

# Topic 04 - Tutorial Sheet-03

## 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
```

---

### Exercise 2:

Write a function:

---

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

---

The return value should always just be 0.

### Exercise 3:

Write a function:

---

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

---

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

### 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`.

### Exercise 5:

Write a function

---

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

---

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

**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

**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>
```

**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.

**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.

**Exercise 10:**

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

**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..."

**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
```

---

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.

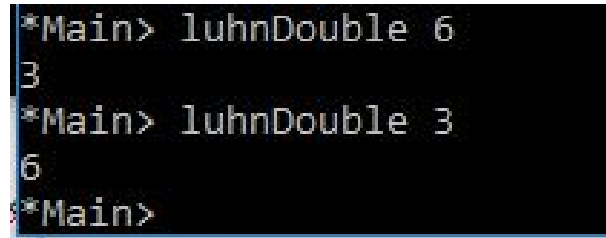
Note that the rightmost digit is the check digit. Define a function

---

```
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>
```

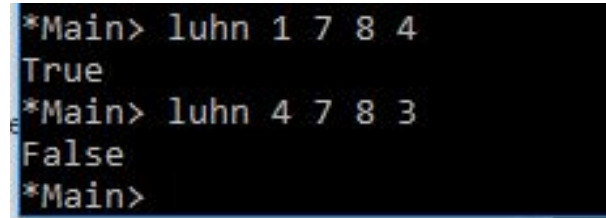
Using *luhnDouble* and the integer remainder function *mod*, define a function

---

```
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>
```

**Exercise 14:**

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

---

```
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
```