2 ^ 16 = 65536

Now, for listing all functions of type (Bool -> Bool) -> Bool I am going to use this additional construct:

```
f nr x = nr .&. bit xIndex /= 0 where xIndex = fromEnum (x False) + fromEnum (x True) * 2
-- encodes function x as a binary number ranging from 0 to 3 and nr as a binary number ranging from 0 to 15 and checks for the x-th bit in nr
```

Now, the requested functions are f 0, f 1, f 2, ..., f 15.