My Report

Me

Tuesday 11th March, 2025

Abstract

We give a toy example of a report in *literate programming* style. The main advantage of this is that source code and documentation can be written and presented next to each other. We use the listings package to typeset Haskell source code nicely.

Contents

1 One Time Pad

In this section we implement the one time pad.

```haskell
module Main where

import System.Environment (getArgs)
import System.Random (randomRs, newStdGen)
import Pad

-- Should generate a key of length n and write it to a file
generateKeyIO :: String -> Int -> IO ()
generateKeyIO keyfile n = do
    key <- generateRandomKeyIO n
    writeFile keyfile key

-- Should read plaintext from input file and key from key file
-- Should write ciphertext to output file
encryptIO :: String -> String -> String -> IO ()
encryptIO output inputFile keyFile = do
    inputContent <- readFile inputFile
    keyContent <- readFile keyFile
    let ciphertext = padString inputContent keyContent
    writeFile output ciphertext

-- Should read ciphertext from input file and key from key file
-- Should write plaintext to output file
decryptIO :: String -> String -> String -> IO ()
decryptIO output inputFile keyFile = do
    inputContent <- readFile inputFile
    keyContent <- readFile keyFile
    let plaintext = padString inputContent keyContent
    writeFile output plaintext

-- Generate a random key of a given length (inside IO) - to not provide a seed manually
generateRandomKeyIO :: Int -> IO String
generateRandomKeyIO n = take n . randomRs ('A', 'Z') <$> newStdGen

-- Given a plaintext, should generate a key of the same length
generateKeyFromPlaintextIO :: String -> String -> IO ()
generateKeyFromPlaintextIO inputFile keyfile = do
    inputContent <- readFile inputFile
    let n = length inputContent
    key <- generateRandomKeyIO n
    writeFile keyfile key

main :: IO ()
main = do
    args <- getArgs
    case args of
        ("generate":inputFile:keyFile:_) -> generateKeyFromPlaintextIO inputFile keyFile
        ("encrypt":outputFile:inputFile:keyFile:_) -> encryptIO outputFile inputFile
            keyFile
        ("decrypt":outputFile:inputFile:keyFile:_) -> decryptIO outputFile inputFile
            keyFile
        _ -> putStrLn "Invalid arguments. Usage:\n\
                      \  generate [input-file.txt] [key-file.txt]\n\
                      \  encrypt [output-file.txt] [input-file.txt] [key-file.txt]\n\
                      \  decrypt [output-file.txt] [input-file.txt] [key-file.txt]"
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

The output of the program is something like this:

```
Hello!
[1,2,3,4,5,6,7,8,9,10]
[100,100,300,300,500,500,700,700,900,900]
[1,3,0,1,1,2,8,0,6,4]
[100,300,42,100,100,100,700,42,500,300]
GoodBye
```

2 Helper Functions

In this section we define helper functions.

```haskell
module Pad (padString) where

-- :set -package bytestring
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as C
--import Data.Word (Word8)
import Data.Bits (xor)

-- xor :: Bits a => a -> a -> a
xorBytes :: B.ByteString -> B.ByteString -> B.ByteString
xorBytes bs1 bs2 = B.pack (B.zipWith xor bs1 bs2)

padString :: String -> String -> String
padString s1 s2 = C.unpack $ xorBytes (C.pack s1) (C.pack s2)

-- main :: IO ()
-- main = do
--     let bytes1 = B.pack [0x01, 0x02, 0x03, 0x04]  -- First byte array
--     let bytes2 = B.pack [0xFF, 0x00, 0xFF, 0x00]  -- Second byte array
--     let result = xorBytes bytes1 bytes2           -- Perform XOR operation
--     print result  -- Output: "\254\STX\252\DLE"
```

References