19CSE313

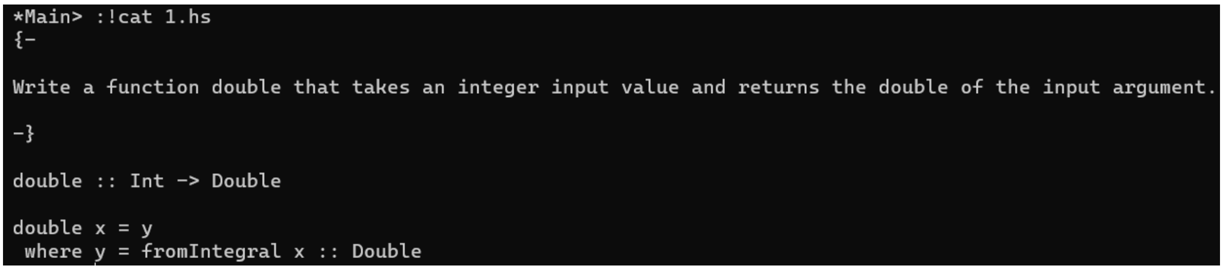
Principles of Programming Languages

Lab 3

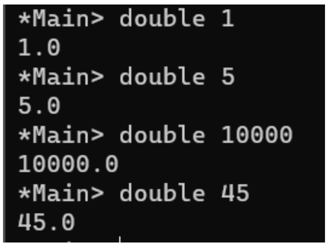
S Abhishek

AM.EN.U4CSE19147

1 - Write a function double that takes an integer input value and returns the double of the input argument.

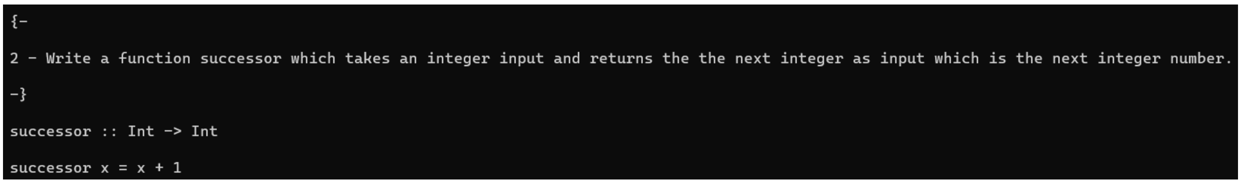


```
*Main> :!cat 1.hs
{-

Write a function double that takes an integer input value and returns the double of the input argument.

-}

double :: Int -> Double

double x = y
 where y = fromIntegral x :: Double
```

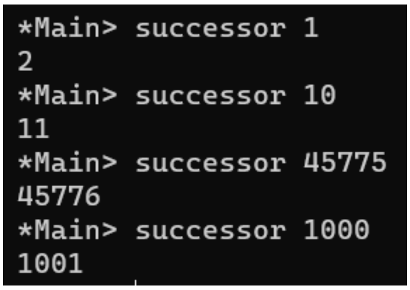


```
*Main> double 1
1.0
*Main> double 5
5.0
*Main> double 10000
10000.0
*Main> double 45
45.0
```

2 - Write a function successor which takes an integer input and returns the next integer as input which is the next integer number.

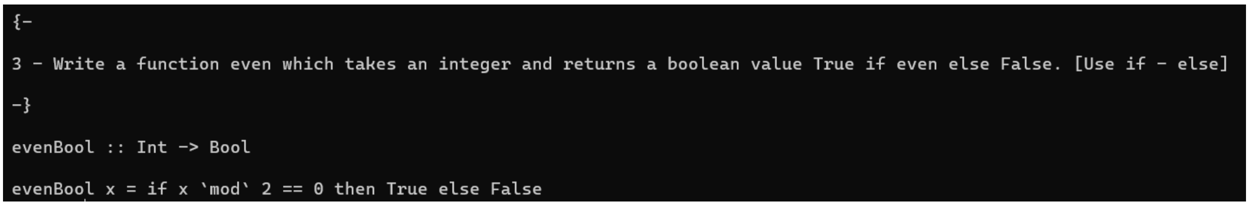


```
{-

2 - Write a function successor which takes an integer input and returns the the next integer as input which is the next integer number.

-}

successor :: Int -> Int

successor x = x + 1
```

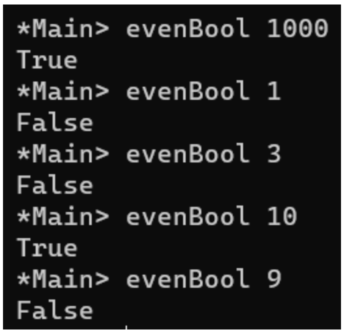


```
*Main> successor 1
2
*Main> successor 10
11
*Main> successor 45775
45776
*Main> successor 1000
1001
```

3 - Write a function even which takes an integer and returns a boolean value True if even else False. [Use if - else]

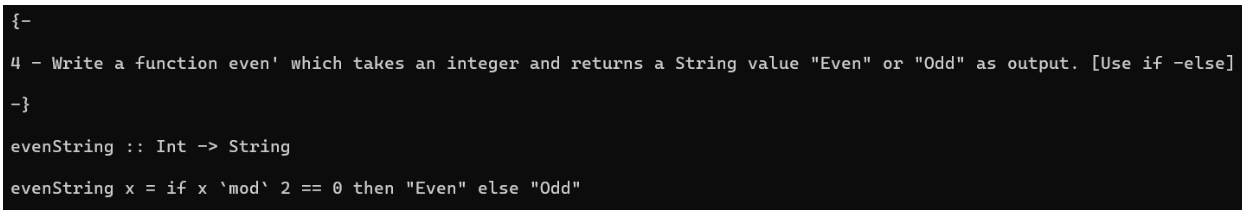


```
{-

3 - Write a function even which takes an integer and returns a boolean value True if even else False. [Use if - else]

-}

evenBool :: Int -> Bool

evenBool x = if x `mod` 2 == 0 then True else False
```

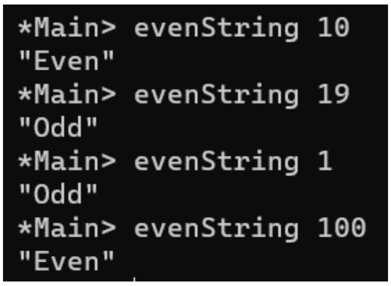


```
*Main> evenBool 1000
True
*Main> evenBool 1
False
*Main> evenBool 3
False
*Main> evenBool 10
True
*Main> evenBool 9
False
```

4 - Write a function even' which takes an integer and returns a String value "Even" or "Odd" as output. [Use if -else]

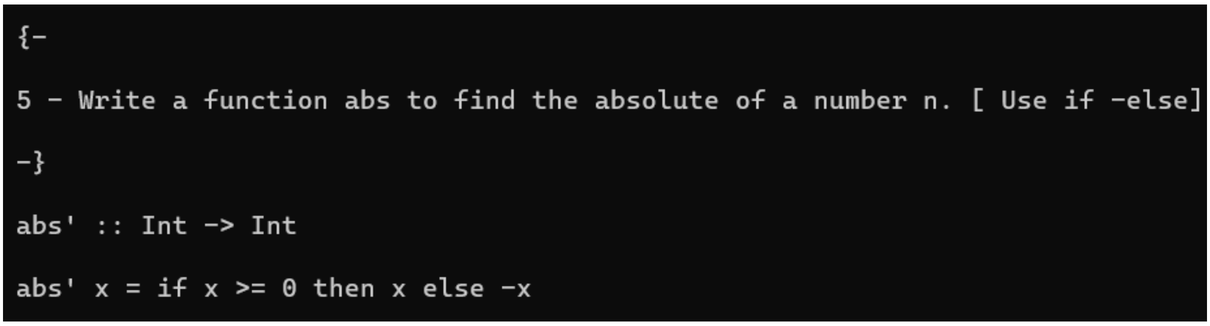


```
{-

4 - Write a function even' which takes an integer and returns a String value "Even" or "Odd" as output. [Use if -else]

-}

evenString :: Int -> String

evenString x = if x `mod` 2 == 0 then "Even" else "Odd"
```

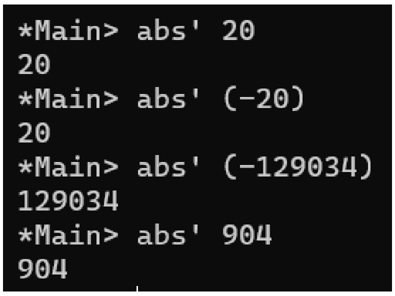


```
*Main> evenString 10
"Even"
*Main> evenString 19
"Odd"
*Main> evenString 1
"Odd"
*Main> evenString 100
"Even"
```

5 - Write a function abs to find the absolute of a number n. [Use if -else]

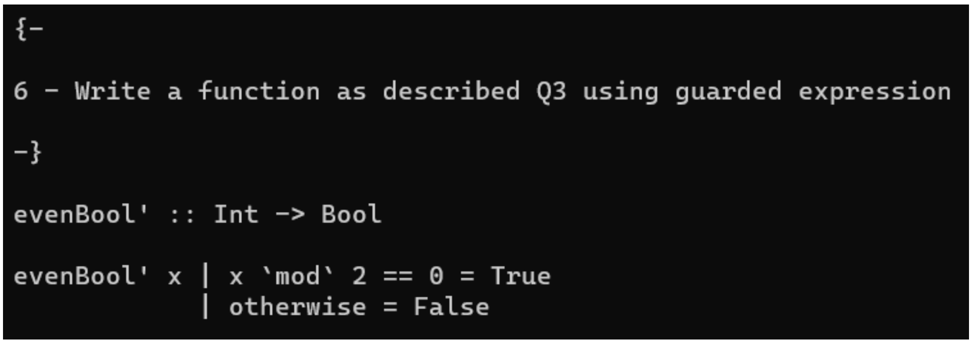


```
{-

5 - Write a function abs to find the absolute of a number n. [ Use if -else]

-}

abs' :: Int -> Int

abs' x = if x >= 0 then x else -x
```

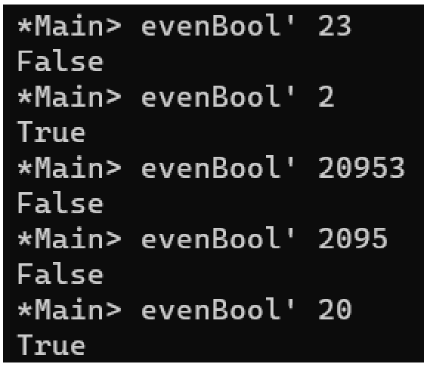


```
*Main> abs' 20
20
*Main> abs' (-20)
20
*Main> abs' (-129034)
129034
*Main> abs' 904
904
```

6 - Write a function as described Q3 using guarded expression

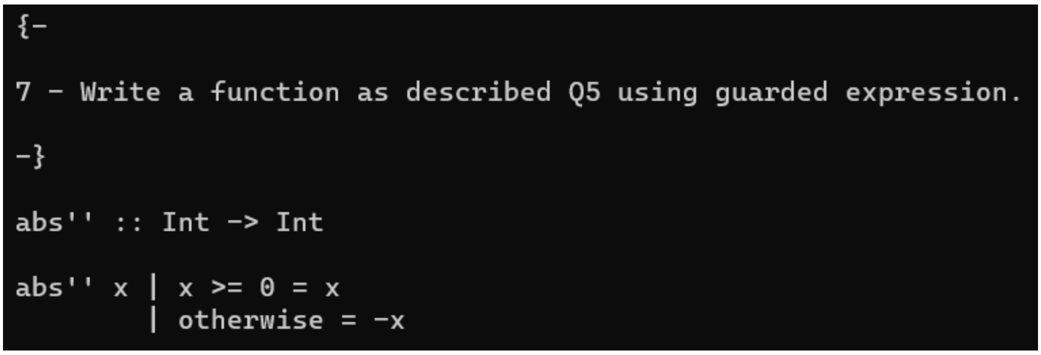


```
{-

6 - Write a function as described Q3 using guarded expression

-}

evenBool' :: Int -> Bool

evenBool' x | x `mod` 2 == 0 = True
            | otherwise = False
```

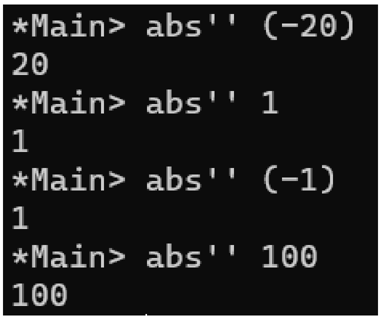


```
*Main> evenBool' 23
False
*Main> evenBool' 2
True
*Main> evenBool' 20953
False
*Main> evenBool' 2095
False
*Main> evenBool' 20
True
```

7 - Write a function as described Q5 using guarded expression.

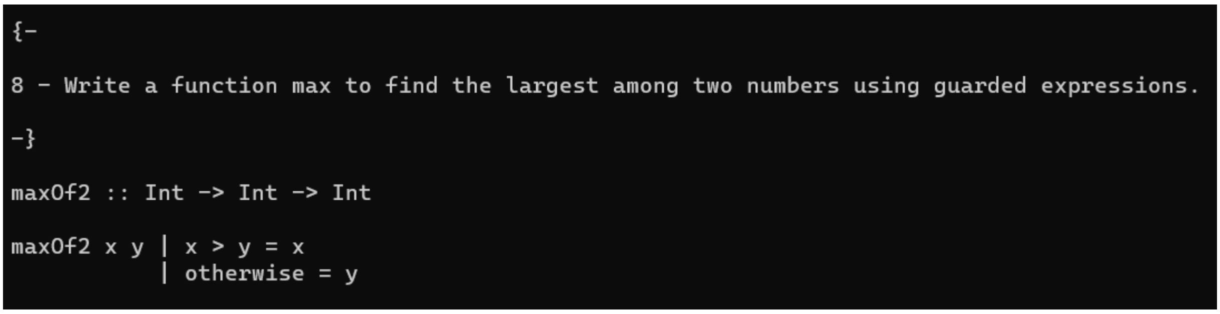


```
{-

7 - Write a function as described Q5 using guarded expression.

-}

abs'' :: Int -> Int

abs'' x | x >= 0 = x
        | otherwise = -x
```

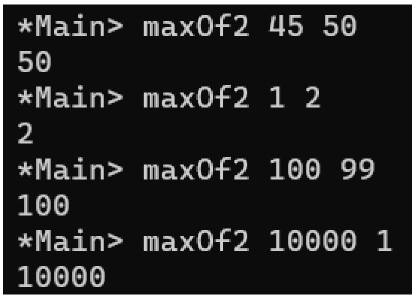


```
*Main> abs'' (-20)
20
*Main> abs'' 1
1
*Main> abs'' (-1)
1
*Main> abs'' 100
100
```

8 - Write a function max to find the largest among two numbers using guarded expressions.

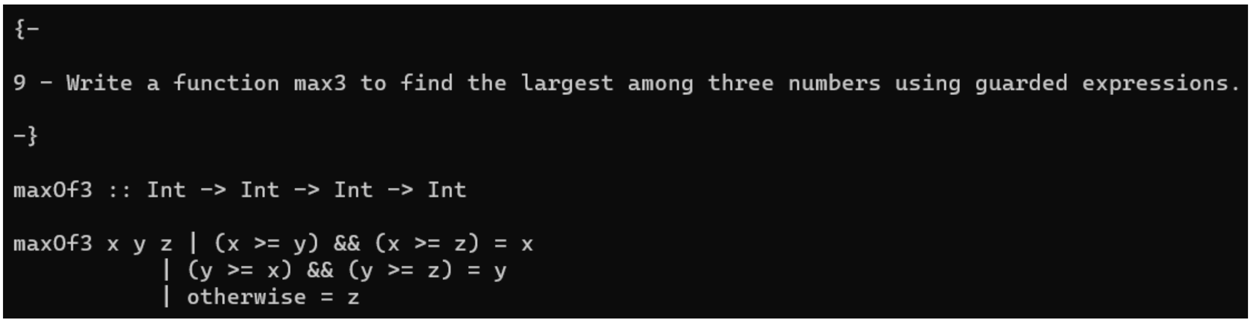


```
{-

8 - Write a function max to find the largest among two numbers using guarded expressions.

-}

maxOf2 :: Int -> Int -> Int

maxOf2 x y | x > y = x
           | otherwise = y
```

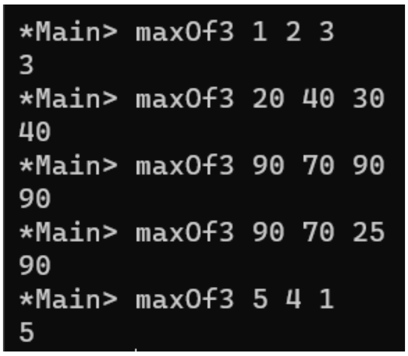


```
*Main> maxOf2 45 50
50
*Main> maxOf2 1 2
2
*Main> maxOf2 100 99
100
*Main> maxOf2 10000 1
10000
```

9 - Write a function max3 to find the largest among three numbers using guarded expressions.



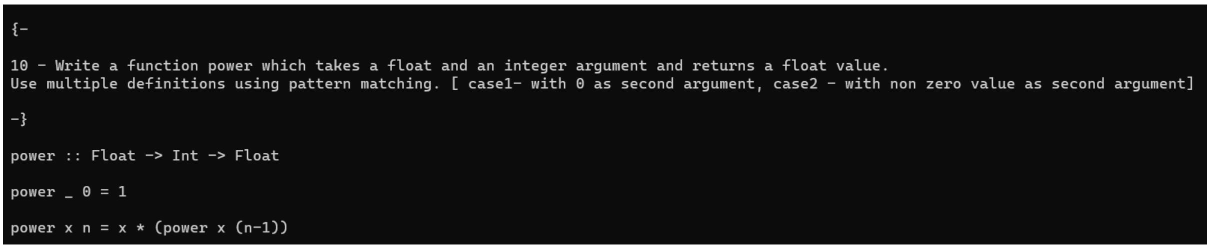
```
{-

9 - Write a function max3 to find the largest among three numbers using guarded expressions.

-}

maxOf3 :: Int -> Int -> Int -> Int

maxOf3 x y z | (x >= y) && (x >= z) = x
             | (y >= x) && (y >= z) = y
             | otherwise = z
```



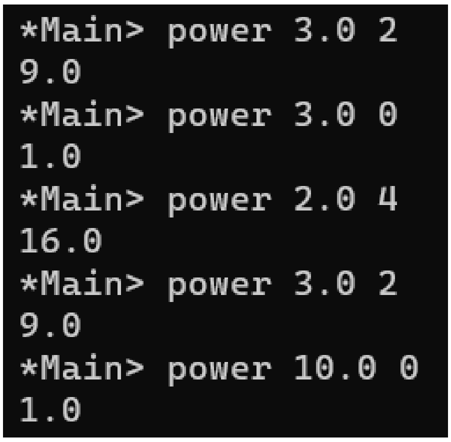
```
*Main> maxOf3 1 2 3
3
*Main> maxOf3 20 40 30
40
*Main> maxOf3 90 70 90
90
*Main> maxOf3 90 70 25
90
*Main> maxOf3 5 4 1
5
```

10 - Write a function power which takes a float and an integer argument and returns a float value.

- Use multiple definitions using pattern matching.
- Case1- with 0 as second argument
- Case2 - with nonzero value as second argument



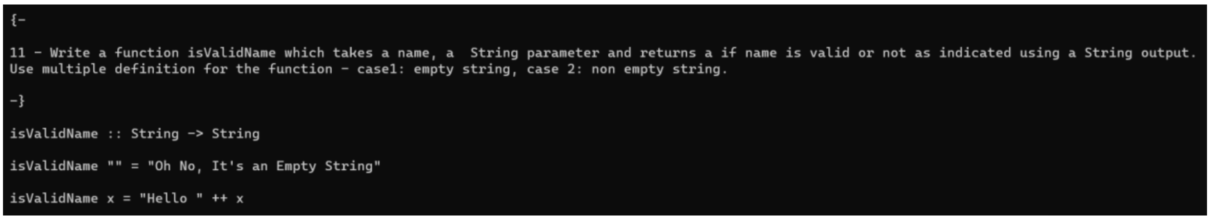
```
{-

10 - Write a function power which takes a float and an integer argument and returns a float value.
Use multiple definitions using pattern matching. [ case1- with 0 as second argument, case2 - with non zero value as second argument]

-}

power :: Float -> Int -> Float

power _ 0 = 1

power x n = x * (power x (n-1))
```



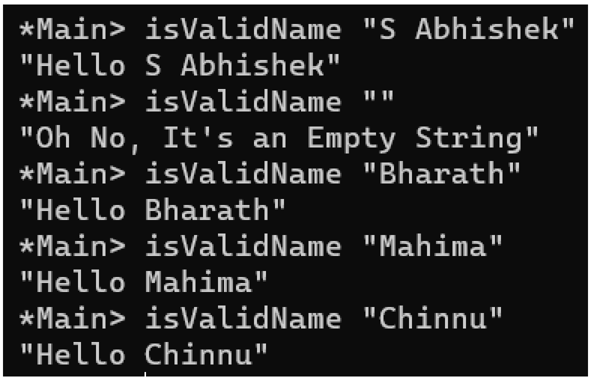
```
*Main> power 3.0 2
9.0
*Main> power 3.0 0
1.0
*Main> power 2.0 4
16.0
*Main> power 3.0 2
9.0
*Main> power 10.0 0
1.0
```

11 - Write a function isValidName which takes a name, a String parameter and returns String.

- If name is valid or not as indicated using a String output.
- Use multiple definition for the function.
- Case1: Empty string
- Case 2: Non empty string.



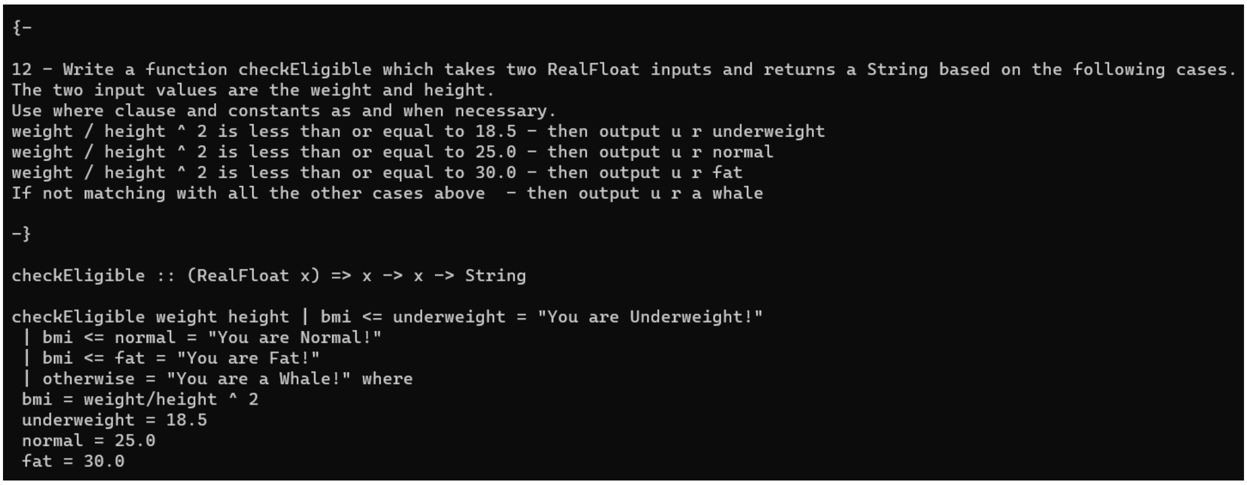
```
{-

11 - Write a function isValidName which takes a name, a  String parameter and returns a if name is valid or not as indicated using a String output.
Use multiple definition for the function - case1: empty string, case 2: non empty string.

-}

isValidName :: String -> String

isValidName "" = "Oh No, It's an Empty String"

isValidName x = "Hello " ++ x
```



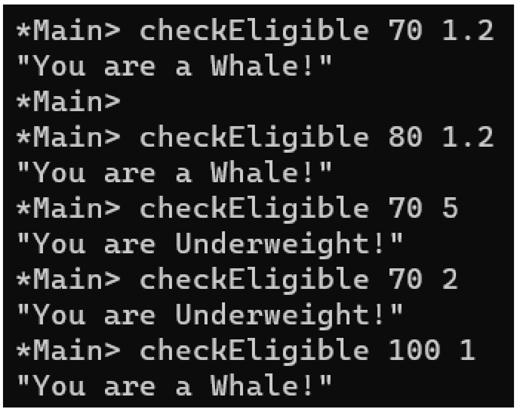
```
*Main> isValidName "S Abhishek"
"Hello S Abhishek"
*Main> isValidName ""
"Oh No, It's an Empty String"
*Main> isValidName "Bharath"
"Hello Bharath"
*Main> isValidName "Mahima"
"Hello Mahima"
*Main> isValidName "Chinnu"
"Hello Chinnu"
```

12 - Write a function checkEligible which takes two RealFloat inputs and returns a String based on the following cases.

- The two input values are the weight and height.
- These are the following cases [use where clause and constants as and when necessary]
    - weight / height ^ 2 is less than or equal to 18.5 - then output u r underweight
    - weight / height ^ 2 is less than or equal to 25.0 - then output u r normal
    - weight / height ^ 2 is less than or equal to 30.0 - then output u r fat
    - If not matching with all the other cases above - then output u r a whale.



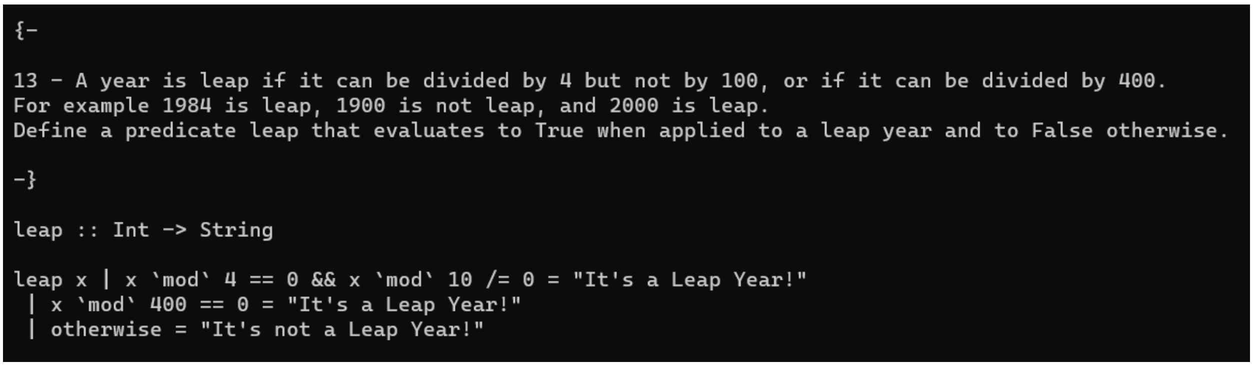
```
{-

12 - Write a function checkEligible which takes two RealFloat inputs and returns a String based on the following cases.
The two input values are the weight and height.
Use where clause and constants as and when necessary.
weight / height ^ 2 is less than or equal to 18.5 - then output u r underweight
weight / height ^ 2 is less than or equal to 25.0 - then output u r normal
weight / height ^ 2 is less than or equal to 30.0 - then output u r fat
If not matching with all the other cases above  - then output u r a whale

-}

checkEligible :: (RealFloat x) => x -> x -> String

checkEligible weight height | bmi <= underweight = "You are Underweight!"
 | bmi <= normal = "You are Normal!"
 | bmi <= fat = "You are Fat!"
 | otherwise = "You are a Whale!" where
 bmi = weight/height ^ 2
 underweight = 18.5
 normal = 25.0
 fat = 30.0
```



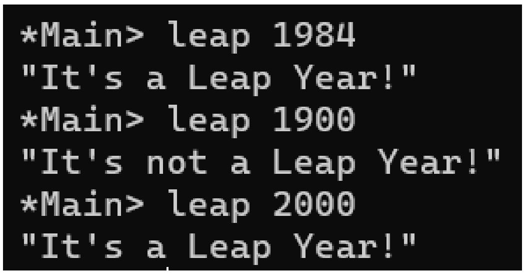
```
*Main> checkEligible 70 1.2
"You are a Whale!"
*Main>
*Main> checkEligible 80 1.2
"You are a Whale!"
*Main> checkEligible 70 5
"You are Underweight!"
*Main> checkEligible 70 2
"You are Underweight!"
*Main> checkEligible 100 1
"You are a Whale!"
```

13 - A year is leap if it can be divided by 4 but not by 100, or if it can be divided by 400.

- For example, 1984 is leap, 1900 is not leap, and 2000 is leap.
- Define a predicate leap that evaluates to True when applied to a leap year and to False otherwise.



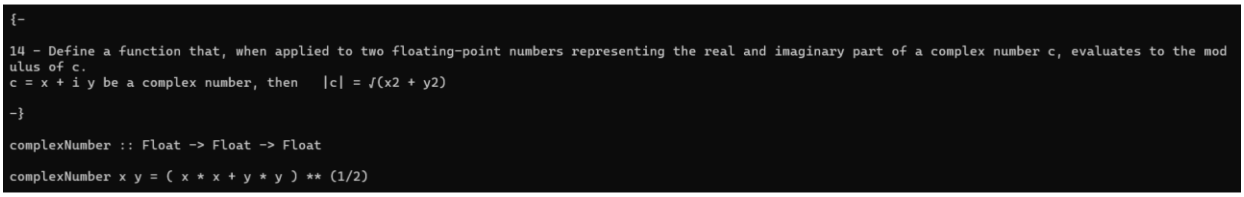
```
{-

13 - A year is leap if it can be divided by 4 but not by 100, or if it can be divided by 400.
For example 1984 is leap, 1900 is not leap, and 2000 is leap.
Define a predicate leap that evaluates to True when applied to a leap year and to False otherwise.

-}

leap :: Int -> String

leap x | x `mod` 4 == 0 && x `mod` 10 /= 0 = "It's a Leap Year!"
 | x `mod` 400 == 0 = "It's a Leap Year!"
 | otherwise = "It's not a Leap Year!"
```



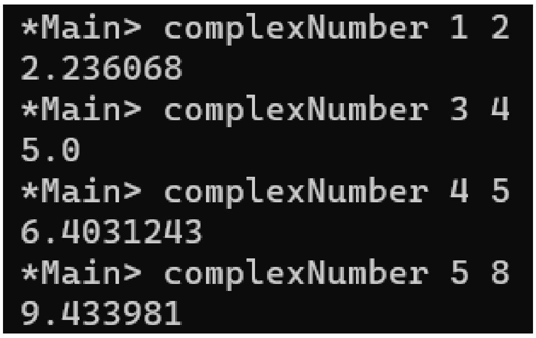
```
*Main> leap 1984
"It's a Leap Year!"
*Main> leap 1900
"It's not a Leap Year!"
*Main> leap 2000
"It's a Leap Year!"
```

14 - Define a function that, when applied to two floating-point numbers representing the real and imaginary part of a complex number c, evaluates to the modulus of c.

- c = x + i y be a complex number, then |c| = √(x2 + y2)



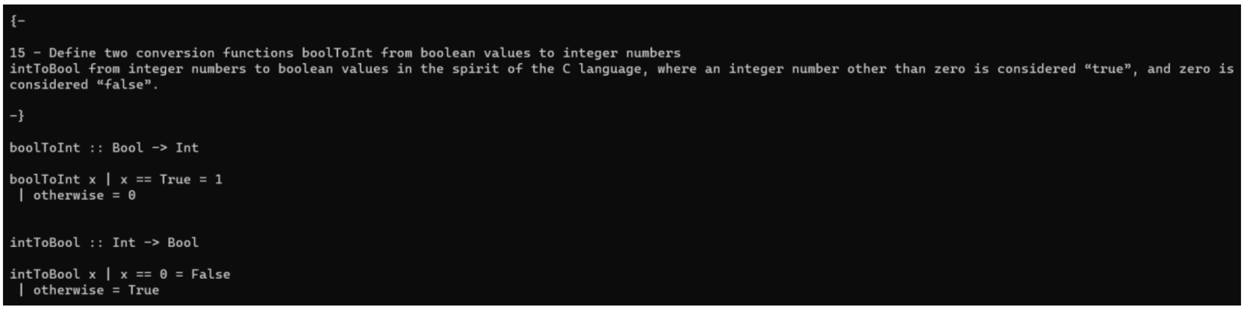
```
{-

14 - Define a function that, when applied to two floating-point numbers representing the real and imaginary part of a complex number c, evaluates to the mod
ulus of c.
c = x + i y be a complex number, then   |c| = √(x2 + y2)

-}

complexNumber :: Float -> Float -> Float

complexNumber x y = ( x * x + y * y ) ** (1/2)
```



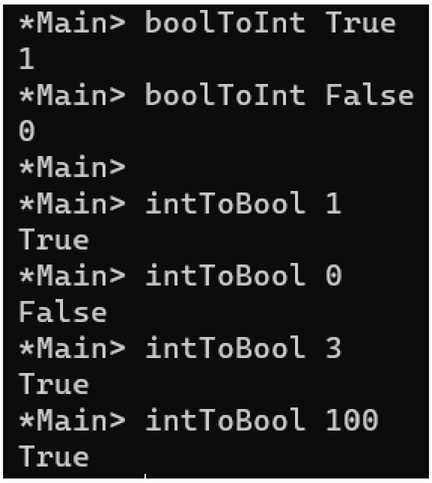
```
*Main> complexNumber 1 2
2.236068
*Main> complexNumber 3 4
5.0
*Main> complexNumber 4 5
6.4031243
*Main> complexNumber 5 8
9.433981
```

15 - Define two conversion functions,

- boolToInt from boolean values to integer numbers
- intToBool from integer numbers to boolean values where an integer number other than zero is considered “true”, and zero is considered “false”.



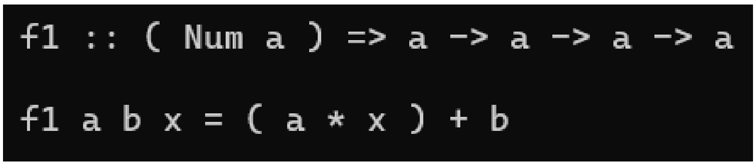
```
{-

15 - Define two conversion functions boolToInt from boolean values to integer numbers
intToBool from integer numbers to boolean values in the spirit of the C language, where an integer number other than zero is considered “true”, and zero is
considered “false”.

-}

boolToInt :: Bool -> Int

boolToInt x | x == True = 1
 | otherwise = 0


intToBool :: Int -> Bool

intToBool x | x == 0 = False
 | otherwise = True
```



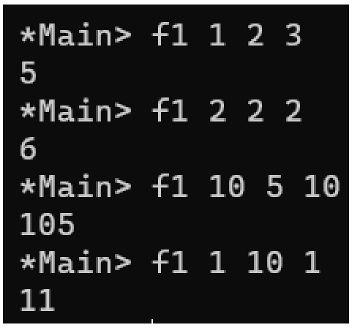
```
*Main> boolToInt True
1
*Main> boolToInt False
0
*Main>
*Main> intToBool 1
True
*Main> intToBool 0
False
*Main> intToBool 3
True
*Main> intToBool 100
True
```

16 - Write Haskell functions corresponding to the following mathematical functions

$$f(a,b,x) \quad = \quad ax+b$$

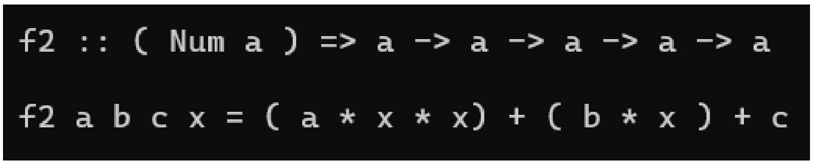


```
f1 :: ( Num a ) => a -> a -> a -> a

f1 a b x = ( a * x ) + b
```

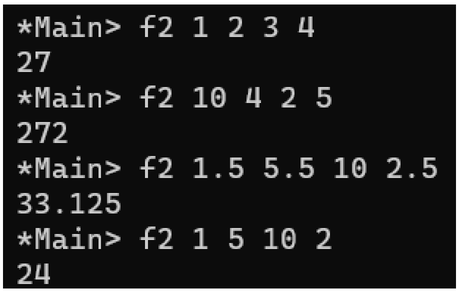


```
*Main> f1 1 2 3
5
*Main> f1 2 2 2
6
*Main> f1 10 5 10
105
*Main> f1 1 10 1
11
```

$$f(a,b,c,x) \quad = \quad ax^2 + bx + c$$

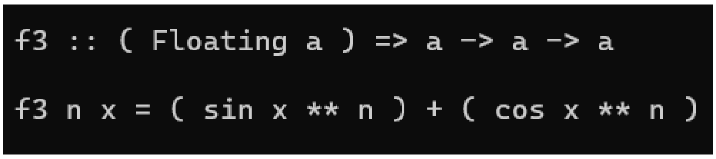


```
f2 :: ( Num a ) => a -> a -> a -> a -> a

f2 a b c x = ( a * x * x) + ( b * x ) + c
```

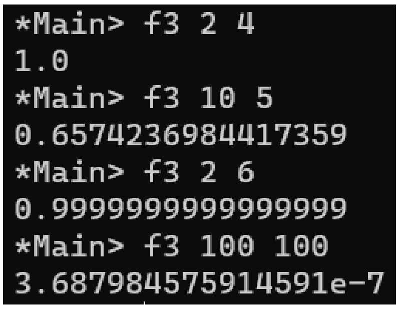


```
*Main> f2 1 2 3 4
27
*Main> f2 10 4 2 5
272
*Main> f2 1.5 5.5 10 2.5
33.125
*Main> f2 1 5 10 2
24
```

$$f(n,x) \quad = \quad \sin^n x + \cos^n x$$

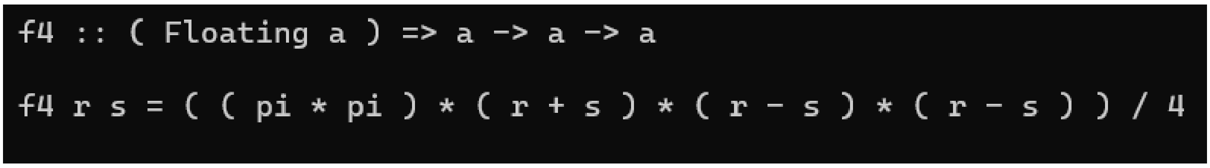


```
f3 :: ( Floating a ) => a -> a -> a

f3 n x = ( sin x ** n ) + ( cos x ** n )
```

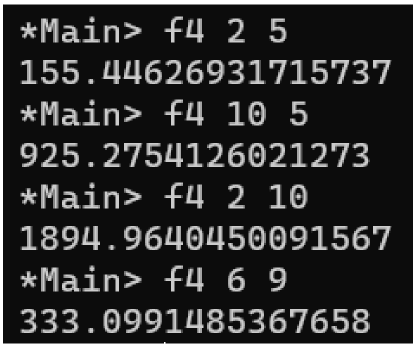


```
*Main> f3 2 4
1.0
*Main> f3 10 5
0.6574236984417359
*Main> f3 2 6
0.9999999999999999
*Main> f3 100 100
3.687984575914591e-7
```

$$f(r,s) \quad = \quad \frac{\pi^2(r+s)(r-s)^2}{4}$$

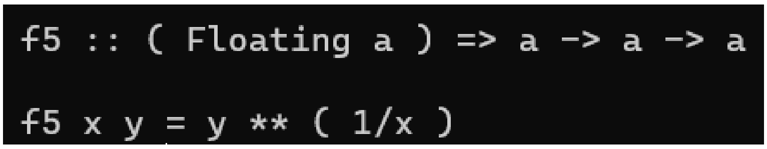


```
f4 :: ( Floating a ) => a -> a -> a

f4 r s = ( ( pi * pi ) * ( r + s ) * ( r - s ) * ( r - s ) ) / 4
```

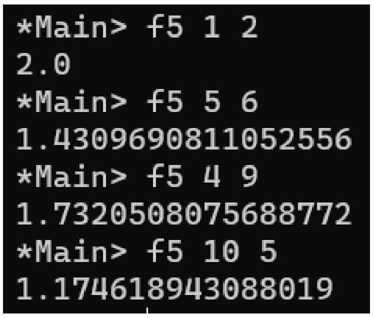


```
*Main> f4 2 5
155.44626931715737
*Main> f4 10 5
925.2754126021273
*Main> f4 2 10
1894.9640450091567
*Main> f4 6 9
333.0991485367658
```

$$f(x, y) = \sqrt[x]{y}$$



```
f5 :: ( Floating a ) => a -> a -> a

f5 x y = y ** ( 1/x )
```



```
*Main> f5 1 2
2.0
*Main> f5 5 6
1.4309690811052556
*Main> f5 4 9
1.7320508075688772
*Main> f5 10 5
1.174618943088019
```

Thankyou!!