

# Functional and Logic Programming

## Home Assignment 3

**Due: Saturday, 7.5.2019 - 23:55**

### Instructions

- Please create a source file called **hw3.hs** and put all the answers there.  
The file should start with a comment which contains your **full name** (in English) and **ID**  
  
*-- John Doe*  
*-- 654321987*
- Make sure the file **is valid** by loading it into GHCi.  
A valid file will load without any errors or warnings.
- If you need a function but you don't know how to implement it - just write it's signature (name and type) and put `undefined` in the function's body.  
That way you'll be able to load the file even though it contains references to undefined names.
- When writing a function - write both the **type** and the **body** of the function.
- Be sure to write functions with **exactly the specified name** (and **type signature** - if it is provided) for each exercise.  
You may create additional auxiliary/helper functions with whatever names and type signatures you wish.
- Try to write **small functions** which perform just **a single task**, and then **combine** them to create more complex functions.

## Exercises

1.

AES algorithm is a common encryption algorithm.

[https://en.wikipedia.org/wiki/Advanced\\_Encryption\\_Standard](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard)

In this exercise we will implement a simpler version of this algorithm.

We shall call it- simple-AES.

Our algorithm will be comprised of the following steps (which will be explained later):

1. Bytes substitution
2. Shift rows
3. Encrypt with Key

Assume that we are given a string which we want to encrypt as a matrix (a list of lists) such that each entry in the matrix is a single small letter (a-z).

a)

In this part we will implement step 1 – byte substitution.

We want to match each letter in our alphabet with another letter.

Implement a function

`byteSub :: Char -> Char`

Which receives a character and returns a letter the following way:

1. If the letter is 'a' then the function should return 'h'
2. If the letter is 'g' then the function should return 'b'
3. For any other letter the function should return the consecutive letter in the alphabet.  
For example:  
For the letter 'c', the letter 'd' should be returned.  
For the letter 'z', the letter 'a' should be returned.
4. For any character which is not a small letter (a-z) the function should return '0' (an arbitrary character)

b) Implement a function

`shiftIn :: Int -> [Char] -> [Char]`

Which receives an integer  $x$  and list of characters and shifts right the original list by  $x$  positions.

Note: You can assume that the integer will be a natural number.

For example:

`shiftIn 2 "abcdefg"="fgabcde"`

c) Implement a function:

`shiftRows :: [[Char]]->[[Char]]`

Which receives a matrix of characters and returns a matrix in which for each  $i^{th}$  row in the original matrix the row was shifted by  $i$  entries.  
(where in the first row  $i=0$ )

For example:

`shiftRows [['a','b','c'],['a','b','c'],['a','b','c']] = ["abc","cab","bca"]`

d)

Assume you are given an encryption function with the signature:

`Char->Char`

Which receives a letter between a-z and returns the encryption for given letter.

Implement a function:

`roundKey :: (Char->Char)->[[Char]]->[[Char]]`

Which receives an encryption function and a matrix of letters (you can assume all between a-z) and returns the matrix such that all of its entries are encrypted.

e)

Now we will apply all the previous functions except the first one.

Implement a function:

`simpleAES :: (Char->Char)->[[Char]]->String`

Which takes an encryption function and our original string after we applied the first step (byte substitution)- as a matrix of letters.

The function applies the last 2 steps of the algorithm (shift rows and encrypt with key) on the given matrix and finally returns the encrypted string.

For example:

`simpleAES byteSub [['a','b','c'],['d','e','f'],['g','h','i']] = "hcdgefijb"`

`simpleAES id [['a','b','c'],['d','e','f'],['g','h','i']] = "abcfdehig"`

2.

### Haskell: Recursive Data Structures – Cont.

Let's look at a "practical" example, and define a JSON data type. JSON (JavaScript Object Notation) is usually used to transmit a data over a network, e.g., from a web service to a browser.

JSON includes 4 basic type of **values**: strings, numbers, Booleans, and a special value called null.  
For example: "string", 2048, true, null.

JSON provides two "structures":

- **Array**: an ordered sequence of values.  
E.g.: ["hello", 94.3, false, null]
- **Object**: unordered collection of name/value pairs. The name in an object is a string.  
E.g.: {"name": "Recha Freier", "born": 1892, "children": ["Shalhevet", "Ammud", "Zerem", "Maayan"]}

```
Data JValue = JStr String | JNum Double | JBool Bool | JNull
            | JObject [(String, JValue)] | JArray [JValue]
```

JStr, JNum, JBool, etc. are constructors. JNull is an empty constructor. Note the recursive definition in JObject and JArray.

For the JSON data type:

```
data JVal = JStr String | JNum Double | JBool Bool | JNull | JObject [(String, JVal)] |
          JArray [JVal]
```

Define the function:

```
jsonToString :: JVal -> String
```

Which receives a JVal and outputs a String representing the given object.

#### Examples:

```
jsonToString (JStr "hi") = "hi"
```

```
jsonToString JNull = "null"
```

```
jsonToString (JNum 3) = "3.0"
```

```
jsonToString (JBool True) = "true"
```

```
jsonToString (JBool False) = "false"
```

```
jsonToString (JArray [JObject [("Name", JStr "Jon")]]) = "[{Name: Jon}]"
```

```
jsonToString (JObject [("name", JStr "Stevie Ray Vaughan"), ("Genre", JStr "Blues"), ("Guitar
Type", JStr "Fender Stratocaster"), ("Revival of blues", JBool True)])
```

```
"{name: Stevie Ray Vaughan, Genre: Blues, Guitar Type: Fender
Stratocaster, Revival of blues: true}"
```

```
jsonToString (JObject [("name", JStr "Jon"), ("Weight", JNum 72.3), ("cars", JArray [JStr "BMW",
JStr "Ford", JStr "Fiat"])])=
```

```
"{name: Jon, Weight: 72.3, cars: [BMW,Ford,Fiat]}"
```

```
jsonToString (JObject [("name", JStr "Bob"), ("Weight", JNum 23.1), ("cars", JObject [("car1", JStr
"Toyota"), ("car2", JStr "BMW"), ("car", JStr "Fiat")])])=
```

```
"{name: Bob, Weight: 23.1, cars: {car1: Toyota, car2: BMW, car: Fiat}}"
```

3. Implement the following functions with one line of code:

a) `myReverse :: [a] -> [a]`

Which takes a list and returns the reversed list.

NOTE: `myReverse = reverse`  
will not get any points.

**Examples:**

```
myReverse [] = []  
myReverse [1] = [1]  
myReverse [1,2,3,4,5] = [5,4,3,2,1]
```

b) `myMap :: (a->b) -> [a]->[b]`

Where *map f xs*

is the list obtained by applying *f* to each element of *xs* .

NOTE: `myMap = map`  
will not get any points.

**Examples:**

```
myMap (+3) [] = []  
myMap (+3) [1,1,1] = [4,4,4]  
myMap sum [[1],[-1]] = [1,-1]
```

c) `myNegate :: [Int] -> [Int]`

Which takes a list of integers and returns a list where every element is negative  
(keeping the same absolute value)

**Examples:**

```
myNegate [] = []  
myNegate [3,-3,-3,-3] = [-3,-3,-3,-3]  
myNegate [-1,1,-1,-1] = [-1,-1,-1,-1]
```

4.

**Include** the function **setElement**, which takes an index **n**, an item **x**, and a list, and returns the given list but with the **n**'th element replaced with **x**. Recall that index 0 refers to the first element of the list. If **n** is negative or larger than the length of the list, the function returns the list without changing it.

```
1. setElement :: Int -> a -> [a] -> [a]
2. setElement n x xs =
3.     if ((n < length xs) && n >= 0) then (take n xs) ++ [x] ++ (drop (n+1) xs)
4.     else xs
```

**Examples:**

```
setElement 0 5 [1, 2, 3, 4] = [5, 2, 3, 4]
setElement 2 5 [1, 2, 3, 4] = [1, 2, 5, 4]
setElement 8 5 [1, 2, 3, 4] = [1, 2, 3, 4]
setElement 3 'x' "hello"    = "helxo"
setElement -5 'x' "hello"   = "hello"
```

Define the function **setElements**, which takes a list **indxs** of pairs of **(n, x)** (where **n** is an **Int**), and a list **xs**, and for each pair **(n, x)** it sets the **n**'th element in **xs** to be **x**.

If a pair contains in its first component a negative index, or an index larger than the length of **xs**, ignore this pair.

If the **indxs** list contains more than one pair with the same key (e.g., [(1,'a'),(1,'b')]), the last occurrence of the key will be the one that holds.

Your function should not be recursive, and should be defined in terms of **foldr**/**foldl** and **map**.

Examples:

```
setElements [(1, 'a'), (-4, 't'), (3, 'b')] "#####" = "#a#b#"
setElements [(2, 50), (50, 2)] [8, 7, 6, 5, 4] = [8, 7, 50, 5, 4]
setElements [(3, 'a'), (5, 'b'), (3, 'c')] "wwwwwww" = "wwwcwbw"
```

The functions signature should be:

```
setElements :: [(Int, a)] -> [a] -> [a]
```