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
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)

Solution:

```
1.  byteSub:: Char ->Char
2.  byteSub 'a' = 'h'
3.  byteSub 'g' = 'b'
4.  byteSub c =  if ((ord c) >= ord 'a') && ((ord c) <= ord 'z') then
5.                  if (c =='z') then 'a'
6.                  else chr((ord c)+1)
7.               else '0'
```

b) Implement a function
`shiftIn` **:: Int -> [Char]->[Char]**

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

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

Solution:

```
1.  shiftIn:: Int ->[Char]->[Char]
2.  shiftIn _ [] = []
3.  shiftIn x lst = if (x<0) then []
4.                  else if (x>(length lst)-1) then shiftIn (x `mod` (length lst)) lst
5.                       else let
                                    y=(length lst) - x
6.                            in
7.                                  (drop y lst)++(take y lst)
```

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 a shifted by i entries.
(where in the first row *i*=0)

Solution:

```
1.  shiftRowsAux :: [[Char]]->Int->[[Char]]
2.  shiftRowsAux [] _ = []
3.  shiftRowsAux (x:xs) i = (shiftIn i x):(shiftRowsAux xs (i+1))
4.
5.  shiftRows :: [[Char]]->[[Char]]
6.  shiftRows mat = shiftRowsAux mat 0
```

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

Solution

```
1.  roundKey:: (Char->Char)->[[Char]]->[[Char]]
2.  roundKey _ [] = []
3.  roundKey enc (x:xs) = (map enc x):(roundKey enc xs)
```

e)

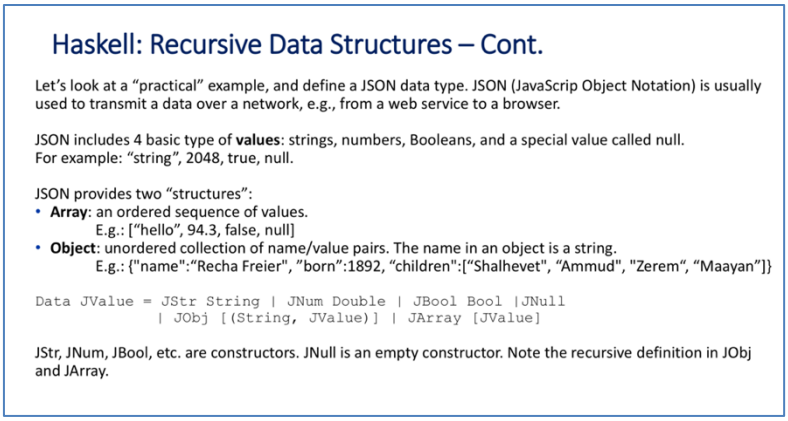
Now we will use all the previous functions except for the first one.
Implement a function:
`simpleAES` **:: (Char->Char)->[[Char]]->String**

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.

Solution

```
1.  implode :: [Char]->String
2.  implode [] = []
3.  implode (x:xs) = x:(implode xs)
4.
5.  simpleAES :: (Char->Char)->[[Char]]->String
6.  simpleAES enc mat = implode(foldr (++) [] (roundKey enc (shiftRows mat)))
```

2. We defined in the lectures a JSON data type:



Haskell: Recursive Data Structures – Cont.

Let's look at a "practical" example, and define a JSON data type. JSON (JavaScrip 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
            | JObj [(String, JValue)] | JArray [JValue]
```

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

For the JSON data type:

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

Define the function:

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

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

Solution

```
1.  import Data.List (intercalate)
2.
3.  displayPairs :: [([Char], JVal)] -> [Char]
4.  displayPairs [] = ""
5.  displayPairs pairs = intercalate ", " (map displayPair pairs)
6.
7.  displayPair :: ([Char], JVal) -> [Char]
8.  displayPair (k,v)   = k ++ ": " ++ jsonToString v
9.
10. displayArray :: [JVal] -> [Char]
11. displayArray [] = ""
12. displayArray vs = intercalate "," (map jsonToString vs)
13.
14. jsonToString :: JVal -> String
15. jsonToString (JStr s)   = s
16. jsonToString (JNum n)   = show n
17. jsonToString (JBool True)  = "true"
18. jsonToString (JBool False) = "false"
19. jsonToString  JNull         = "null"
20. jsonToString (JObj obj) = "{" ++ (displayPairs obj) ++ "}"
21. jsonToString (JArray jArray) = "[" ++ (displayArray jArray) ++ "]"
```

3. Implement the following functions with <u>one line</u> of code:

a) `myReverse :: [a] -> [a]`
Which takes a list and returns the reversed list.

b) `myMap :: (a->b) -> [a]->[b]`
Where *map f xs*
is the list obtained by applying f to each element of xs .

c) `myNegate :: [Int] -> [Int]`
Which takes a list of integers and returns a list where every element is negative (keeping the same absolute value)

Solution:

```
1.  myReverse :: [a] -> [a]
2.  myReverse xs = foldl (flip (:)) [] xs;
3.
4.  myMap :: (a -> a1) -> [a] -> [a1]
5.  myMap f = foldr (\x xs -> f x : xs) []
6.
7.  myNegate::[Int]->[Int]
8.  myNegate xs  = map (negate . abs) xs
```

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.

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

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**.

The functions signature should be:
`setElements` **:: [(Int, a)] -> [a] -> [a]**

Solution:

```
setElements :: [(Int, a)] -> [a] -> [a]
setElements   inds   xs   =   foldr   ($)   xs   (map (\x -> setElement (fst x) (snd x)) (reverse inds))
```

or

```
setElements :: [(Int, a)] -> [a] -> [a]
setElements  inds   xs   =   foldl   (\x f -> f x)   xs   (map (\x -> setElement (fst x) (snd x)) inds)
```