ASSIGNMENT 3
LAB TASK-1

NAME: B.THARUN
ROLL NO: 422116
SECTION: A

1.What is the difference between the type Char and the type String? Do the two expressions "a" and 'a' represent the same value?

Difference between Char and String
Char Represents a single character, denoted by single quotes (e.g., 'a').
String Represents a sequence of characters, denoted by double quotes (e.g., "a").

Expressions 'a' and "a" don't represent the same value: 'a' is a Char "a" is a String containing a single character

2.Given the function definition
3.square :: Int -> Int
square x = x * x
and the previous definitions of inc and double. What is the value of

1.inc (square 5)

square 5=5*5=25
inc (square 5) =25+1=26

2.square (inc 5)

inc 5 =5+1=6
square(inc 5) =6*6=36

3.average (inc 3) (inc 5)

inc 3=3+1=4
inc 5=5+1=6
average 4 6=(4+6)/2=5

4.If you remove the optional type annotation from the above definition of square, what type will the compiler infer? You can find out by pressing ⌘ -i in Haskell for Mac, while your cursor is on the function name, or by typing :type square or :t square in GHCi.

Num a=>a->a

```
ghci> :l sample.hs
[1 of 2] Compiling Main             ( sample.hs, interpreted )
Ok, one module loaded.
ghci> :t sq
sq :: Num a => a -> a
ghci> sq 4
16
ghci>
```

5.Which of the following identifiers can be function or variable names?

Valid Identifiers
square_1 :Valid
1square Invalid (cannot start with a number)
Square Valid (case matters; square is Square is different from square)
square! Invalid (exclamation marks aren't allowed)
square' Valid (apostrophes are allowed)

6.Define a new function showResult, that, for example, given the number 123, produces a string as follows:
showResult 123 ⇒ "The result is 123"

```
showResult :: Int -> String
showResult x = "The result is " ++ show x
```

```
ghci> :l sample.hs
[1 of 2] Compiling Main             ( sample.hs, interpreted )
Ok, one module loaded.
ghci> showResult 123
"The result is 123"
ghci>
```

7.Write a function showAreaOfCircle which, given the radius of a circle, calculates the area
of the circle,
showAreaOfCircle 12.3 ⇒ "The area of a circle with radius 12.3cm
is about 475.2915525615999 cm^2"

Use the show function, as well as the predefined value pi :: Floating a => a to write showAreaOfCircle.

```
showAreaOfCircle :: Float -> String
showAreaOfCircle radius = "The area of a circle with radius " ++ show radius ++ "cm is about " ++ show area ++ " cm^2"
  where
    area = pi * radius * radius
```

```
ghci> showAreaOfCircle 1
"The area of a circle with radius 1.0cm is about 3.1415927 cm^2"
ghci> showAreaOfCircle 5.2
"The area of a circle with radius 5.2cm is about 84.948654 cm^2"
ghci>
```

8.Write a function sort2 :: Ord a => a -> a -> (a, a) which accepts two Int values as arguments and returns them as a sorted pair, so that sort2 5 3 is equal to (3,5). How can you define the function using a conditional, how can you do it using guards?

Using Conditionals:

```
sort2 :: Ord a => a -> a -> (a, a)
sort2 x y = if x <= y then (x, y) else (y, x)
```

Using Guards:

```
sort2 :: Ord a => a -> a -> (a, a)
sort2 x y
  | x <= y    = (x, y)
  | otherwise = (y, x)
```

```
ghci> :l sample.hs
[1 of 2] Compiling Main             ( sample.hs, interpreted )
Ok, one module loaded.
ghci> sort2 5 3
(3,5)
ghci>
```

9.Consider a function

almostEqual :: Eq a => (a, a) -> (a, a) -> Bool

which compares the values of two pairs. It returns True if both pairs contain the same

values, regardless of the order. For example, almostEqual (3,4) (4,3) is True, but almostEqual (3,4) (3,5) is False. Which of the following definitions return the correct value? Which of the definitions would you consider good style? Why?

(The operator (&&) is logical ”and”, the operator (||) is logical ’or’, and (==) tests if two values are equal. The first two are of type Bool -> Bool -> Bool; the third is of type Eq a=> a -> a -> Bool.)

```
almostEqual (x1, y1) (x2, y2)
  | (x1 == x2) && (y1 == y2) = True
  | (x1 == y2) && (y1 == x2) = True
  | otherwise = False
```

This version is **correct**. It checks both the conditions where the pairs are ordered or swapped and returns True if the values match.

```
almostEqual (x1, y1) (x2, y2)
  | (x1 == x2) = (y1 == y2)
  | (x1 == y2) = (y1 == x2)
  | otherwise = False
```

This version is also **correct**. It checks the cases where the pairs are either equal or swapped.

```
almostEqual pair1 pair2
  = (pair1 == pair2) || (swap pair1 == pair2)
  where
    swap (x, y) = (y, x)
```

This version is **correct**. It checks if the pairs are equal as-is or if swapping the first pair makes them equal.

```
almostEqual pair1 pair2
  = (pair1 == pair2) || (swap pair1 == swap pair2)
  where
    swap (x, y) = (y, x)
```

This version is **incorrect**. It checks if swapping both pairs makes them equal, which doesn't handle the situation where only one pair needs to be swapped.

```
almostEqual (x1, y1) (x2, y2)
  = if (x1 == x2)
    then if (y1 == y2)
         then True
         else False
    else if (x1 == y2)
```

then if (x2 == y1)
then True
else False
else False

This version is **correct** but quite verbose.

Best Choice:

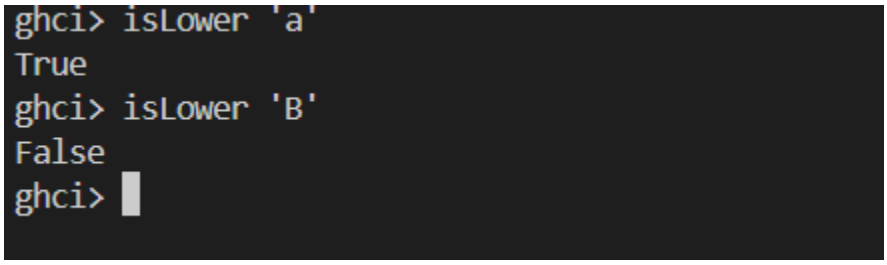
The most concise and efficient version is:

```
almostEqual pair1 pair2 = (pair1 == pair2) || (swap pair1 == pair2)
where
    swap (x, y) = (y, x)
```

It's preferred due to readability and avoiding redundant checks.

10.Define a function isLower :: Char -> Bool which returns True if a given character is a lower case letter. You can use the fact that characters are ordered, and for all lower case letters ch we have 'a' ≤ ch and ch ≤ 'z'.

```
isLower :: Char -> Bool
isLower ch = ch >= 'a' && ch <= 'z'
```



11.Write a function mangle :: String -> String which removes the first letter of a word and attaches it at the end. If the string is empty, mangle should simply return an empty string:

mangle "Hello" ⇒ "elloH"
mangle "l" ⇒ "l"
mangle "" ⇒ ""

```
mangle :: String -> String
mangle "" = ""
mangle (x:xs) = xs ++ [x]
```

```
ghci> mangle "Hello"
"elloH"
ghci> mangle "I"
"I"
ghci> mangle ""
""
```

12.Implement division on Int, divide :: Int -> Int -> Int using the list functions described in this section. Hint: first, write a function that returns all the multiples of a given number up to a specific limit.

divide 5 10 ⇒ 2

divide 5 8 ⇒ 1

divide 3 10 ⇒ 3

```
multiples :: Int -> Int -> [Int]
multiples x limit = takeWhile (<= limit) [x, 2 * x..]

divide :: Int -> Int -> Int
divide x y = length (multiples x y)
```

```
ghci> divide 10 5
0
ghci> divide 5 10
2
ghci> divide 5 8
1
ghci> divide 3 10
3
ghci>
```

13.Define the function length :: [a] -> Int. It is quite similar to sum and product in the way it traverses its input list. Since length is defined in the Prelude, don't forget to hide it by adding the line

import Prelude hiding (length)

to your module.

```
import Prelude hiding (length)
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

```
ghci> length [1,2,3,4]
4
ghci> length []
0
ghci>
```

14.What are the values of the following expressions and what is wrong with the ones that give errors?

1:[2,3,4]
1:2:3:4:[]
[1,2,3]:[4..7]
[1,2,3] ++ [4..7]
1:['a','b']
"abc"++"cd"
"a" : "bCc"
"a" ++ "bCc"
'a':'b'
'a':"b"
[1,4,7] ++ 4:[5:[]]
[True,True:[]]
True:[True,False]

`1:[2,3,4]` : `[1, 2, 3, 4]`.
`1:2:3:4:[]` : `[1, 2, 3, 4]`.
`[1,2,3]:[4..7]` : Invalid, mismatched types (list cannot be prepended to another list like this).
`[1,2,3] ++ [4..7]` : `[1, 2, 3, 4, 5, 6, 7]`.
`1:['a','b']` : Invalid, mismatched types (trying to mix `Int` and `Char`).
`"abc" ++ "cd"` : `"abccd"`.
`"a":"bCc"` : Invalid, mismatched types (you can't prepend a `String` to another string like this).
`"a" ++ "bCc"` : `"abCc"`.
`'a':'b'` : Invalid, mismatched types.
`'a':"b"` : `"ab"`.
`[1,4,7] ++ 4:[5:[]]`: Invalid, mismatched types.
`[True,True:[]]` : Invalid, incorrect list construction.
`True:[True,False]` : `[True, True, False]`.

15.Write a recursive function fact to compute the factorial of a given positive number (ignore the case of 0 for this exercise). fact n = 1 * 2 * ... * n Why is the function fact a partial function? Add an appropriate error case to the function definition.

```
fact :: Int -> Int
fact 0 = 1
fact n
  | n < 0     = error "Factorial of a negative number is undefined"
  | otherwise = n * fact (n - 1)
```

```
ghci> fact 5
120
ghci> fact 3
6
ghci> fact (-4)
*** Exception: Factorial of a negative number is undefined
CallStack (from HasCallStack):
  error, called at sample.hs:10:17 in main:Main
ghci>
```

16.In the previous chapter, we introduced the ellipsis list notation in Haskell, which allows us to write
[m..n]
as shorthand for the list
[m, m+1, m+2, ..., n]
for numbers m and n, with n greater or equal m. Write a recursive function enumFromTo which produces such a list given m and n, such that
enumFromTo m n = [m..n]
As enumFromTo is a Prelude function, you have to add the line
import Prelude hiding (enumFromTo)
to your program.

```
import Prelude hiding (enumFromTo)

enumFromTo :: Int -> Int -> [Int]
enumFromTo m n
  | m > n     = []
  | otherwise = m : enumFromTo (m + 1) n
```

```
ghci> enumFromTo 3 8
[3,4,5,6,7,8]
ghci> enumFromTo 10 3
[]
ghci>
```

17.Write a recursive function countOdds which calculates the number of odd elements in a list of Int values:
countOdds [1, 6, 9, 14, 16, 22] = 2
Hint: You can use the Prelude function odd :: Int -> Bool, which tests whether a number is odd.

```
countOdds :: [Int] -> Int
countOdds [] = 0
countOdds (x:xs)
  | odd x     = 1 + countOdds xs
  | otherwise = countOdds xs
```

```
ghci> countOdds [1,2,3,4,5]
3
ghci> countOdds [0,9,8,7,6,5,4,3,2,1]
5
ghci>
```

18.Write a recursive function removeOdd that, given a list of integers, removes all odd numbers from the list, e.g.,
removeOdd [1, 4, 5, 7, 10] = [4, 10]

```
removeOdd :: [Int] -> [Int]
removeOdd [] = []
removeOdd (x:xs)
  | odd x     = removeOdd xs
  | otherwise = x : removeOdd xs
```

```
ghci> removeOdd [1,2,3,4,5]
[2,4]
ghci> removeOdd [0,9,8,7,6,5,4,3,2,1]
[0,8,6,4,2]
ghci>
```

19.Challenge: At the end of the last screencast, demonstrating the implementation of

closestPoint :: Point -> [Point] -> Point,

we mentioned that the final implementation is less efficient than one might hope, as it uses the distance functions twice —instead of once— per recursive step. Improve the implementation to avoid that inefficiency.

20.Implement a function colouredFTree :: Float -> Int -> Colour -> Line -> Picture that elaborates on fractalTree by accepting the colour of the tree as an additional argument.

21.Vary colouredFTree by using the fade function, which we discussed in the context of spiral rays, to incrementally alter the colour in each recursive step.

22. Vary colouredFTree further by implementing and using factor as demonstrated in the last screencast.