19CSE313

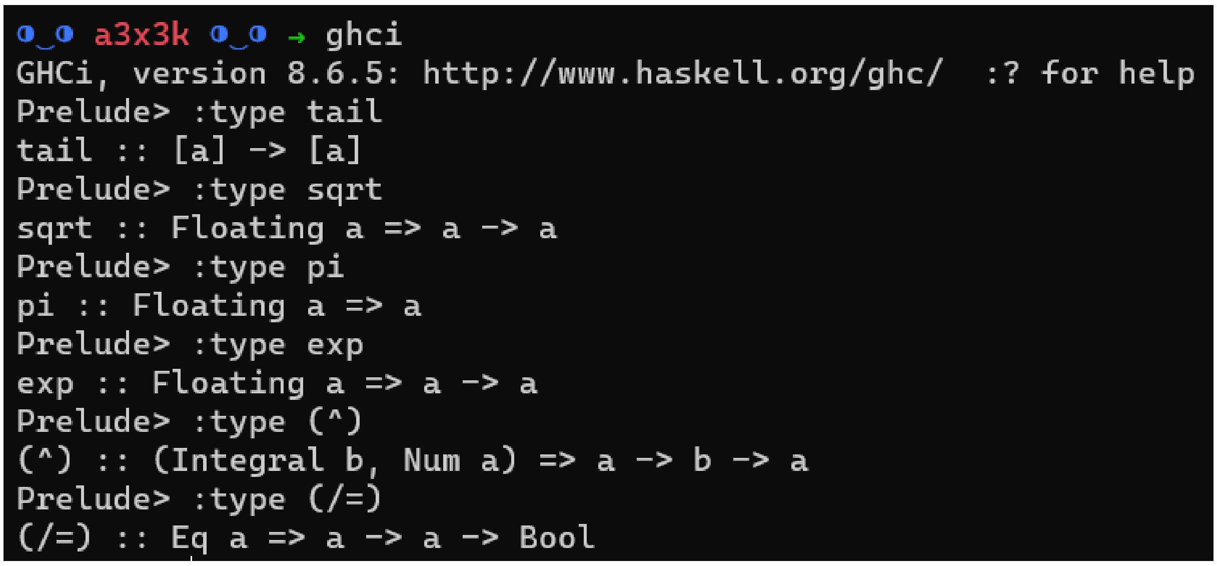
Principles of Programming Languages

Lab 2

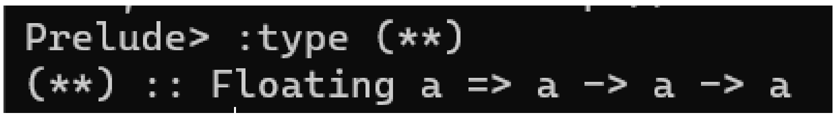
S Abhishek

AM.EN.U4CSE19147

1 - What is the type of the following functions? tail, sqrt, pi, exp, (ˆ), (/=) and (**)? How can you query the interpreter for the type of an expression and how can you explicitly specify the types of functions in your program?

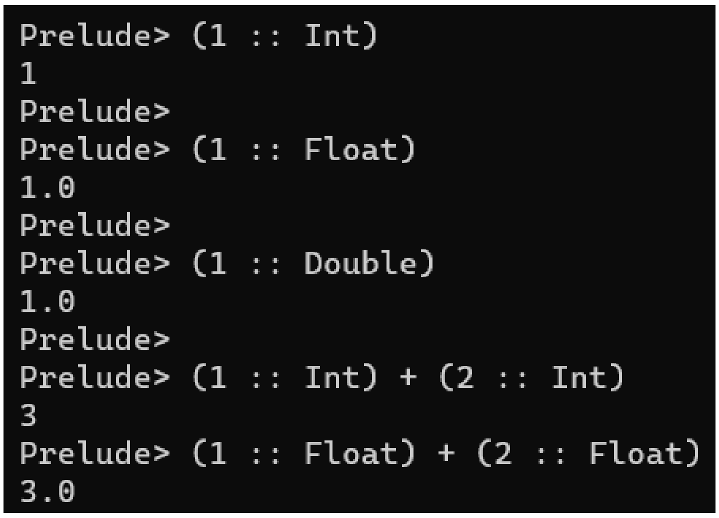


```
o_o a3x3k o_o → ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :type tail
tail :: [a] -> [a]
Prelude> :type sqrt
sqrt :: Floating a => a -> a
Prelude> :type pi
pi :: Floating a => a
Prelude> :type exp
exp :: Floating a => a -> a
Prelude> :type (^)
(^) :: (Integral b, Num a) => a -> b -> a
Prelude> :type (/=)
(/=) :: Eq a => a -> a -> Bool
```



```
Prelude> :type (**)
(**) :: Floating a => a -> a -> a
```

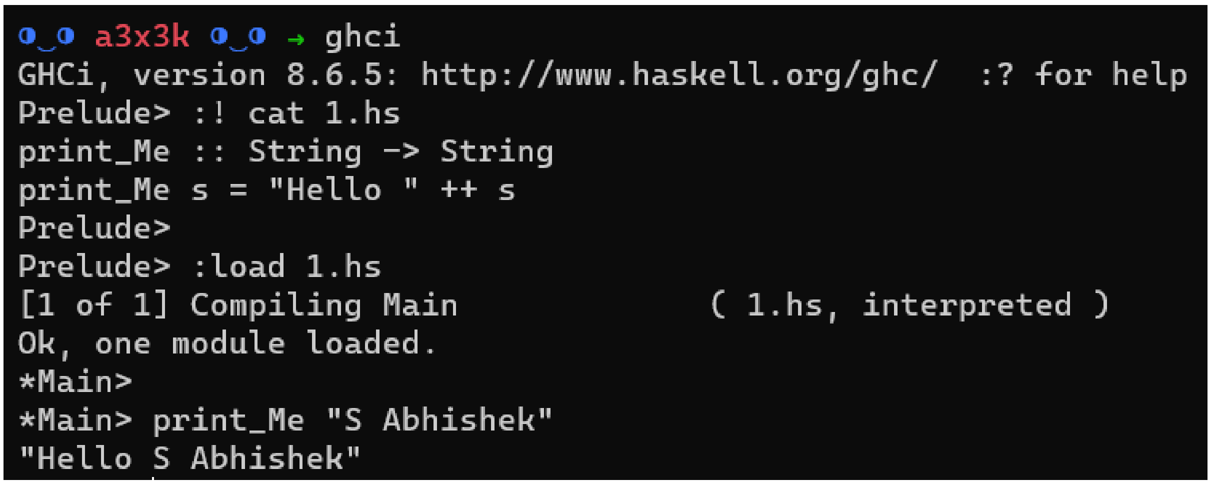
- We shall query the interpreter for the type of an expression using,
    - :: <Type> Expression.



```
Prelude> (1 :: Int)
1
Prelude>
Prelude> (1 :: Float)
1.0
Prelude>
Prelude> (1 :: Double)
1.0
Prelude>
Prelude> (1 :: Int) + (2 :: Int)
3
Prelude> (1 :: Float) + (2 :: Float)
3.0
```

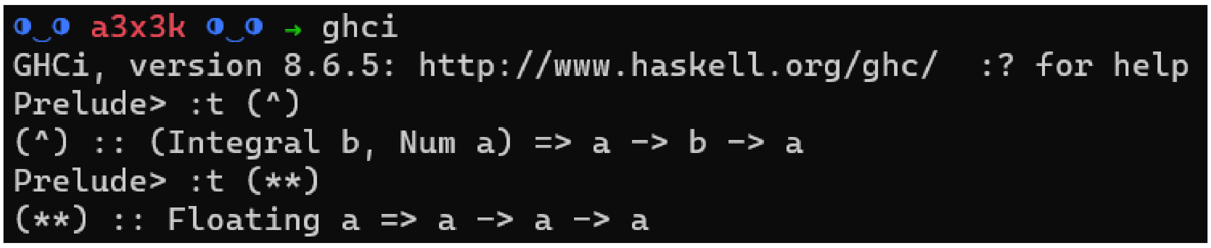
We shall explicitly specify the types of functions using,

- Function Name :: Argument Type -> Return Type

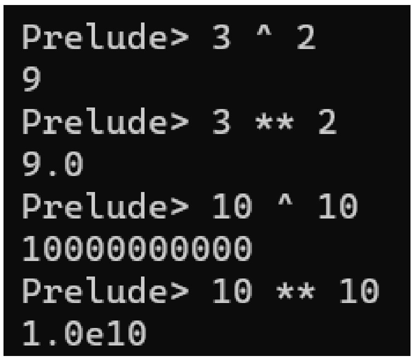


```
◑‿◑ a3x3k ◑‿◑ → ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :! cat 1.hs
print_Me :: String -> String
print_Me s = "Hello " ++ s
Prelude>
Prelude> :load 1.hs
[1 of 1] Compiling Main             ( 1.hs, interpreted )
Ok, one module loaded.
*Main>
*Main> print_Me "S Abhishek"
"Hello S Abhishek"
```

Find the difference between (^) and (**) function by looking into their type.



```
o_o a3x3k o_o → ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :t (^)
(^) :: (Integral b, Num a) => a -> b -> a
Prelude> :t (**)
(**) :: Floating a => a -> a -> a
```



```
Prelude> 3 ^ 2
9
Prelude> 3 ** 2
9.0
Prelude> 10 ^ 10
10000000000
Prelude> 10 ** 10
1.0e10
```

(^) – Accepts the arguments of typeclass Integral which includes types such as Int and Integer and returns the result in the form of Num typeclass which includes all numeric types such as Int, Integer, Float and Double.

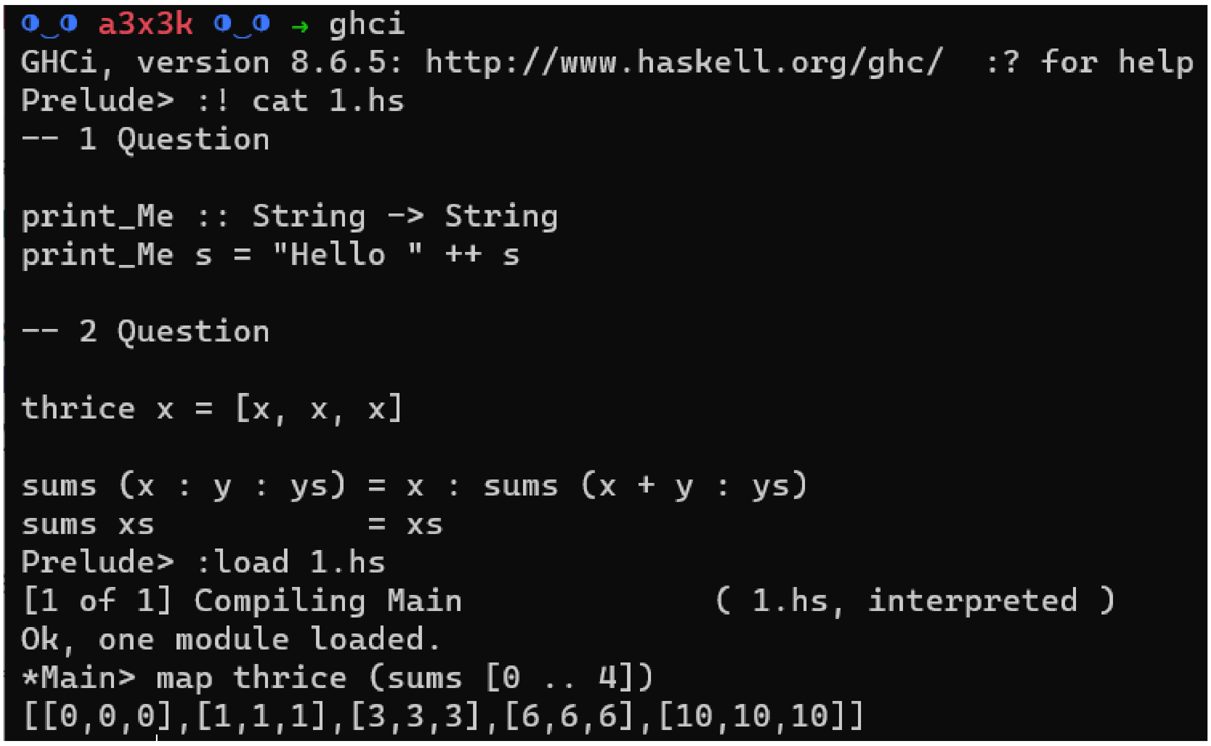
(**) – Accepts the arguments of type class Floating which includes types such as Float and Double and returns the result in the form of Floating typeclass which includes types such as Float and Double.

2 - Given the following definitions:

- thrice x = [x, x, x]
- sums (x : y : ys) = x : sums (x + y : ys)
- sums xs = xs

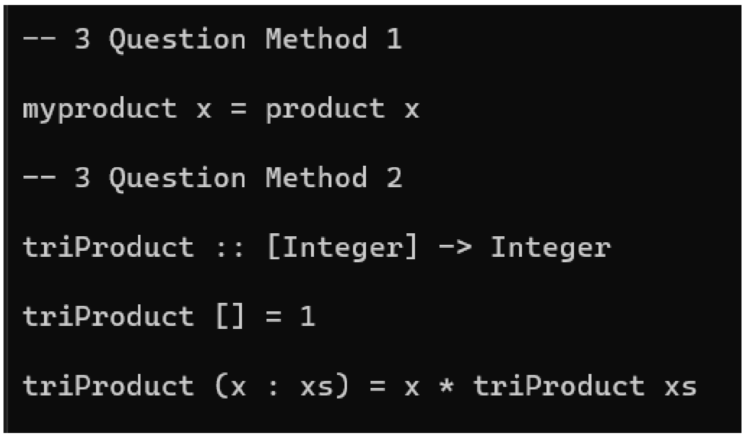
What does the following expression evaluate to?

- map thrice (sums [0 .. 4])

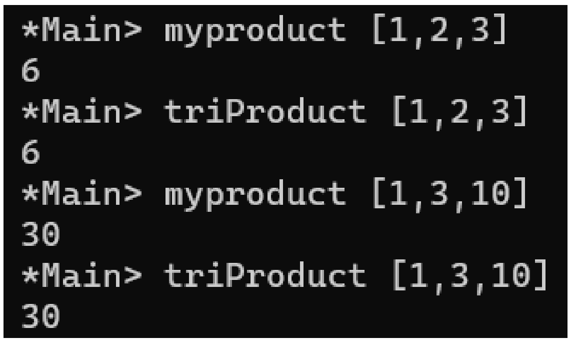


```
◑‿◑ a3x3k ◑‿◑ → ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :! cat 1.hs
-- 1 Question

print_Me :: String -> String
print_Me s = "Hello " ++ s

-- 2 Question

thrice x = [x, x, x]

sums (x : y : ys) = x : sums (x + y : ys)
sums xs           = xs
Prelude> :load 1.hs
[1 of 1] Compiling Main             ( 1.hs, interpreted )
Ok, one module loaded.
*Main> map thrice (sums [0 .. 4])
[[0,0,0],[1,1,1],[3,3,3],[6,6,6],[10,10,10]]
```

3 - Define a function myProduct that produces the product of a list of numbers, and show using your definition that myProduct [2,3,4] = 24. [Use product build-in function]



```
-- 3 Question Method 1

myproduct x = product x

-- 3 Question Method 2

triProduct :: [Integer] -> Integer

triProduct [] = 1

triProduct (x : xs) = x * triProduct xs
```



```
*Main> myproduct [1,2,3]
6
*Main> triProduct [1,2,3]
6
*Main> myproduct [1,3,10]
30
*Main> triProduct [1,3,10]
30
```

4 - Record the types of the following values

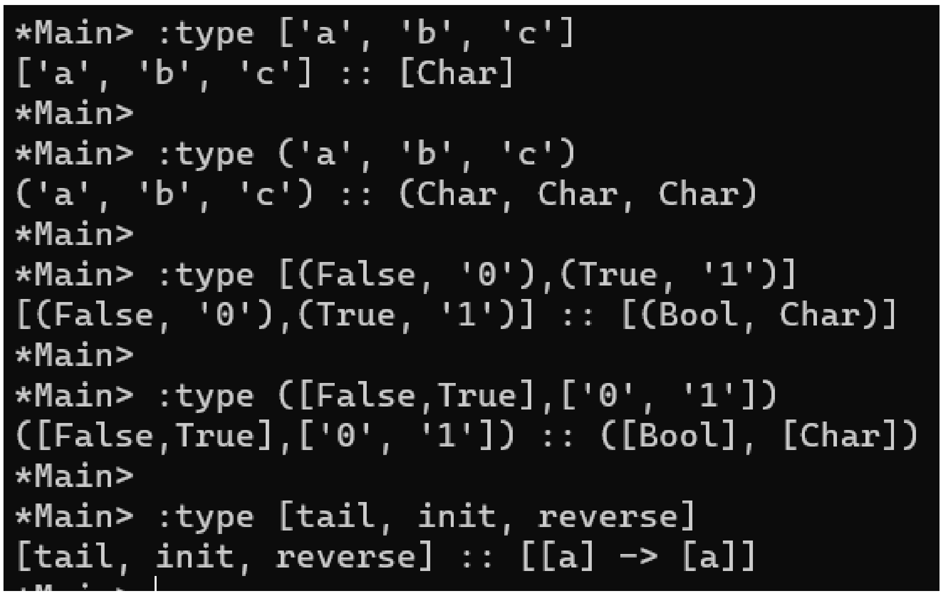
[‘a’, ‘b’, ‘c’]

(‘a’, ‘b’, ‘c’)

[(False, ‘0’),(True, ‘1’)]

([False,True],[‘0’, ‘1’])

[tail, init, reverse]



```
*Main> :type ['a', 'b', 'c']
['a', 'b', 'c'] :: [Char]
*Main>
*Main> :type ('a', 'b', 'c')
('a', 'b', 'c') :: (Char, Char, Char)
*Main>
*Main> :type [(False, '0'),(True, '1')]
[(False, '0'),(True, '1')] :: [(Bool, Char)]
*Main>
*Main> :type ([False,True],['0', '1'])
([False,True],['0', '1']) :: ([Bool], [Char])
*Main>
*Main> :type [tail, init, reverse]
[tail, init, reverse] :: [[a] -> [a]]
```

5 - Record down definitions that have the following types; it does not matter what the definitions actually do as long as they are type correct.

bools :: [Bool] - bools is a list of boolean value.

nums :: Int - nums is a variable of type Int.

add :: Int -> Int -> Int -> Int – add is a function which takes 3 arguments of type Int and returns a value of type Int.

copy :: a -> (a, a)

- **copy a = (a, a)**

copy is a function of which takes a single argument and returns the list of 2 elements which have the same value as the argument passed. There is no specific type declaration and thus the compiler can infer the type by itself and returns the value of same type.

apply :: (a -> b) -> a -> b

- **apply f x = f x**

apply function takes a function, and an input to that function, and applies the function. A generic function f has type a -> b. Second argument x is also input to f. Output apply f x is the same as f x.

6 - Record the types of the following functions

second xs = head (tail xs)

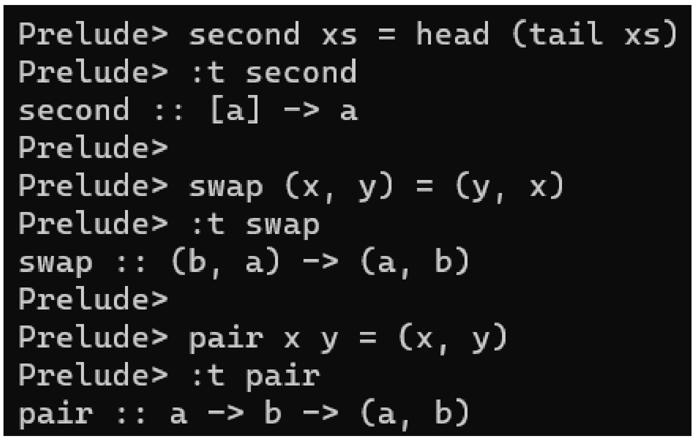
swap (x, y) = (y, x)

pair x y = (x, y)

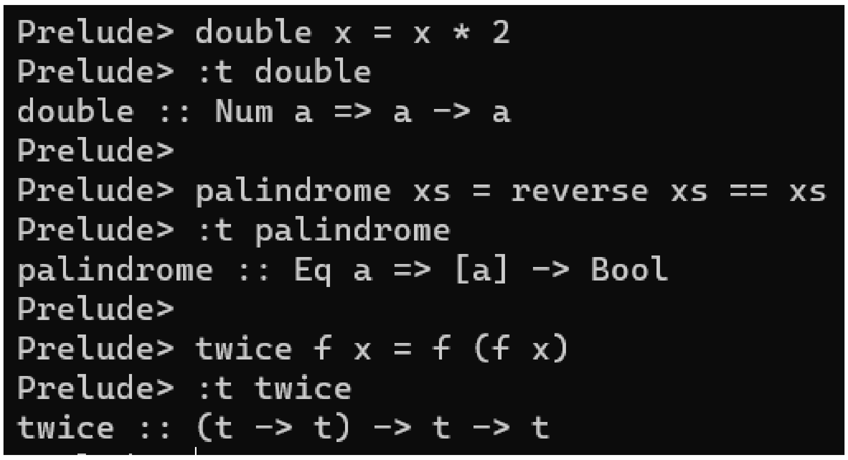
double x = x * 2

palindrome xs = reverse xs == xs

twice f x = f (f x)



```
Prelude> second xs = head (tail xs)
Prelude> :t second
second :: [a] -> a
Prelude>
Prelude> swap (x, y) = (y, x)
Prelude> :t swap
swap :: (b, a) -> (a, b)
Prelude>
Prelude> pair x y = (x, y)
Prelude> :t pair
pair :: a -> b -> (a, b)
```

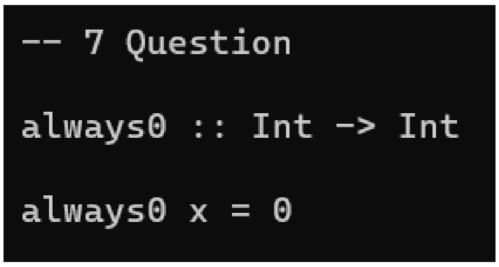


```
Prelude> double x = x * 2
Prelude> :t double
double :: Num a => a -> a
Prelude>
Prelude> palindrome xs = reverse xs == xs
Prelude> :t palindrome
palindrome :: Eq a => [a] -> Bool
Prelude>
Prelude> twice f x = f (f x)
Prelude> :t twice
twice :: (t -> t) -> t -> t
```

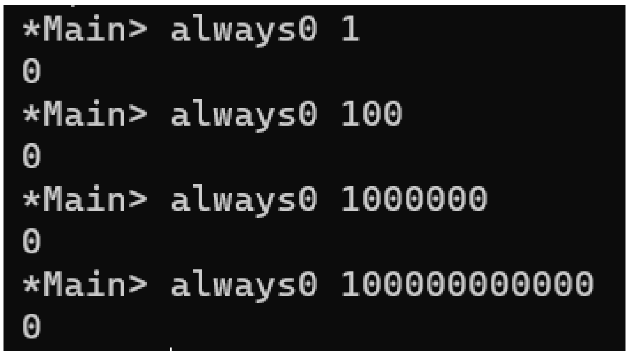
7 - Write a function named always0 ::Int → Int. The return value should always just be 0.

always0 :: Int -> Int

always0 x = 0



```
-- 7 Question

always0 :: Int -> Int

always0 x = 0
```

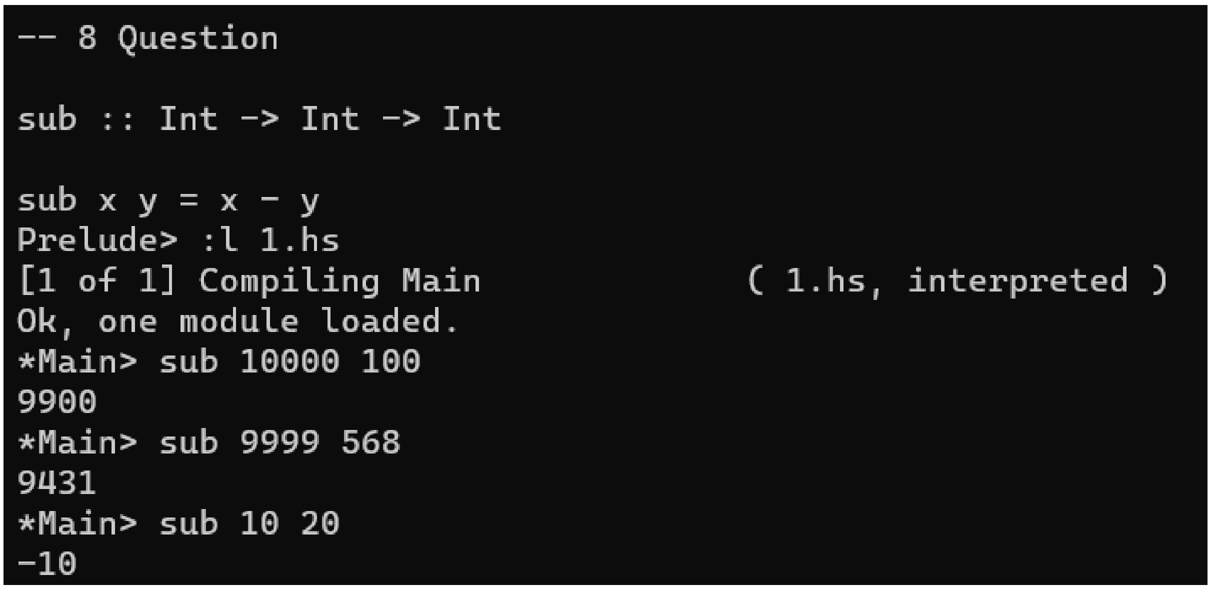


```
*Main> always0 1
0
*Main> always0 100
0
*Main> always0 1000000
0
*Main> always0 100000000000
0
```

8 - Write a function subtract :: Int → Int → Int that takes two numbers (that is, Ints) and subtracts them.

sub :: Int -> Int -> Int

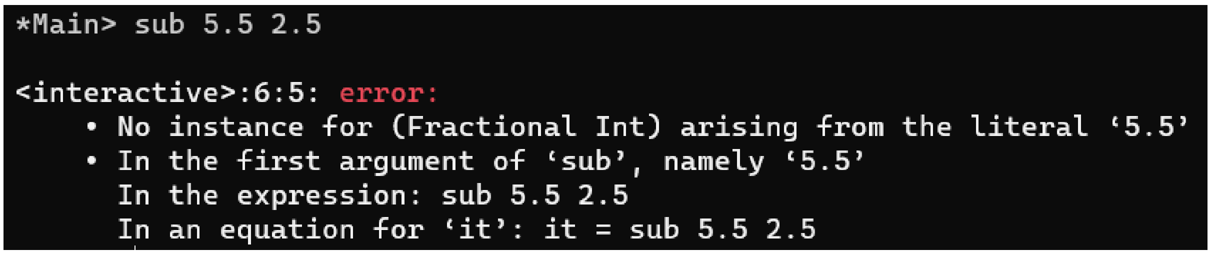
sub x y = x - y



```
-- 8 Question

sub :: Int -> Int -> Int

sub x y = x - y
Prelude> :l 1.hs
[1 of 1] Compiling Main             ( 1.hs, interpreted )
Ok, one module loaded.
*Main> sub 10000 100
9900
*Main> sub 9999 568
9431
*Main> sub 10 20
-10
```

A - Will the above function work for Float type arguments? If not, rewrite the function.

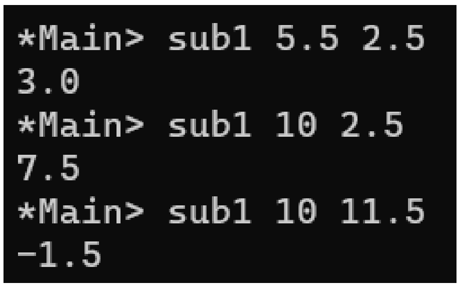
- The above function will not work with Float type arguments.



```
*Main> sub 5.5 2.5

<interactive>:6:5: error:
    • No instance for (Fractional Int) arising from the literal ‘5.5’
    • In the first argument of ‘sub’, namely ‘5.5’
      In the expression: sub 5.5 2.5
      In an equation for ‘it’: it = sub 5.5 2.5
```

sub1 :: Float -> Float -> Float

sub1 x y = x - y

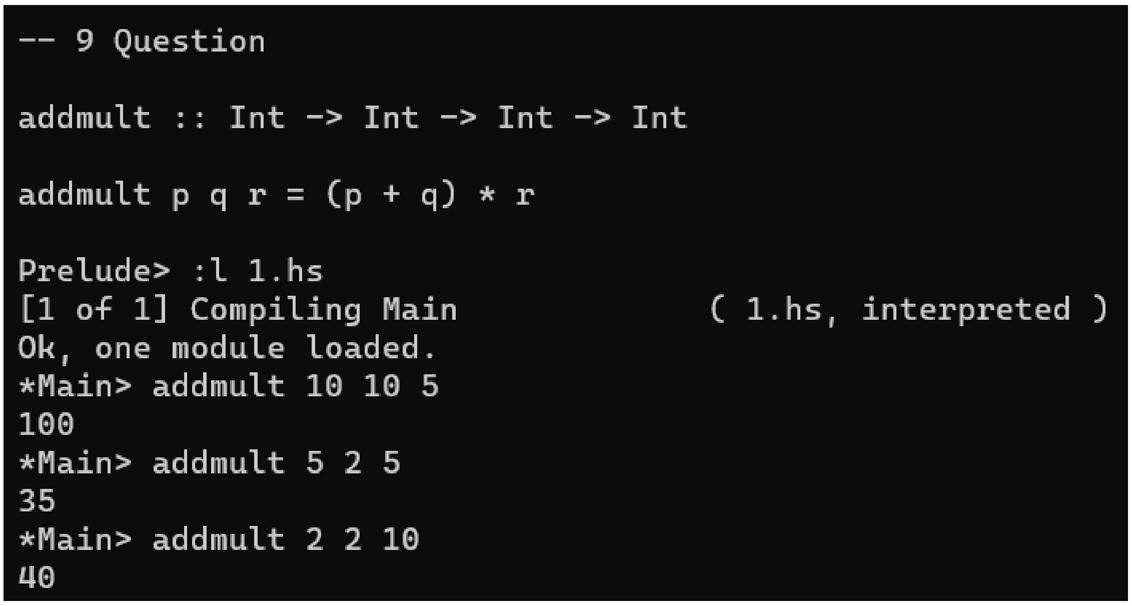


```
*Main> sub1 5.5 2.5
3.0
*Main> sub1 10 2.5
7.5
*Main> sub1 10 11.5
-1.5
```

9 - Write a function addmult that takes three numbers. Let's call them p, q, and r. addmult should add p and q together and then multiply the result by r.

addmult :: Int -> Int -> Int -> Int

addmult p q r = (p + q) * r



```
-- 9 Question

addmult :: Int -> Int -> Int -> Int

addmult p q r = (p + q) * r

Prelude> :l 1.hs
[1 of 1] Compiling Main             ( 1.hs, interpreted )
Ok, one module loaded.
*Main> addmult 10 10 5
100
*Main> addmult 5 2 5
35
*Main> addmult 2 2 10
40
```

Thankyou!!