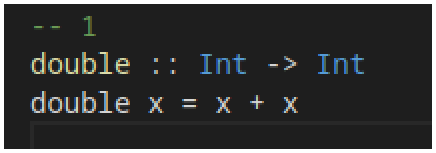
19CSE313-Principles of Programming Languages

Lab Exercise-3

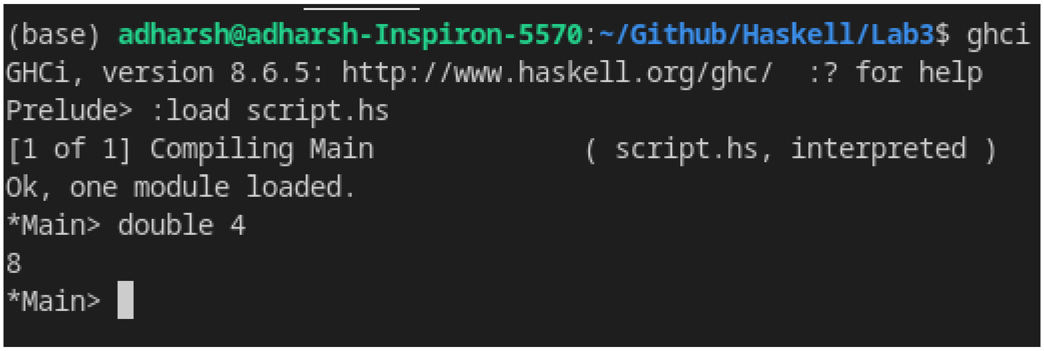
Done By:

Adharsh S Mathew
CSE-D
AM.EN.U4CSE19302

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

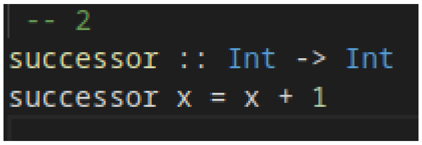


```
-- 1
double :: Int -> Int
double x = x + x
```

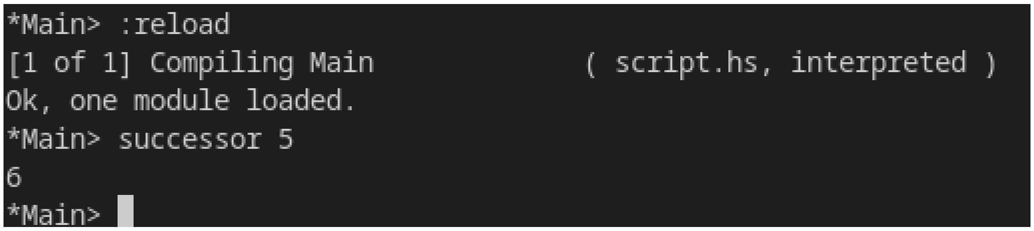


```
(base) adharsh@adharsh-Inspiron-5570:~/Github/Haskell/Lab3$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :load script.hs
[1 of 1] Compiling Main             ( script.hs, interpreted )
Ok, one module loaded.
*Main> double 4
8
*Main>
```

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

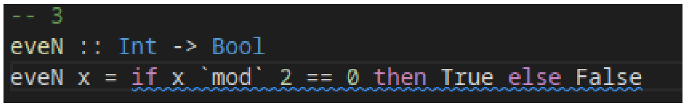


```
-- 2
successor :: Int -> Int
successor x = x + 1
```

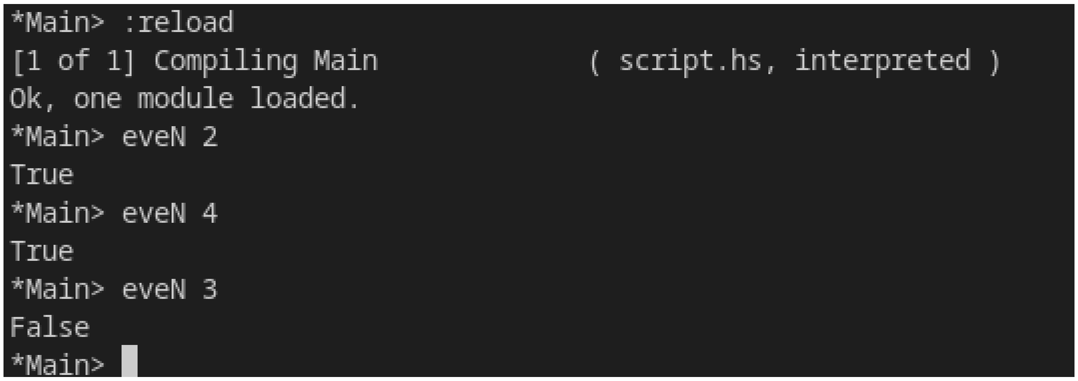


```
*Main> :reload
[1 of 1] Compiling Main             ( script.hs, interpreted )
Ok, one module loaded.
*Main> successor 5
6
*Main>
```

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



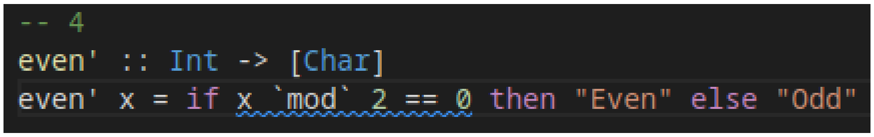
```
-- 3
eveN :: Int -> Bool
eveN x = if x `mod` 2 == 0 then True else False
```



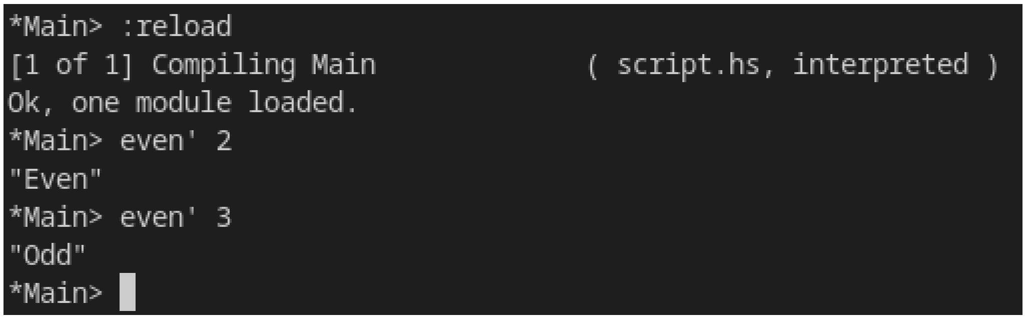
```
*Main> :reload
[1 of 1] Compiling Main             ( script.hs, interpreted )
Ok, one module loaded.
*Main> eveN 2
True
*Main> eveN 4
True
*Main> eveN 3
False
*Main>
```

// an ambiguity error was faced when “even” was used. So it is changed to “eveN” to fix the error.

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

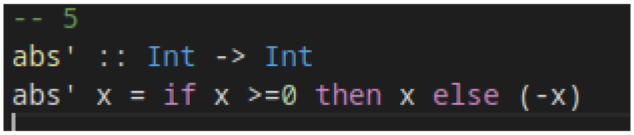


```
-- 4
even' :: Int -> [Char]
even' x = if x `mod` 2 == 0 then "Even" else "Odd"
```

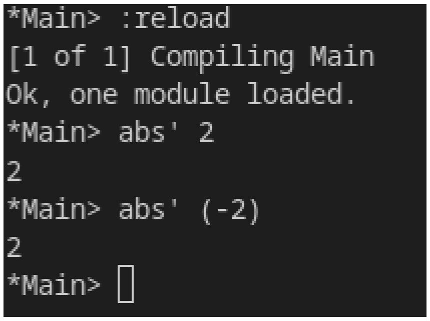


```
*Main> :reload
[1 of 1] Compiling Main             ( script.hs, interpreted )
Ok, one module loaded.
*Main> even' 2
"Even"
*Main> even' 3
"Odd"
*Main>
```

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

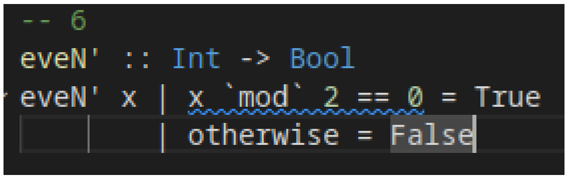


```
-- 5
abs' :: Int -> Int
abs' x = if x >=0 then x else (-x)
```

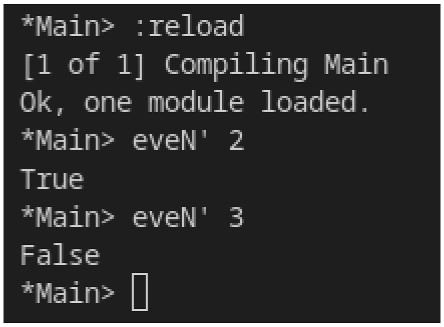


```
*Main> :reload
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> abs' 2
2
*Main> abs' (-2)
2
*Main>
```

6. Write a function as described **Q3** using guarded expression.

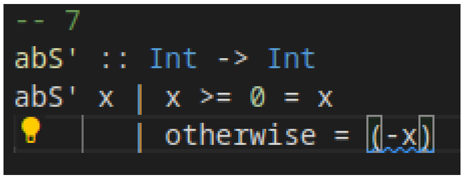


```
-- 6
eveN' :: Int -> Bool
eveN' x | x `mod` 2 == 0 = True
        | otherwise = False
```

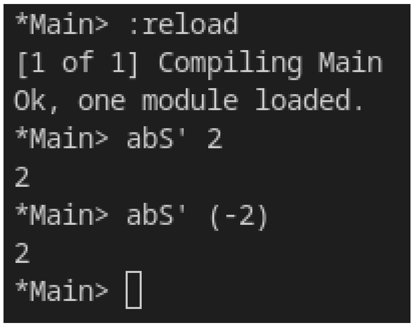


```
*Main> :reload
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> eveN' 2
True
*Main> eveN' 3
False
*Main>
```

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

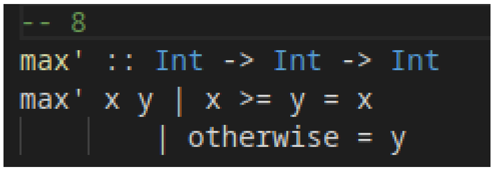


```
-- 7
abS' :: Int -> Int
abS' x | x >= 0 = x
       | otherwise = (-x)
```

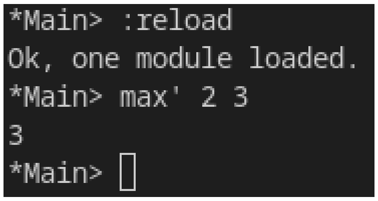


```
*Main> :reload
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> abS' 2
2
*Main> abS' (-2)
2
*Main>
```

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

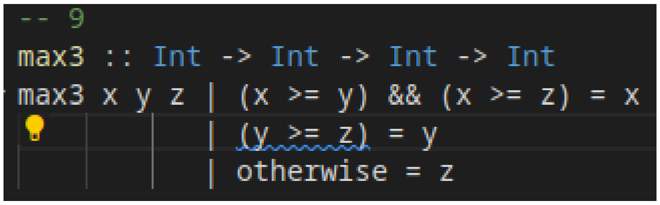


```
-- 8
max' :: Int -> Int -> Int
max' x y | x >= y = x
         | otherwise = y
```

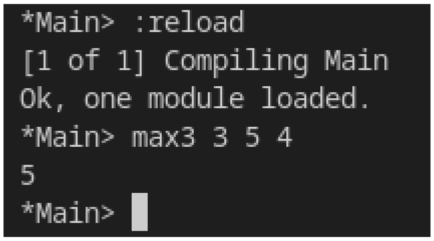


```
*Main> :reload
Ok, one module loaded.
*Main> max' 2 3
3
*Main>
```

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

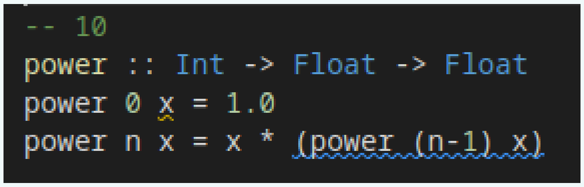


```
-- 9
max3 :: Int -> Int -> Int -> Int
max3 x y z | (x >= y) && (x >= z) = x
           | (y >= z) = y
           | otherwise = z
```

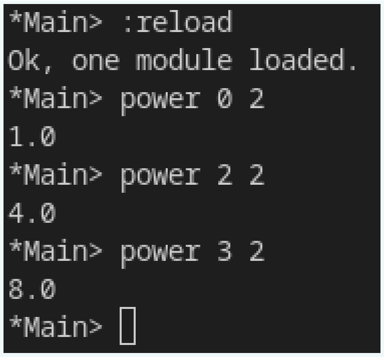


```
*Main> :reload
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> max3 3 5 4
5
*Main>
```

10. Write a function **power** which takes a float and an integer argument and returns a float value. Use multiple definitions using pattern matching.

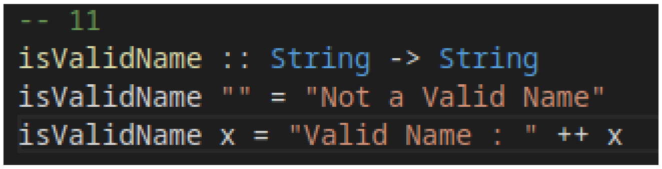


```haskell
-- 10
power :: Int -> Float -> Float
power 0 x = 1.0
power n x = x * (power (n-1) x)
```

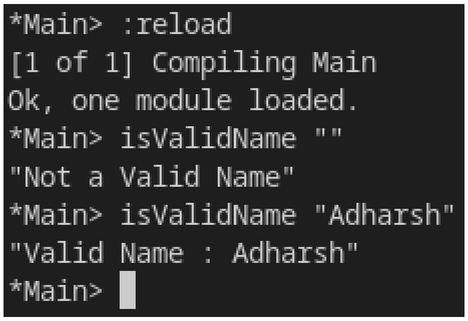


```
*Main> :reload
Ok, one module loaded.
*Main> power 0 2
1.0
*Main> power 2 2
4.0
*Main> power 3 2
8.0
*Main> 
```

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



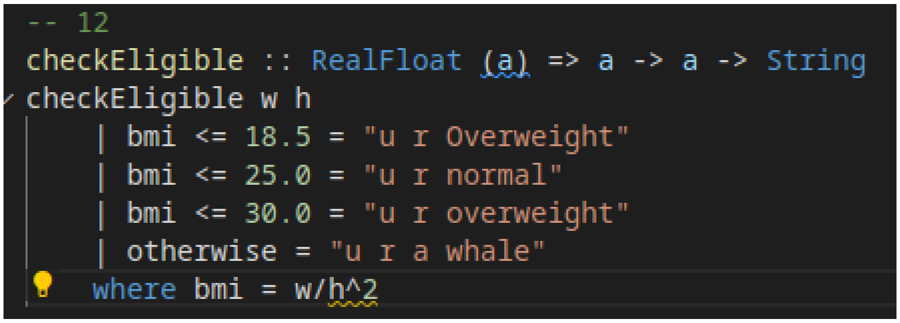
```haskell
-- 11
isValidName :: String -> String
isValidName "" = "Not a Valid Name"
isValidName x = "Valid Name : " ++ x
```



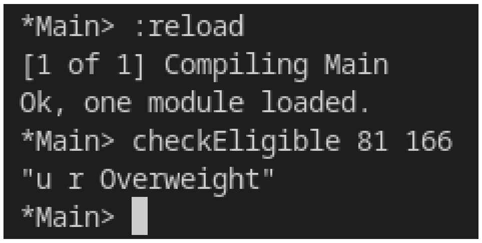
```
*Main> :reload
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> isValidName ""
"Not a Valid Name"
*Main> isValidName "Adharsh"
"Valid Name : Adharsh"
*Main> 
```

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]

1. weight / height ^ 2 is less than or equal to 18.5 - then output u r underweight
2. weight / height ^ 2 is less than or equal to 25.0 - then output u r normal
3. weight / height ^ 2 is less than or equal to 30.0 - then output u r fat
4. if not matching with all the other cases above - then output u r a whale

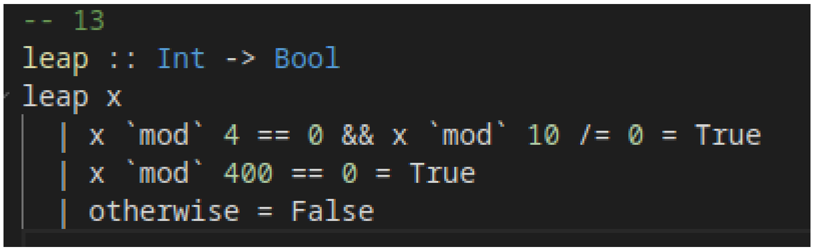


```haskell
-- 12
checkEligible :: RealFloat (a) => a -> a -> String
checkEligible w h
    | bmi <= 18.5 = "u r Overweight"
    | bmi <= 25.0 = "u r normal"
    | bmi <= 30.0 = "u r overweight"
    | otherwise = "u r a whale"
  where bmi = w/h^2
```

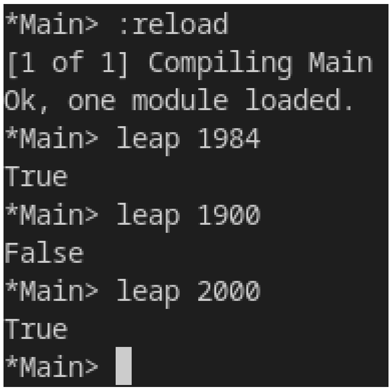


```
*Main> :reload
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> checkEligible 81 166
"u r Overweight"
*Main> 
```

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.

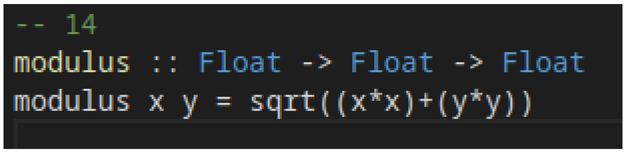


```
-- 13
leap :: Int -> Bool
leap x
  | x `mod` 4 == 0 && x `mod` 10 /= 0 = True
  | x `mod` 400 == 0 = True
  | otherwise = False
```

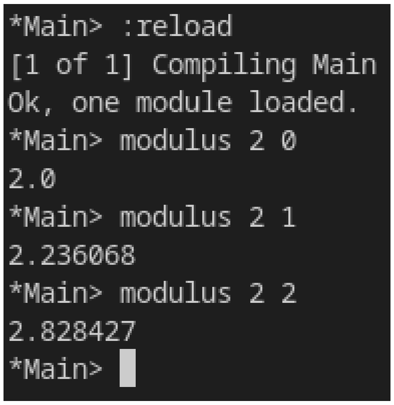


```
*Main> :reload
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> leap 1984
True
*Main> leap 1900
False
*Main> leap 2000
True
*Main>
```

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

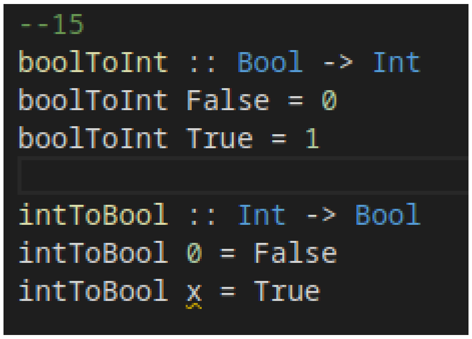


```
-- 14
modulus :: Float -> Float -> Float
modulus x y = sqrt((x*x)+(y*y))
```



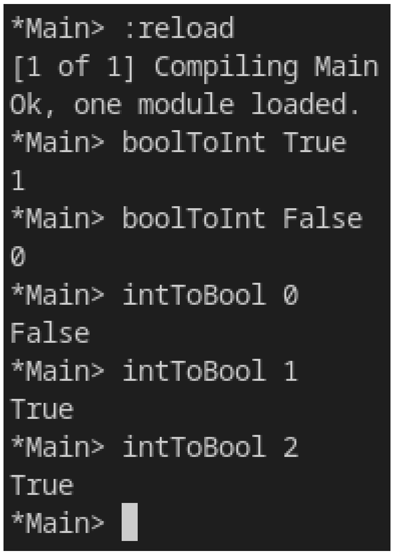
```
*Main> :reload
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> modulus 2 0
2.0
*Main> modulus 2 1
2.236068
*Main> modulus 2 2
2.828427
*Main>
```

15. Define two conversion functions **boolToInt** from boolean values to integer numbers and **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”.



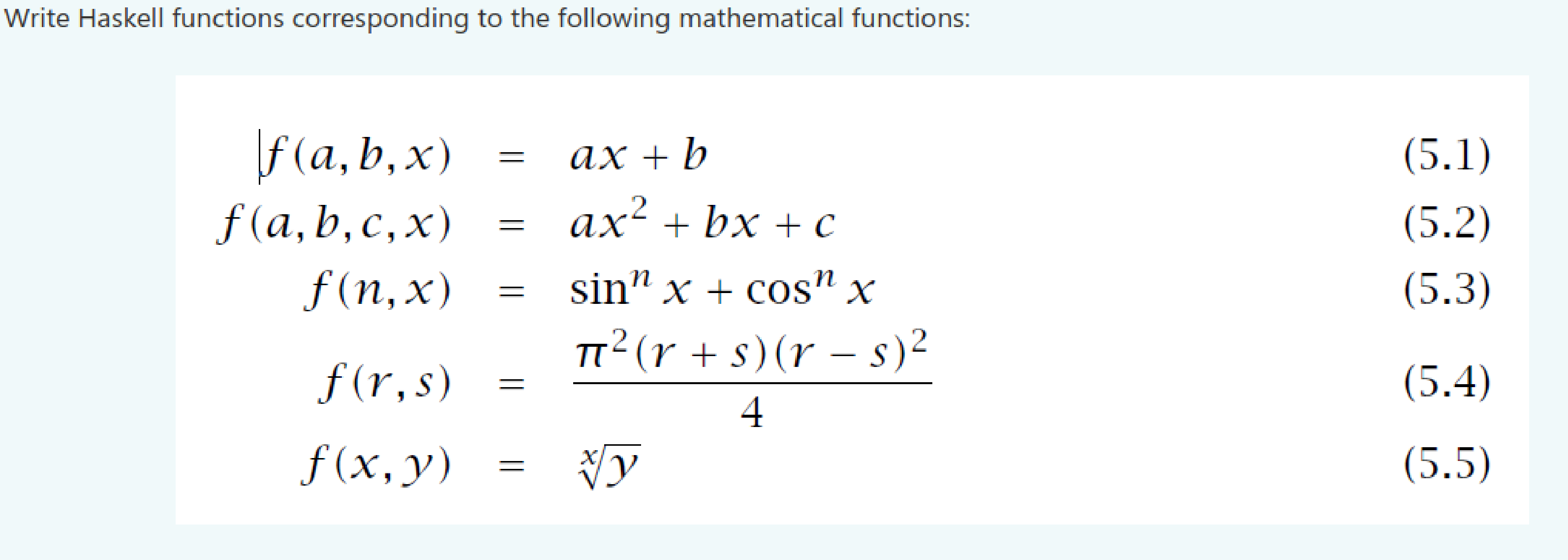
```
--15
boolToInt :: Bool -> Int
boolToInt False = 0
boolToInt True = 1

intToBool :: Int -> Bool
intToBool 0 = False
intToBool x = True
```



```
*Main> :reload
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> boolToInt True
1
*Main> boolToInt False
0
*Main> intToBool 0
False
*Main> intToBool 1
True
*Main> intToBool 2
True
*Main>
```

16.



Write Haskell functions corresponding to the following mathematical functions:

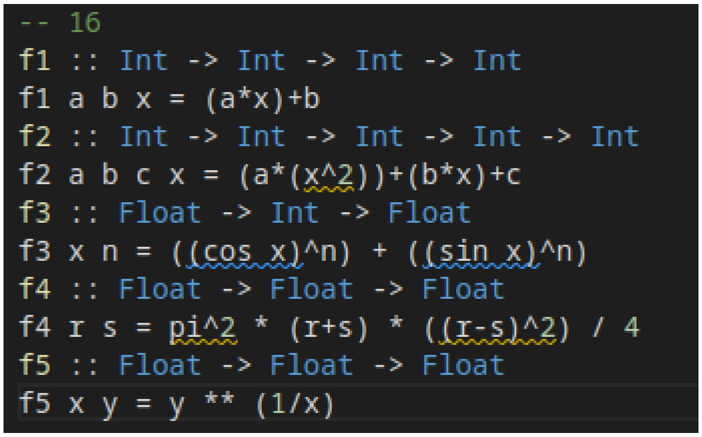
$$f(a,b,x) = ax + b \tag{5.1}$$

$$f(a,b,c,x) = ax^2 + bx + c \tag{5.2}$$

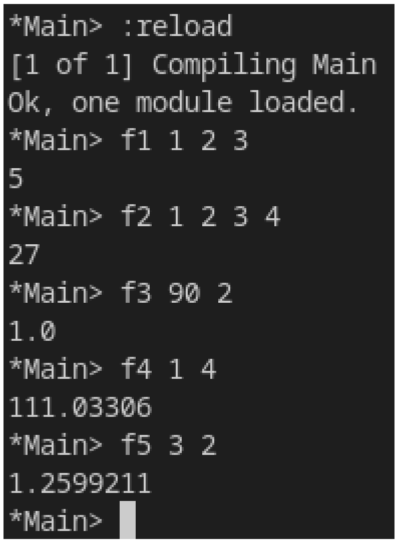
$$f(n,x) = \sin^n x + \cos^n x \tag{5.3}$$

$$f(r,s) = \frac{\pi^2(r+s)(r-s)^2}{4} \tag{5.4}$$

$$f(x,y) = \sqrt[x]{y} \tag{5.5}$$

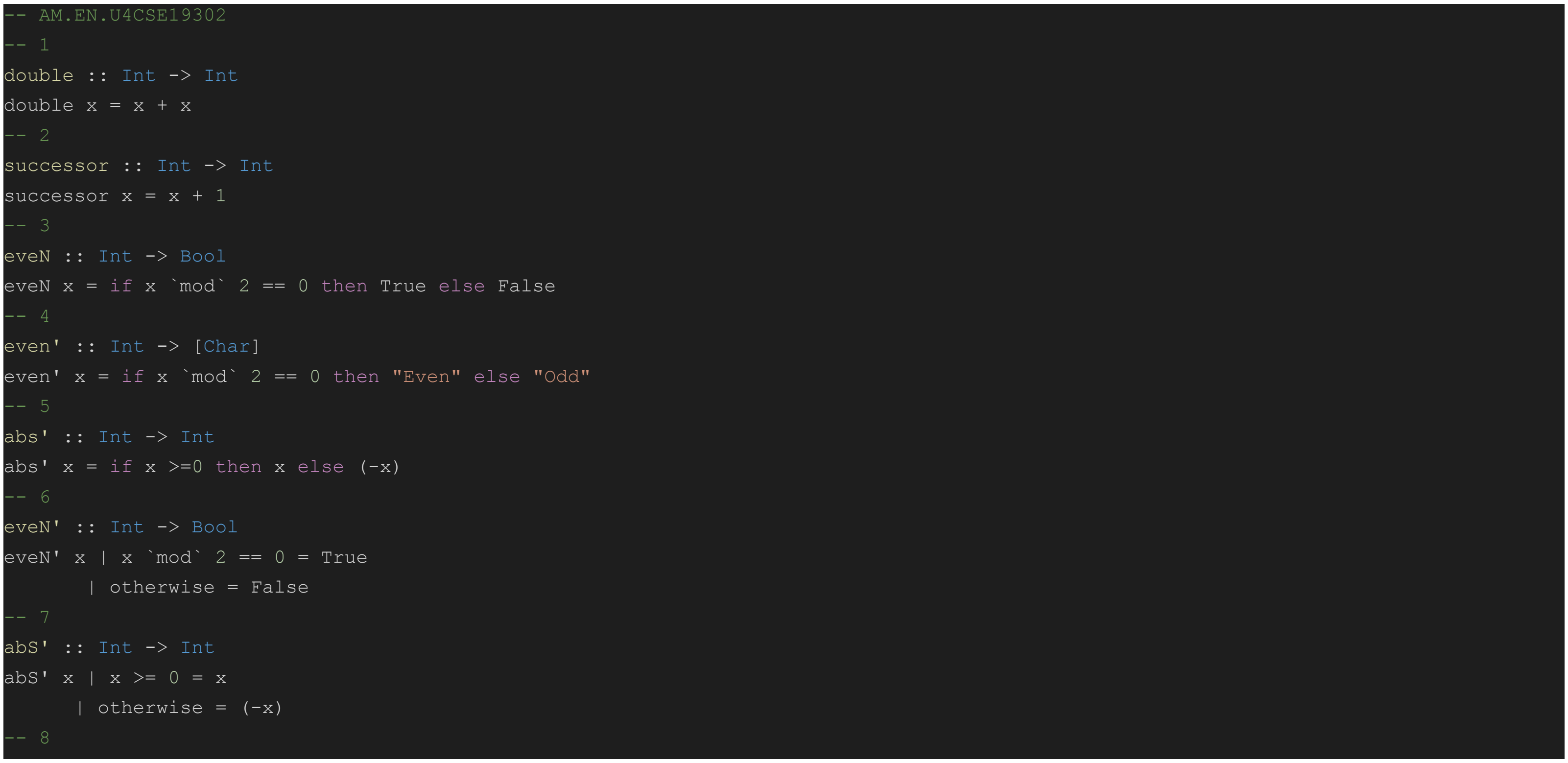


```
-- 16
f1 :: Int -> Int -> Int -> Int
f1 a b x = (a*x)+b
f2 :: Int -> Int -> Int -> Int -> Int
f2 a b c x = (a*(x^2))+(b*x)+c
f3 :: Float -> Int -> Float
f3 x n = ((cos x)^n) + ((sin x)^n)
f4 :: Float -> Float -> Float
f4 r s = pi^2 * (r+s) * ((r-s)^2) / 4
f5 :: Float -> Float -> Float
f5 x y = y ** (1/x)
```

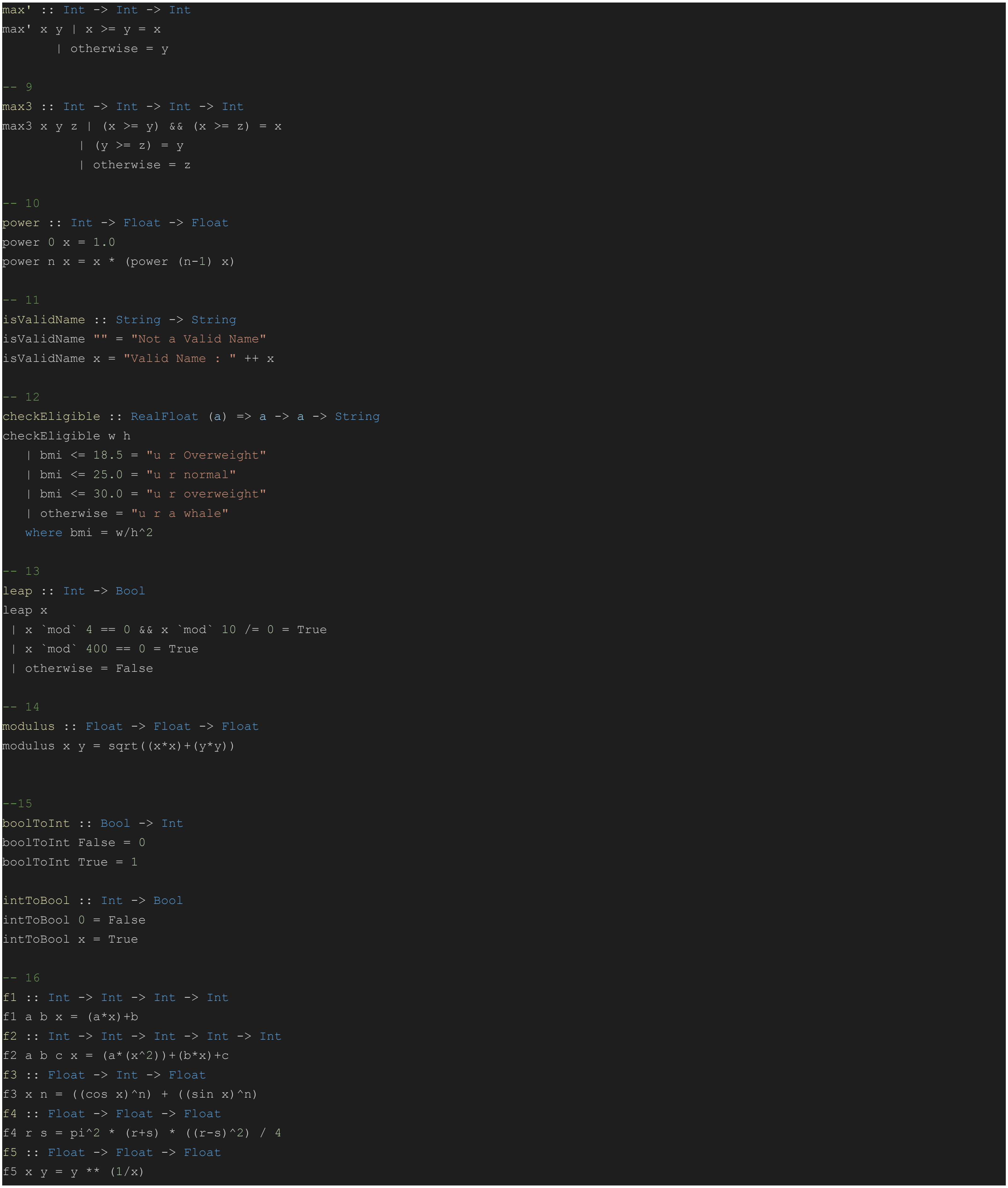


```
*Main> :reload
[1 of 1] Compiling Main
Ok, one module loaded.
*Main> f1 1 2 3
5
*Main> f2 1 2 3 4
27
*Main> f3 90 2
1.0
*Main> f4 1 4
111.03306
*Main> f5 3 2
1.2599211
*Main>
```

Script



```
-- AM.EN.U4CSE19302
-- 1
double :: Int -> Int
double x = x + x
-- 2
successor :: Int -> Int
successor x = x + 1
-- 3
eveN :: Int -> Bool
eveN x = if x `mod` 2 == 0 then True else False
-- 4
even' :: Int -> [Char]
even' x = if x `mod` 2 == 0 then "Even" else "Odd"
-- 5
abs' :: Int -> Int
abs' x = if x >=0 then x else (-x)
-- 6
eveN' :: Int -> Bool
eveN' x | x `mod` 2 == 0 = True
        | otherwise = False
-- 7
abS' :: Int -> Int
abS' x | x >= 0 = x
       | otherwise = (-x)
-- 8
```



```haskell
max' :: Int -> Int -> Int
max' x y | x >= y = x
       | otherwise = y

-- 9
max3 :: Int -> Int -> Int -> Int
max3 x y z | (x >= y) && (x >= z) = x
          | (y >= z) = y
          | otherwise = z

-- 10
power :: Int -> Float -> Float
power 0 x = 1.0
power n x = x * (power (n-1) x)

-- 11
isValidName :: String -> String
isValidName "" = "Not a Valid Name"
isValidName x = "Valid Name : " ++ x

-- 12
checkEligible :: RealFloat (a) => a -> a -> String
checkEligible w h
   | bmi <= 18.5 = "u r Overweight"
   | bmi <= 25.0 = "u r normal"
   | bmi <= 30.0 = "u r overweight"
   | otherwise = "u r a whale"
   where bmi = w/h^2

-- 13
leap :: Int -> Bool
leap x
 | x `mod` 4 == 0 && x `mod` 10 /= 0 = True
 | x `mod` 400 == 0 = True
 | otherwise = False

-- 14
modulus :: Float -> Float -> Float
modulus x y = sqrt((x*x)+(y*y))



--15
boolToInt :: Bool -> Int
boolToInt False = 0
boolToInt True = 1

intToBool :: Int -> Bool
intToBool 0 = False
intToBool x = True

-- 16
f1 :: Int -> Int -> Int -> Int
f1 a b x = (a*x)+b
f2 :: Int -> Int -> Int -> Int -> Int
f2 a b c x = (a*(x^2))+(b*x)+c
f3 :: Float -> Int -> Float
f3 x n = ((cos x)^n) + ((sin x)^n)
f4 :: Float -> Float -> Float
f4 r s = pi^2 * (r+s) * ((r-s)^2) / 4
f5 :: Float -> Float -> Float
f5 x y = y ** (1/x)
```