Quiz (Week 1)

Typing

Assuming for the sake of simplicity that all numeric literals are of type `Int`, What is the type of the following Haskell expressions?

Question 1

```
"hello world"
```

1. ✗ `string`
2. ✔ `[Char]`
3. ✗ `char*`
4. ✗ `[String]`

In Haskell, strings are actually just lists of characters, and the names of types (like `Char`) are always written with an initial upper-case letter.

Question 2

```
(1, 'x', [True])
```

1. ✗ `List`
2. ✗ `(Int, Char, Bool)`
3. ✗ `Tuple`
4. ✔ `(Int, Char, [Bool])`

In Haskell, a tuple `(x,y)` is typed according to the following rule:

$$\frac{x : \tau_1 \quad y : \tau_2}{(x, y) : (\tau_1, \tau_2)}$$

This can be read as `(x,y)` is of type (τ_1, τ_2) if `x` is of type τ_1 and `y` is of type τ_2.

A similar rule exists for triples like `(1,'x',[True])`, and as `1` is of type `Int`, `'x'` is of type `Char`, and `[True]` is of type `[Bool]`, we have answer 4 as the only correct answer.

Question 3

```
["x":[]]
```

1. ✗ `[[Char]]`
2. ✔ `[[[Char]]]`
3. ✗ `[Char]`
4. ✗ `String`

Keeping in mind that `String` is a synonym for `[Char]`, we have the type of cons (the `(:)` operator) as:

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

If we apply the first argument, `"x"`:

```
("x":) :: [[Char]] -> [[Char]]
```

Then apply the second argument, `[]`, and we have:

```
("x":[]) :: [[Char]]
```

Lastly, this list of list of characters is in turn put in a list, as it is surrounded by square brackets. So the answer is number 2, or a list of lists of lists of characters.

Question 4

```
map (\x -> x + 1)
```

1. ✗ `[a] -> [b]`
2. ✗ `Int -> Int`
3. ✔ `[Int] -> [Int]`
4. ✗ `(Int -> Int) -> [Int] -> [Int]`
5. ✗ Invalid, as not enough arguments are given to `map`.

It's worth noting that *all functions in Haskell accept one argument and return one result*. Multi-argument functions are emulated by writing a function that, given its first argument, returns a *function* that awaits further arguments. This technique is called *currying*.

For example, the function `map` has the following type:

```
map :: (a -> b) -> [a] -> [b]
```

This can be more explicitly expressed with the right-associated parentheses, as follows:

```
map :: (a -> b) -> ([a] -> [b])
```

Given the argument function `(\x -> x + 1)`, a lambda expression of type `Int -> Int`, `map` shall return a function of type `[Int] -> [Int]`, or option 3.

Evaluation

Choose all expressions that are equivalent to the following expressions[1]:

Question 5

```
3 : [40] ++ [50] ++ 5 : [60]
```

1. ✔ `3 : [40] ++ ([50] ++ 5 : [60])`
2. ✗ `3 : ([40] ++ [50] ++ 5) : [60]`
3. ✔ `(3 : [40] ++ [50]) ++ (5 : [60])`
4. ✔ `3 : ([40] ++ [50] ++ (5 : [60]))`
5. ✔ `3 : [40, 50] ++ [5, 60]`

It's important to note that the `(++)` operator is associative, that is:

```
(xs ++ ys) ++ zs == xs ++ (ys ++ zs)
```

This can be proven by induction on `xs`, with the aid of some helper lemmas. Because of this associativity, the placement of parentheses around `++`-terms is not important, which makes options 1,3 and 4 correct. In addition, option 5 is also correct as we know that `[40] ++ [50] = [40,50]` and that `5 : [60] = [5,60]`.

Question 6

```
map ($ 5) [(-),(+),(*)]
```

1. ✗ `map (\f x -> f x 5) [(-),(+),(*)]`
2. ✔ `map (\f x -> f 5 x) [(-),(+),(*)]`
3. ✗ `[(- 5),(+ 5),(* 5)]`
4. ✔ `[(5 -),(5 +),(5 *)]`
5. ✗ The expression is invalid.

The `($)` operator is defined as follows:

```
($) : (a -> b) -> a -> b
f $ x = f x
```

That is, it applies everything on the right as an argument to the function given on the left. It is typically used to eliminate parentheses in Haskell code, but can also be used in a section like this, where `($ 5)` is equivalent to `\f -> f 5` which is equivalent to `\f x -> f 5 x` by η-expansion. Thus option 2 is correct. Taking option 2 and evaluating it further, we will get the list:

```
[\x -> (-) 5 x, \x -> (+) 5 x, \x -> (*) 5 x]
```

Which is equivalent to the operator sections used in answer 4. Answers 1 and 3 are incorrect as they flip the order of arguments used for the function. Answer 3 is even more incorrect as `(- 5)` will be interpreted as a negative number, not an operator section, and thus produce a type error.

Question 7

Note: The functions `ord` and `chr` are from `Data.Char`. They convert `Char` values to/from their ASCII (or unicode) numbers, respectively. For these questions, the answers may have a more general type than the original expression. So long as a given answer has equivalent behaviour *for the type of the original expression*, we consider the answer to be equivalent.

```
let increment x = 1 + x
in \xs -> map chr (map increment (map ord xs))
```

1. ✔ `map chr . map (1+) . map ord`

2. ✔ `map (chr . (1+) . ord)`
3. ✔ `map succ`
4. ✗ `map chr $ map (1+) $ map ord`
5. ✔ `\xs -> map chr . map (1+) $ map ord xs`

The following bit of equational reasoning hits every answer in this question, except 4, which is not type correct.

```
let increment x = 1 + x
in \xs -> map chr (map increment (map ord xs))
= -- Shift argument into lambda
let increment = \x -> 1 + x
in \xs -> map chr (map increment (map ord xs))
= -- Simplify lambda to operator section
let increment = (1 +)
in \xs -> map chr (map increment (map ord xs))
= -- Reduce let expression by substitution
\xs -> map chr (map (1 +) (map ord xs))
= -- Introduce composition, f (g x) = (f . g) x
\xs -> (map chr . map (1 +)) (map ord xs))
= -- Remove parentheses with ($)
\xs -> map chr . map (1 +) $ map ord xs -- Answer 5
= -- Introduce further composition: f $ g x = (f . g) x
\xs -> (map chr . map (1 +) . map ord) xs
= -- η-reduction
map chr . map (1 +) . map ord -- Answer 1
= -- Map (functor) law, map f . map g = map (f . g)
map (chr . (1 +) . ord) -- Answer 2
= -- succ is defined for Char values as chr . (1 +) . ord
map succ -- Answer 3
```

Question 8

```
foldr (&&) True . map (>= 0)
```

1. ✔ `and . map (>= 0)`
2. ✔ `all (>= 0)`
3. ✗ `any (>= 0)`
4. ✔ `foldr (\a b -> a >= 0 && b) True`
5. ✗ `foldl (\a b -> a && b > 0) True`

The following derivation shows the equivalence to answers 1 and 2.

```
foldr (&&) True . map (>=0)
= -- and = foldr (&&) True
and . map (>=0) -- Answer 1
= -- all f = and . map f
all (>=0) -- Answer 2
```

Furthermore, Answer 4 is also equivalent, as the following derivation shows:

```
foldr (&&) True . map (>=0)
= -- η-expansion on the (&&)
foldr (\a b -> a && b) True . map (>=0)
= -- We have a fold/map rule
  -- foldr (\x y -> z) . map f = foldr (\x y -> z[x := f x])
  -- (where z[x:=f x] is a substitution).
foldr (\a b -> (>= 0) a && b) True
= -- Nicer syntax
foldr (\a b -> a >= 0 && b) True -- Answer 4
```

Footnotes:

[1] : By "equivalent", we mean will evaluate to equal results. We consider two functions equal if, for any input, they produce equal outputs (functional extensionality).

Submission is already closed for this quiz. You can click here to check your submission (if any).