Exercises
Functions

Exercise 1:

Write a function named add1 that takes an Int and returns an Int that is one greater than its input. For example, if we compute add1 5, we should get 6. If you want to write a type signature for add1, it would be

```
add1 :: Int -> Int
```

Solution 1:

```
add1 :: Int -> Int
add1 x = x+1
```

Exercise 2:

Write a function:

```
   always0 :: Int-> Int
```

The return value should always just be 0.

Solution 2:

```
always0 :: Int -> Int
always0 _ = 0
```

Note: This function could also be written (with a different type, i.e. no input argument) as

```
always0 :: Int
always0 = 0
```

Exercise 3:
Write a function:

```
subtract' :: Int-> Int-> Int
```

that takes two numbers (that is, Ints) and subtracts them.

Solution 3:

```
    subtract' :: Int -> Int -> Int
    subtract' x y = x-y
```

Exercise 4:
Write a function:

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

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.

Solution 4:

```
addmult :: Int -> Int -> Int -> Int
addmult x y z = (x+y) * z
```

Exercise 5:
Write a function

```
greaterThan0 :: Int -> String
```

That returns the String "Yes!" if the number is greater than 0, and "No!" otherwise.

Solution 5:

```
greaterThan0 :: Int -> String
greaterThan0 n = if n>0 then "Yes" else "No!"
```

Exercise 6:
Look at the function

```
pushOut :: Int -> Int
```

that takes a number and returns the number that is one step further from 0. That is,

- pushOut 3 is 4,
- pushOut (-10) is (-11), and
- pushOut 0 is 0.

That last one is because we don't know which direction to go!.

Write this function using

1. if .. then .. else (call this pushOut)
2. guarded equations (call this pushOut')

Remember that, in Haskell, have to put parentheses around negative numbers

Solution 6:

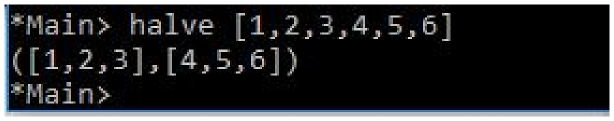
1.
```
pushOut :: Int -> Int
pushOut n = if n > 0 then n+1
            else if n < 0 then n-1
            else 0
```

2.
```
pushOut' :: Int -> Int
pushOut' n | n > 0    = n+1
           | n < 0    = n-1
           | otherwise = 0
```

Exercise 7:
Using library functions, define a function

```
halve :: [a] -> ([a], [a])
```

that splits an even-lengthed list into two halves. For example:



```
*Main> halve [1,2,3,4,5,6]
([1,2,3],[4,5,6])
*Main>
```

Solution 7:

```
halve :: [a] -> ([a], [a])
halve xs = (take half xs, drop half xs) where
      half = length xs ‘div‘ 2
```

Exercise 8:
Define a function ***third***

```
third :: [a] -> a
```

that returns the third element in a list that contains at least this many elements using:

1. ***head*** and ***tail***;
2. list indexing ***!!***;
3. pattern matching.

Solution 8:

```
third :: [a] -> a
third xs = head (tail (tail xs))

third’ :: [a] -> a
third’ xs = xs !! 2

third’’ :: [a] -> a
third’’ (_:_:x:_) = x
```

Exercise 9:

Consider a function ***safetail*** that behaves in the same way as ***tail***, except that ***safetail*** maps the empty list to the empty list, whereas tail gives an error in this case. Define ***safetail*** using:

1. a conditional expression;
2. guarded equations;
3. pattern matching.

Hint: the library function ***null*** :: [a] − > Bool can be used to test if a list is empty.

Solution 9:

```
safetail :: [a] -> [a]
safetail xs  =  if  null xs  then  [] else tail xs

safetail' :: [a] -> [a]
safetail' xs | null xs = []
             | otherwise = tail xs

safetail'' ::[a] -> [a]
safetail'' [] = []
safetail'' (_:xs) = xs
```

Exercise 10:

In a similar way to ***&&*** in this section's slides, show how the disjunction operator || can be defined in three different ways using pattern matching.(Call it **myOr)** **Solution 10:**

```
myor:: Bool -> Bool -> Bool
False `myor` False = False
_ `myor` _         = True

myor':: Bool -> Bool -> Bool
False `myor'` b = b
True `myor'` _ = True
```

Exercise 11:
Given the function with the following type :

```
lucky :: Integral a => a-> String
```

Write the function definition that returns the following strings given the following inputs:

1. When input is 7, the output is the String "Lucky you.. Proceed directly to buy a lottery ticket.".
2. When input is 13, the output is the String "You, sadly are quite unlucky. Do not, under any circumstances, invest money today."
3. For any other input, the output is the String "Mmmm.... Can't really say...."

Solution 11:

```
-- (from Learn you a good Haskell)
lucky :: Integral a => a-> String
lucky 7 = "Lucky you.. Proceed directly to buy a lottery ticket."
lucky 13 = "You, sadly are quite unlucky. Do not, under any
    circumstances, invest money today "
lucky _ = "Mmmm.... Can't really say...."
```

Exercise 12:

Given the two (Prelude) functions ***fst*** and ***snd*** who return the first and second element of a 2-tuple respectively as in :

```
fst:: (a,b) -> a
fst(x, _) = x
snd :: (a,b) -> b
snd(_, y) = y
```

Write similar functions - first, second and third who return the first, second and third element of a three tuple.

```
second (2,4,'e') = 4
```

Solution 12:

```
fst':: (a,b,c) -> a
fst'(x, _, _) = x
snd' :: (a,b,c) -> b
snd'(_, y,_) = y
thrd' :: (a,b,c) -> c
thrd'(_, _,z) = z
```

The next two exercises are from Graham Hutton's book "Programming in Haskell" (2nd Edition) and are optional.

Exercise 13:

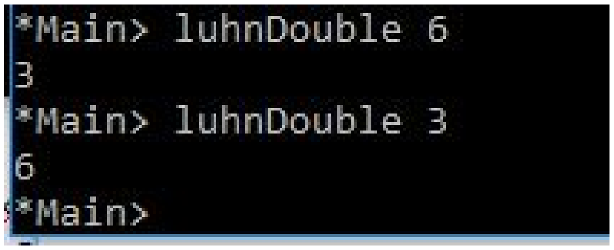
The ***Luhn Algorithm*** is used to check bank card numbers for simple errors such as mistyping a digit and proceeds as follows:

- consider each digit as a separate number;
- moving left, double every other number from the second last;
- subtract 9 from each number that is now greater than 9;
- add all the resulting numbers together;
- if the total is divisible by 10, the card number is valid.

Note that the rightmost digit is the check digit. Define a function

```
luhnDouble :: Int -> Int
```

that doubles the a digit and subtracts 9 if the number is greater than 9. For example

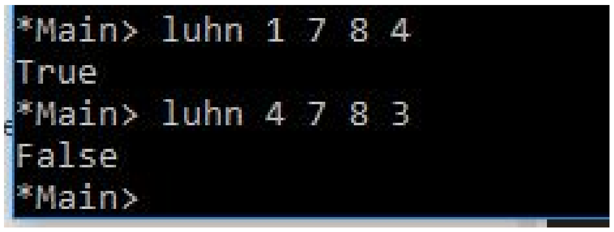


```
*Main> luhnDouble 6
3
*Main> luhnDouble 3
6
*Main>
```

Using ***luhnDouble*** and the integer remainder function ***mod***, define a function

```
luhn :: Int -> Int-> Int-> Int-> Bool
```

that decides if a four digit number is valid. For example:



```
*Main> luhn 1 7 8 4
True
*Main> luhn 4 7 8 3
False
*Main>
```

Solution 13:
(From Hutton)

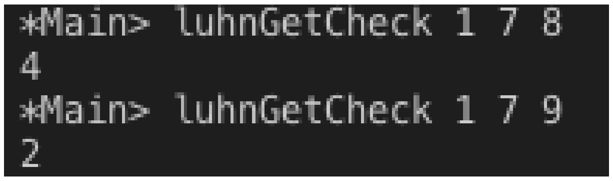
```haskell
luhnDouble :: Int -> Int
luhnDouble x = if (x*2> 9) then x*2-9 else x*2

luhn :: Int -> Int -> Int -> Int -> Bool
luhn a b c d = (a' + b + c'+ d ) `mod` 10 == 0 where
      a' = luhnDouble a
      c' = luhnDouble c
```

Exercise 14:
Using the same definition of the luhn algorithm and remembering that the rightmost digit is the check digit, write a function

```haskell
luhnGetCheck:: Int -> Int-> Int-> Int
```

that, given the leftmost three digits as per the previous example, calculates the check digit. For example



```
*Main> luhnGetCheck 1 7 8
4
*Main> luhnGetCheck 1 7 9
2
```

Solution 14:
(From Hutton)

```haskell
luhnGetCheck :: Int-> Int -> Int -> Int
luhnGetCheck a b c = (10 - (luhnSum `mod` 10)) `mod` 10 where
                    -- seccond `mod` to check for 10
      luhnSum = luhnDouble a + b + luhnDouble c
```

And alternative from Elijah Hartvigsen (thanks Elijah!)

```haskell
   getCheckLuhn x y z = if luhnNum == 10 then 0 else luhnNum where
     luhnNum = 10 - partialSum `mod` 10
     partialSum = luhnDouble x + y + luhnDouble z
```