COMP105: Programming Paradigms
Class Test Preparation Questions

This document contains questions that are intended to aid with revision for the class test. Note that, while the class test will be multiple-choice, the questions here are not. Solutions for all of these questions are available on the COMP105 website.

Recursion.

1. Look at the following code:

   ```
   p n = if n == 0 then 1 else 2 * p (n-1)
   ```

   What is the result of the query `p 4`?

2. What is the most general type annotation for `p`?

3. Look at the following code:

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

   What is the result of the query `f "abcdef"`?

4. What is the most general type annotation for `f`?

5. What is *mutual recursion?* Give an example of a mutually recursive function.

6. What is *multiple recursion?* Give an example of a multiply recursive function.

7. What is *tail recursion?* Give an example of a function that uses tail recursion.

Higher order functions.

8. What is the type of the `.` operator?

9. Consider the following code:

   ```
   apply_twice f = f . f
   ```

What is the answer to the following query?

```
ghci> apply_twice (+1) 0
```

10. What is the type of `apply_twice`? Why?
11. What is the type of `apply_twice (+1)`? Why?
12. What are the answers to the following queries?

```
ghci> map (2^) [1..5]

ghci> map show [1..5]

ghci> map (\x -> show x !! 0) [10..19]
```

13. What are the answers to the following queries?

```
ghci> filter even [1..10]

ghci> filter (<5) [1..10]

ghci> filter (\x -> x `mod` 3 == 0 || x `mod` 5 == 0) [1..10]
```

14. What are the answers to the following queries?

```
ghci> foldr (*) 1 [1,2,3,4]

ghci> foldr (+) 0 [2,4..10]

ghci> foldr (\x acc -> x : acc) [] [1,2,3,4]

ghci> foldr1 (\x acc -> x) [1,2,3,4]
```

15. For each of the queries above, what happens when `foldr` is replaced with `scanr`?
16. For each of the queries above, what happens if `foldr` is replaced with `foldl`? Which of the queries need to be changed? Why?
17. What are the types of `dropWhile, takeWhile, and zipWith`?
18. What does the following query return? Why?

```
ghci> (dropWhile (==' ') . dropWhile (/=' ')) "one two three"
```

Custom types.

19. Consider the following custom type

```
data Direction = North | South | East | West deriving (Show, Read, Eq, Ord)
```

    What do the four type classes in the `deriving` clause represent? Which functions can we now use on the `Direction` type?

20. What are the results for the following queries?

```
ghci> North < South

ghci> West < East

ghci> show North

ghci> read "North" :: Direction
```

21. Consider the following custom type

```
data List a = Empty | Cons a (List a)
```

    Construct an instance of this type that is equivalent to `[1,2,3]`.

22. Which of these are valid instances of `List`?

```
Empty

Cons 1 Empty

1 `Cons` (2 `Cons` Empty)

1 `Cons` ('a' `Cons` Empty)
```

23. Consider the following code:

```
data Tree = Leaf | Branch Tree Tree deriving (Show)

g 0 = Leaf
g 1 = Leaf
g n = Branch (g (n-1)) (g (n-2))

depth Leaf = 1
depth (Branch l r) = 1 + max (depth l) (depth r)
```

    What is the result of the following query?

```
ghci> depth (g 4)
```

General questions

24. What does the `Maybe` type do? Where is it most useful?
25. What does the `Either` type do? Where is it most useful?
26. What is the `IO` type? Where is it used?
27. What is the difference between IO code and pure functional code?
28. What does it mean for a function have a *side effect*?
29. What does it mean for a function to be *deterministic*?
30. Consider the following code:

    ```
    action :: IO (Int)
    action = do
        x <- return 1
        y <- return 2
        z <- return 3
        return (x + y + z)
    ```

    What is the result of running `action`? What is the type of the returned value?
31. What is the result of the following query? Why?

    ```
    ghci> putStrLn (getLine)
    ```

32. What is the difference between *lazy* evaluation and *strict* evaluation?
33. What does the `$!` operator do? In what circumstances will the `$!` operator lead to less memory usage.