Problem 1

f x y z = x : ([y] : [z]) What is the type of *f*?
***f* has a type of *[a] − > a − >[a] − > [[a]]*. So, the function takes in a list, an element, and another list, which are all of the same type. The function then returns a list of a list of the same type.**

Problem 2

(a) The test *False* < *True* is allowed because < is provided by a typeclass that *Bool* is an instance of. What is the typeclass and what is the type (<)(including the typeclass)?
< has a type of *Ord a* => *a − > a − > Bool*. So, (<) takes any two elements of the same type and returns a boolean. The type of those two elements must be of the Ord typeclass. Ord are for types that have an ordering and it also must be of the Eq typeclass(*:info* (<) outputs *class Eq a* => *Ord a where ...* (<) *:: a − > a − > Bool*).

(b) What are the functions that give the *ASCII* code for a character and give the *ASCII* character for an integer(if you use a type notation *:: Char*)? (i.e., *fnc1 'a'* yields 97, *fnc2 97:: Char* yields *'a'*.) Also, what are their types(including the typeclass)?
***fromEnum, toEnum* are functions that give ASCII code for a character and give the ASCII character for an integer respectively. *fromEnum* has a type of *Enum a* => *a − > Int*. It takes in an element of a type from Enum typeclass and returns an Int. *toEnum* has a type of *Enum a* => *Int − > a*. It takes in an Int and returns an element of a type from the Enum typeclass.**

(c) The functions in part (b) are provided by a typeclass that *Char* is an instance of. What is the typeclass?
The typeclass is Enum.

Problem 3

The function *twice list* should return true iff some values occurs twice in the list. E.g.,

filter twice [[],[1],[1,2],[2,2],[1,2,3],[1,2,1],[1,1,2],[1,2,2]]
[[2,2],[1,2,1],[1,1,2],[1,2,2]]

(a) What is the type of *twice*?
***twice* has a type of *Eq a* => *[[a]] − > Bool*. It takes in a list of lists and returns a boolean whether or not some values occur twice. The typeclass is Eq.**

(b) Briefly describe the syntactic and semantic bugs in the program below.
:{

twice [] = False
twice [_] = False
twice [x,x] = False
twice (_ ++ [x] ++ _ ++ [y] ++ _) = x == y
twice (h1 : h2 : t) == (h1 == h2 || twice h1 t)
:}
***twice [x,x] = False*, results in an error because *x* was used twice which resulted in conflicting definitions for *x*, because this defines *x* twice.**
***twice (_ ++ [x] ++ _ ++ [y] ++ _) = x == y*, results in an error because it uses ++ and the arguments are _ and a list. It should instead use *:* and remove the brackets.**
***twice (h1 : h2 : t) == (h1 == h2 || twice h1 t)* results in an error because of the == and *twice h1 t* (wrong type). To make it compile, it should instead be *twice (h1 : h2 : t) = (h1 == h2 || twice (h1 : t))*.**

(c) Rewrite *twice* to make it work. Keep using definition by cases; feel free to add/change/delete cases as you see fit.
:{
twice [] = False
twice [_] = False
twice [x,y] = x == y
twice (h1 : t) = (h1 ‘elem‘ t || twice t)
:}

(d) Write a definition by cases for *twice* that only has two cases(one recursive, one not).
:{
twice [] = False
twice (h1 : t) = (h1 ‘elem‘ t || twice t)
:}

(e) Rewrite your definition from part (c) using cases and guards; break up the 3-clause logical or test to use a sequence of guards. (Don’t leave any || in the definition)

twice x pattern
$| \; guard1 = result1$
$| \; guard2 = result2$
$(omitted)$

:{
twice x | length x <= 1 = False ; otherwise = twice (h1 : t) = (h1 ‘elem‘ t || twice t)
:}

(f) Rewrite your definition from part (c) to be of the form twice x = case x of .... You can add guards to a case clause using the syntax

$$\begin{aligned} &case\ expr\ of\ pattern \mid guard1 -> \ result1 \\ &\qquad\qquad\qquad\qquad\ \mid guard2 -> \ result2 \\ &\qquad\qquad\qquad\qquad\ (omitted) \end{aligned}$$

:{
twice x = case length x of
0 − > False
1 − > False
_ − > let x' = x in twice (x' : xs) = (x' 'elem' xs || twice xs)
:}

Problem 4

Consider the following claim: "A Haskell function is higher order if and only if its type has more than one arrow." Is this correct? Give a brief argument.
This is true because by definition, a higher order function in haskell either takes another function as a parameter or returns a another function as a result. To do this, the type must have more than one arrow.
For example:
f_squared f x = f (f x) **has type of** ***(t − > t) − > t − > t*** **which means that it takes in a function as an argument and returns a result. This particular function just applies the function twice. As you can see, it has more than one arrow.**

Problem 5

Let f :: $(a->a->a)->a->a->a$

(a) Rewrite $f * (2\ 3)$ so that it has no syntax errors and yield 6 if $f\ h\ x\ y = h\ x\ y$
f (*) 2 3 **which yields 6.**

(b) Write the definition of a function g :: $((a,a)->a,(a,a))->a$ so that g is an uncurried version of f. Calling your function on ∗, 2, and 3 should yield 6.
g (h, (x,y)) = h (x,y) **has a type of** ***((a,a) − > a, (a,a)) − > a*** **but you have to call the function with** ***uncurry (*), (2,3)*** **such that** ***g(uncurry (*), (2,3))*** **yields 6.**
If you want to just call the function with ****,2,3***, **you can do**
g (h, (x,y)) = h x y **which yields 6 if you call it with just** ***(*), (2,3)***, **however it no longer is the same type as specified in the problem description.**

Problem 6

Let *f1 = filter (\x − > x >0)* and *f2 = filter (\x − > x < 0)*, and let *nbrFilter g x = length (filter g x).*

(a) Rewrite *f1 (f2 [-5..15])* so that it uses function composition to apply just one function to the list.
(f1 . f2) [-5..15]. **The infix dot combines the two functions together to one function. So, it only applies one function to the list rather than two.**

(b) Rewrite the *nbrFilter* function definition to have the form:
nbrFilter g = function composition involving *length* and *filter* ... and leaving out *x*.
nbrFilter g = length.(filter g)

Problem 7

(a) Rewrite *f g x y = g x (y x)* three ways, first *f g x* = unnamed lambda function, then *f g* = unnamed lambda function, and finally *f* = unnamed lambda function.
f g x = \y − > g x (y x)
f g = \x y − > g x (y x)
f = \g x y − > g x (y x)

(b) Briefly, how does *var = lambda function* relate to first-class function in Haskell?
That is one way Haskell supports first-class functions. First-class functions means functions are treated like any variable. You can use functions as expressions and can have variables whose values are functions. For example, *var = lambda function*

Problem 8

Let's re-implement the *foldl* function in multiple ways. Your *foldl* only needs to work on lists.

(a) Write a definition for *foldl* using conditional expressions:
foldl1a f a x = if x == [] then etc.
:{
foldl1a f a x = let len = length x in
if len == 0 then a
else if len == 1 then (f) a (head x)
else foldl1a f a (h:t) = let temp = (f) a h in foldl1a f temp t
:}

(b) Rewrite the definition using function definition by cases: *foldl2...*

:{
foldl2 f a [] = a
foldl2 f a (h:t) = let temp = (f) a h in foldl2 f temp t
:}

(c) Rewrite the definition using a case expression: *foldl3 f a x = case x ...*

foldl3 f a x = case length x of
0 −> a
1 −> (f) a (head x)
_ −> let x' = init(reverse x)
let temp = (f) a (head x)
in foldl3 f temp x'