Information theory: HW #3 code listing

```haskell
{-# LANGUAGE FlexibleContexts    #-}
{-# LANGUAGE MultiWayIf          #-}
{-# LANGUAGE OverloadedStrings   #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TemplateHaskell     #-}
{-# LANGUAGE TupleSections       #-}
{-# LANGUAGE TypeApplications    #-}
{-# LANGUAGE ViewPatterns        #-}

-- | Homework 3

module Archivers () where

import qualified Base                   as B (Show (..))
import           Codec.Compression.GZip (CompressParams (..))
import qualified Codec.Compression.GZip as Z
import           Control.Exception      (assert)
import           Control.Lens           (makeLenses, to, use, uses, view, (%=), (+=),
                                         (.=), (<>=), (^.), _1, _2, _3)
import           Control.Monad.Writer   (MonadWriter, Writer, runWriter, tell)
import           Data.Bifunctor         (first, second)
import           Data.Bits              (testBit)
import qualified Data.ByteString        as BS
import qualified Data.ByteString.Char8  as BSC
import qualified Data.ByteString.Lazy   as BSL
import           Data.List              (dropWhileEnd, findIndex, last, nub, (!!))
import qualified Data.Map.Strict        as M
import           Data.Maybe             (fromJust, mapMaybe)
import           Data.Number.BigFloat   (BigFloat (..), Prec50, PrecPlus20)
import           Data.Ord               (comparing)
import           Data.Ratio             (denominator, numerator, (%))
import qualified Data.Text              as T
import           Data.Word              (Word16)
import           Numeric                (showGFloat)
import           Universum              hiding ((%))

type String = [Char]

dropEnd :: Int -> [a] -> [a]
dropEnd i xs = take (length xs - i) xs

takeEnd :: Int -> [a] -> [a]
takeEnd i xs = drop (length xs - i) xs

log2 :: Floating a => a -> a
log2 k = log k / log 2
log2' = log2 . fromIntegral

proverb :: ByteString
proverb =
    encodeUtf8
        ("Love_the_heart_that_hurts_you,_but_never_hurt_the_heart_that_loves_you." :: Text)

testProverb :: ByteString
testProverb =
    encodeUtf8
        ("if_we_cannot_do_as_we_would_we_should_do_as_we_can" :: Text)

loremIpsum :: ByteString
loremIpsum = encodeUtf8 (
    "Lorem ipsum dolor sit amet, consectetur adipiscing elit. \
    \Cras ornare diam nec interdum mollis. Phasellus tortor felis, \
    \dapibus eu bibendum eu, commodo quis erat. Vestibulum fringilla, \
    \purus semper eleifend laoreet, sem dui volutpat lectus, sed ullamcorper \
    \ante neque id lectus. Nulla ullamcorper egestas nisl, at convallis \
    \leo tempus vel. Sed mi lacus, aliquam ullamcorper purus vitae, vulputate \
    \dignissim ipsum. Nam in est eu quam maximus blandit. Integer nec iaculis \
    \felis. Vestibulum ut cras amet. " :: Text)


newtype Logger w a = Logger
    { getLogger :: Writer [w] a
    } deriving (Functor, Applicative, Monad, MonadWriter [w])

fromWord8 :: [Word8] -> [Char]
fromWord8 = map (chr . fromIntegral)
```

```haskell

-------------------------------------------------------------------------------
-- Huffman
-------------------------------------------------------------------------------

data HuffmanTrace = HuffmanTrace
    { hCurChar   :: Char
    , hProb      :: Ratio Int
    , hCodeWord  :: String
    , hMsgLength :: Int
    }

instance Show HuffmanTrace where
    show HuffmanTrace {..} =
        intercalate
            "|"
            [[hCurChar], show hProb, hCodeWord, show hMsgLength]

huffman :: BSC.ByteString -> Logger HuffmanTrace ((Map Char (String,Int)),String)
huffman input = encode 0 []
  where
    encode i s | i >= BSC.length input = pure (table1, s)
    encode i s = do
        let c = input `BSC.index` i
            (codeWord,m) = table1 M.! c
            cl = length codeWord
            s' = s ++ codeWord
            hProb = m % n
            hMsgLength = length s'
        tell $ [HuffmanTrace {hCurChar = c, hCodeWord = codeWord, ..}]
        encode (i+1) s'
    n = BSC.length input
    firstPass :: Map Char Int
    firstPass = BSC.foldr' (M.alter (pure . maybe 1 (+1))) M.empty input
    calcWords :: [(Double,[(Char,Int)])]
    calcWords =
        map (\(k, x) -> (fromIntegral x / fromIntegral n,[(k,0)])) $ M.assocs firstPass
    calcCodeWordDo [(p,x)] = assert (p == 1) x
    calcCodeWordDo (sortOn fst -> ((p0,lefts):(p1,rights):xs)) =
        let inc = map (second (+1))
        in calcCodeWordDo $ sortOn fst $ (p0 + p1, inc lefts ++ inc rights):xs
    codeWords :: [(Char, String)]
    codeWords = let res = sortOn snd $ calcCodeWordDo calcWords
                in foldl' codeNext [] res
    codeNext [] (c,l) = [(c, replicate l '0')]
    codeNext xs@((_,pr):_) (c,l) =
        let -- generates next codeword after pr of length l
            nextWord = let pr' = dropEnd 1 $ dropWhileEnd (== '1') pr
                       in pr' ++ "1" ++ replicate (l - length pr' - 1) '0'
        in (c,nextWord):xs
    table1 = M.fromList $ map (\(c,s) -> (c,(s,firstPass M.! c))) codeWords

runHuffman x = do
    let ((tbl1, str), tbl2) = runWriter $ getLogger $ huffman x
    forM_ (M.assocs tbl1) print
    forM_ tbl2 print


-------------------------------------------------------------------------------
-- Adaptive arithmetic fixed precision
-------------------------------------------------------------------------------

data ArithmState = ArithmState
    { _aLow     :: Word16
    , _aHigh    :: Word16
    , _aWord    :: [Bool]
    , _aLetters :: [Word8]
    , _aGLog    :: Double
    } deriving Show

makeLenses ''ArithmState

data ArithmTrace = ArithmTrace
    { aCurChar   :: [Char]
    , aProb      :: Rational
    , aCodeWord  :: String
    , aMsgLength :: Int
    }

type ArithM a = StateT ArithmState (Writer [ArithmTrace]) a

instance Show ArithmTrace where
```

```haskell
    show ArithmTrace {..} =
        intercalate
            "|"
            ["", aCurChar, show aProb, aCodeWord, show aMsgLength, ""]

convertToBits :: (Bits a) => a -> Int -> [Bool]
convertToBits x i = reverse $ map (\i -> testBit x i) [0 .. i-1]

arithmStep :: Map Word8 Rational -> Word8 -> ArithM ()
arithmStep prob w = do
    low <- use aLow
    (delta :: Double) <- uses aHigh $ fromIntegral . (\x -> x - low)
    let member = M.member w prob
        letter = bool 0xff w member
        cast = fromRational . toRational
        p, p' :: Word16
        p = round $
            delta *
            M.foldrWithKey
                (\w' pr acc -> bool acc (acc + cast pr) (w' < letter))
                0.0
                prob
        p' = p + round (delta * cast (prob M.! letter))
        matches =
            maximum $ 0 :
            filter
                (\i -> all (\j -> testBit p j == testBit p' j) [16-i .. 15])
                [1 .. 16]
        sameBits = take matches $ convertToBits p 16
        low', high' :: Word16
        low' = shiftL p matches
        high' | matches == 0 = p'
              | otherwise = let s = shiftL p' matches
                            in s .|. (s - 1)
--    traceShowM p
--    traceShowM p'
--    traceShowM sameBits
    aLow .= low'
    aHigh .= high'
    aWord <>= sameBits
    when member $ aLetters %= (letter:)
    l <- uses aWord length
    aGLog += (- (log2 (cast $ prob M.! letter)))
    tell $ [ArithmTrace (chr' letter) (prob M.! letter) (map (bool '0' '1') sameBits) l]

    newLetters <- uses aLetters $ \letters -> filter (not . (`elem` letters)) [0..0xff]
    let probWithEscape =
            M.fromList $ map (\i -> (i, 1 / (fromIntegral $ length newLetters))) newLetters
    when (letter /= w) $ arithmStep probWithEscape w
  where
    chr' 0xff = "esc"
    chr' x    = [chr $ fromIntegral x]

finalizeArith :: ArithM ()
finalizeArith = do
    high <- uses aHigh fromIntegral
    low <- uses aLow fromIntegral
    curL <- uses aWord length
    let delta, deltaP :: Double
        delta = high - low
        deltaP = delta / 0xffff
    bits <- uses aGLog $ \l ->
        take (1 + (ceiling l) - curL) $
        convertToBits @Word16 (round $ low + delta / 2) 16
    aWord <>= bits
    l <- uses aWord length
    tell $ [ArithmTrace "final" 0 (map (bool '0' '1') bits) l]

runAdaptiveArithm :: ByteString -> ArithM ()
runAdaptiveArithm input = do
    forM_ [0..BS.length input-1] $ \k -> do
        letters <- use aLetters
        let n = fromIntegral $ length letters
            probM = M.fromList $
                map (second (/(n+1))) $
                (0xff, 1):
                (map (\l -> (l, fromIntegral $ length $ filter (==l) letters)) $ nub letters)
        arithmStep probM $ BS.index input k
    finalizeArith

execAdaptiveArithm x =
    runWriter $ (runStateT (runAdaptiveArithm x) (ArithmState 0 0xffff [] [] 0))
```

```haskell

-------------------------------------------------------------------------------
-- Enumerative
-------------------------------------------------------------------------------

factorial :: Integer -> Integer
factorial 1 = 1
factorial 0 = 1
factorial x = x * factorial (x-1)

enumerative :: BS.ByteString -> Integer
enumerative input = l1+l2
  where
    n = fromIntegral $ BS.length input
    chars = BS.unpack input
    unique = nub chars
    occurences =
        M.fromList $
        map (\i -> (i, fromIntegral $ length $ filter (== i) chars)) unique
    comp, compcomp, comp' :: [Integer]
    comp = reverse $
           sort $ map (\i -> fromMaybe 0 $ M.lookup i occurences) [0 .. 0xff]
    m = length comp
    compcomp = map (fromIntegral . length) $ group comp
    comp' = filter (> 0) comp
    l2 = ceiling $
         log2' $ foldr (\x acc -> acc `div` (factorial x)) (factorial n) comp'
    l11 = ceiling $ log2' $ n * product comp'
    l12 = ceiling $
          log2' $
          foldr (\x acc -> acc `div` (factorial x))
                (factorial $ fromIntegral $ length comp)
                compcomp
    l1 = l11 + l12


-------------------------------------------------------------------------------
-- Universal coding
-------------------------------------------------------------------------------

bin' :: Int -> [Bool]
bin' x = drop 1 $ dropWhile not $ convertToBits x 32

unar :: Int -> [Bool]
unar n = replicate (n-1) True ++ [False]

elias :: Int -> [Bool]
elias n = p1 ++ p2 ++ p3
  where
    p1 = unar $ 2 + length p2
    p2 = bin' $ length p3
    p3 = bin' n

mon :: Int -> [Bool]
mon n = p1 ++ p2
  where
    p1 = unar $ length p2 + 1
    p2 = bin' n

-------------------------------------------------------------------------------
-- LZ-77
-------------------------------------------------------------------------------


data LZ77State = LZ77State
    { _lzDict :: [Word8]
    , _lzWord :: [Bool]
    } deriving Show

makeLenses ''LZ77State

data LZ77Trace = LZ77Trace
    { lzFlag      :: Bool
    , lzCurString :: String
    , lzDist      :: Maybe Int
    , lzLength    :: Int
    , lzCodeWord  :: [Bool]
    , lzBits      :: Int
    , lzMsgLength :: Int
    }

instance Show LZ77Trace where
```

```
    show LZ77Trace {..} =
        intercalate
            "|"
            [ ""
            , showBool lzFlag
            , lzCurString
            , showMaybe lzDist
            , show lzLength
            , concatMap showBool lzCodeWord
            , show lzBits
            , show lzMsgLength
            , ""
            ]
      where
        showBool False = "0"
        showBool True  = "1"
        showMaybe Nothing  = ""
        showMaybe (Just a) = show a

type LZ77M a = StateT LZ77State (Writer [LZ77Trace]) a

lz77Do :: (Int -> [Bool]) -> BS.ByteString -> Int -> Int -> LZ77M ()
lz77Do uni input _ i | i >= BS.length input = pure ()
lz77Do uni input window i = do
    bestMatch <- uses lzDict workingInputs
--    traceShowM bestMatch
--    traceShowM =<< uses lzDict (map (first fromWord8) . subwords)
    maybe onNewWord onMatch bestMatch
    stripDictionary
    lz77Do uni input window $ i + maybe 1 (length . fst) bestMatch
  where
    onMatch (match,i) = do
        let lzFlag = True
            lzCurString = fromWord8 match
            lzLength = length match
        lzDist <- uses lzDict $ \d -> length d - i
        dictSizeLog <-
            uses lzDict $ ceiling . log2' . (+1) . fromIntegral . length
--        traceShowM lzCurString
--        traceShowM =<< uses lzDict length
        let lzCodeWord =
                lzFlag :
                convertToBits lzDist dictSizeLog ++
                uni lzLength
            lzBits = length lzCodeWord
        lzWord <>= (lzCodeWord)
        lzMsgLength <- uses lzWord length
        tell $ [LZ77Trace {lzDist = Just lzDist,..}]
        lzDict <>= match
    onNewWord = do
        let lzCodeWord = lzFlag : convertToBits (fromJust $ head input') 8
            lzFlag = False
            lzCurString :: String
            lzCurString = fromWord8 $ take 1 input'
            lzDist = Nothing
            lzLength = 0
            lzBits = length lzCodeWord
        lzWord <>= lzCodeWord
        lzMsgLength <- uses lzWord length
        tell $ [LZ77Trace {..}]
        lzDict <>= [fromJust $ head input']
    workingInputs :: [Word8] -> Maybe ([Word8], Int)
    workingInputs dict =
        let filtered = filter (\(pr, _) -> pr `isPrefixOf` input') $
                subwords dict
        in bool (Just $ maximumBy (comparing (length . fst)) filtered)
                Nothing
                (null filtered)
    stripDictionary = do
        l <- uses lzDict length
        when (l > window) $ lzDict %= drop (l - window)
    input' = BS.unpack $ BS.drop i input
    tails' xs = dropEnd 1 (tails xs) `zip` [0 ..]
    inits' xs = concatMap (\(str, i) -> drop 1 $ (,i) <$> inits str) xs
    subwords :: [a] -> [([a],Int)]
    subwords = reverse . inits' . tails'

lz77Encode :: (Int -> [Bool]) -> BS.ByteString -> Int -> LZ77M ()
lz77Encode uni bs w = lz77Do uni bs w 0

execLz77 :: (Int -> [Bool]) -> ByteString -> Int -> (((), LZ77State), [LZ77Trace])
execLz77 u x w = runWriter $ (runStateT (lz77Encode u x w) (LZ77State [] []))
```

```haskell

lz77other x = map (\(u,w) -> (toUniS u, w, exec (toUni u) w)) testData
  where
    toUni 0 = unar
    toUni 1 = mon
    toUni 2 = elias
    toUniS 0 = "Unary"
    toUniS 1 = "Levenshtein"
    toUniS 2 = "Elias"
    testData = [(uni, w) | uni <- [0..2], w <- [50,100,200,500,1000,2000,4000]]
    exec uni w = (execLz77 uni x w) ^. _1 . _2 . lzWord . to length

-------------------------------------------------------------------------------
-- LZ78 (LZW)
-------------------------------------------------------------------------------

data LzwState = LzwState
    { _lzwDict :: [[Word8]]
    , _lzwWord :: [Bool]
    } deriving Show

makeLenses ''LzwState

data LzwTrace = LzwTrace
    { lzwNewWord   :: Maybe String
    , lzwMatch     :: String
    , lzwWordId    :: Int
    , lzwCodeWord  :: [Bool]
    , lzwBits      :: Int
    , lzwMsgLength :: Int
    }

instance Show LzwTrace where
    show LzwTrace {..} =
        intercalate
            "|"
            [ ""
            , fromMaybe "" lzwNewWord
            , lzwMatch
            , show lzwWordId
            , concatMap showBool lzwCodeWord
            , show lzwBits
            , show lzwMsgLength
            , ""
            ]
      where
        showBool False = "0"
        showBool True  = "1"

type LzwM a = StateT LzwState (Writer [LzwTrace]) a


lzwDo :: BS.ByteString -> Int -> LzwM ()
lzwDo input i | i >= BS.length input = pure ()
lzwDo input i = do
    matchIndex <- uses lzwDict workingInputs
    (match :: [Word8]) <- uses lzwDict (!! matchIndex)
    let matchLen | match == [92] = 0
                 | otherwise = length match
    dictSizeLog <-
        uses lzwDict $ ceiling . log2' . pred . fromIntegral . length
    let lzwCodeWord = convertToBits matchIndex dictSizeLog ++
            (if matchIndex == 0
             then convertToBits (fromJust $ head input') 8
             else [])
        lzwBits = length $ lzwCodeWord
    lzwWord <>= lzwCodeWord
    let newWord = let w = take (matchLen + 1) input'
                  in bool (Just w) Nothing (w == match)
        lzwNewWord = fromWord8 <$> newWord
        lzwWordId = matchIndex
        lzwMatch | match == [92] = ""
                 | otherwise = fromWord8 match
    lzwMsgLength <- uses lzwWord length
    whenJust newWord $ \w -> lzwDict <>= [w]
    tell $ [LzwTrace{..}]
    lzwDo input $ i + (bool (length match) 1 $ matchIndex == 0)
  where
    workingInputs :: [[Word8]] -> Int
    workingInputs dict =
        fromMaybe 0 $
        getLast $
```

```haskell
        mconcat $
        map Last $
        map (\n -> findIndex (\w -> length w == n && w `isPrefixOf` input') dict)
            [0..length dict-1]
    input' = BS.unpack $ BS.drop i input

lzwEncode :: BS.ByteString -> LzwM ()
lzwEncode bs = lzwDo bs 0

execLzw :: ByteString -> (((), LzwState), [LzwTrace])
execLzw x = runWriter $ (runStateT (lzwEncode x) (LzwState [BS.unpack "\\"] []))

--------------------------------------------------------------------------------
-- PPMA
--------------------------------------------------------------------------------

data PpmState = PpmState
    { _pLetters :: [Word8]
    , _pGLog    :: Double
    } deriving Show

makeLenses ''PpmState

data PpmTrace = PpmTrace
    { pCurChar      :: Char
    , pContextTimes :: [Int]
    , pContext      :: [Char]
    , pEscProbs     :: [Ratio Int]
    , pCharProb     :: Ratio Int
    }

type PpmM a = StateT PpmState (Writer [PpmTrace]) a

instance Show PpmTrace where
    show PpmTrace {..} =
        intercalate
            "|"
            [ ""
            , [pCurChar]
            , bool (show pContext) "#" (null pContext)
            , intercalate "," (map show pContextTimes)
            , intercalate "," (map showRat pEscProbs)
            , showRat pCharProb
            , ""
            ]
      where showRat r = show (numerator r) ++ "/" ++ show (denominator r)

ppmCalculate :: Int -> ByteString -> Int -> PpmM ()
ppmCalculate _ input i | i >= BS.length input = pure ()
ppmCalculate d input i = do
    history <- use pLetters
    let c = BS.index input i
        maxD = min d (length history `div` 2)
        startContext :: [Word8]
        startContext =
            fromMaybe [] $ head $
            mapMaybe (\d' -> dropEnd 1 . view _2 <$>
                             head (findSubstring history $ takeEnd d' history)) $
            reverse [0..maxD]
        -- Input: exceptions list (match <> c), string s
        -- Output: probability p_t(a|s), new exceptions
        calcProb :: [[Word8]] -> [Word8] -> (Ratio Int, [[Word8]])
        calcProb exs s =
            let τ = filter (\(_,match,_) -> not (match `elem` exs)) $
                    findSubstring (dropEnd (length s) history) s
                τsa = filter ((== c) . view _3) τ
            in (length τsa % (length τ + 1),
               concatMap (tails . view _2) τ)
        calcEscProb :: [[Word8]] -> [Word8] -> Ratio Int
        calcEscProb exs s =
            let τ = filter (\(_,match,_) -> not (match `elem` exs)) $
                    findSubstring (dropEnd (length s) history) s
            in 1 % (length τ + 1)
        encodeEscapes probs ms exs s = do
            let (prob, nextExc) = calcProb exs s
                matchN = length $ findSubstring history s
                probEsc = calcEscProb exs s
                probs' = probEsc:probs
                ms' = matchN:ms
                nonMetProb = 1 % (256 - (length $ nub $ history))
            if | prob /= 0 -> (probs,ms',prob)
               | length s > 0 -> encodeEscapes probs' ms' (nub $ exs++nextExc) $ drop 1 s
```

```haskell
                | otherwise -> (probs',ms',nonMetProb)
        r@(probs,matchNs,prob) = encodeEscapes [] [] [] startContext
--    traceM $ "MaxD: " <> show maxD
--    traceM $ "Start context: " <> show (fromWord8 startContext)
--    traceShowM r
    tell [PpmTrace (chr $ fromIntegral c)
                   (reverse matchNs)
                   (fromWord8 startContext)
                   (reverse probs)
                   prob]
    pLetters <>= [c]
    pGLog += (- (log2 (fromRational . toRational $ product probs * prob)))
    ppmCalculate d input $ i + 1

-- | For a text and pattern it returns the list of matches -- index of
-- start and the next char after the match.
findSubstring :: (Eq a) => [a] -> [a] -> [(Int, [a], a)]
findSubstring t pat =
    mapMaybe (\(i,m) -> guard (pat `isPrefixOf` m) >> pure (i, m, last m)) $
    map (second $ take l) $
    filter ((>= l) . length . snd) $
    [0..] `zip` tails t
  where
    l = length pat + 1

execPpm :: Int -> ByteString -> (((), PpmState), [PpmTrace])
execPpm d x = runWriter $ (runStateT (ppmCalculate d x 0) (PpmState [] 0))

ppmBytes p w = 1 + ceiling (_pGLog $ snd $ fst $ execPpm w p)

-------------------------------------------------------------------------------
-- Burrows-Wheeler
-------------------------------------------------------------------------------

bwTransform :: (Show a, Ord a) => [a] -> ([a], Int)
bwTransform input = (lastCol, fromJust $ findIndex (==input) mapped)
  where
    n = length input
    cycled = cycle input
    mapped = sort $ map (\i -> take n $ drop i $ cycled) [0..n-1]
    lastCol = map last mapped

data MtfState = MtfState
    { _mtfLetters :: [Word8]
    , _mtfOutput  :: [Bool]
    } deriving Show

makeLenses ''MtfState

data MtfTrace = MtfTrace
    { mtfCurChar   :: Char
    , mtfNew       :: Bool
    , mtfDist      :: Int
    , mtfDiff      :: Int
    , mtfCodeWord  :: [Bool]
    , mtfBits      :: Int
    , mtfMsgLength :: Int
    }

instance Show MtfTrace where
    show MtfTrace {..} =
        intercalate
            "|"
            [ ""
            , [mtfCurChar]
            , showBool mtfNew
            , show mtfDist
            , show mtfDiff
            , concatMap showBool mtfCodeWord
            , show mtfBits
            , show mtfMsgLength
            , ""
            ]
      where
        showBool False = "0"
        showBool True  = "1"

type MtfM a = StateT MtfState (Writer [MtfTrace]) a


findIndexLast pred xs =
    (\i -> length xs - i - 1) <$> (findIndex pred $ reverse xs)
```

```haskell

-- | MTF encoding, straight-forward
mtfEncode :: (Int -> [Bool]) -> ByteString -> Int -> MtfM ()
mtfEncode _ bs i | i >= BS.length bs = pure ()
mtfEncode u bs i = do
    history <- use mtfLetters
    let c = BS.index bs i
        mtfNew = not $ c `elem` history
        foundIx = findIndexLast (== c) history
        diffAbsent = (length $ nub history) + 256 - fromIntegral c
        diffPresent i = length $ nub $ drop (i + 1) history
        diff = maybe diffAbsent diffPresent foundIx
        distAbsent = length history + 256 - fromIntegral c
        dist = maybe distAbsent (\j -> i - j + 1) foundIx
        codeWord = u $ diff + 1
    mtfOutput <>= codeWord
    mtfLetters <>= [c]
    mtfMsgLength <- uses mtfOutput length
    tell [MtfTrace (chr $ fromIntegral c) mtfNew dist diff codeWord (length codeWord) mtfMsgLength]
    mtfEncode u bs $ succ i


execMtf :: (Int -> [Bool]) -> ByteString -> (((), MtfState), [MtfTrace])
execMtf u bs = runWriter $ (runStateT (mtfEncode u bs 0) (MtfState [] []))



data MtfSimpleState = MtfSimpleState
    { _mtfsLetters :: [Word8]
    , _mtfsOutput  :: [Word8]
    } deriving Show

makeLenses ''MtfSimpleState

-- Just forms the string for encoding with enumerative later
mtfEncodeEsc :: [Word8] -> State MtfSimpleState ()
mtfEncodeEsc [] = pure ()
mtfEncodeEsc (c:xs) = do
    history <- use mtfsLetters
    let foundIx = findIndexLast (== c) history
        diffAbsent = fromIntegral $ ord '\\'
        diffPresent i = fromIntegral $ length $ nub $ drop (i + 1) history
        diff :: Word8
        diff = maybe diffAbsent diffPresent foundIx
    mtfsOutput <>= [diff]
    mtfsLetters <>= [c]
    mtfEncodeEsc xs

runMtfs bs = execState (mtfEncodeEsc $ BS.unpack bs) $ MtfSimpleState [] []

-------------------------------------------------------------------------------
-- Zlib/Gzip
-------------------------------------------------------------------------------

gzipCompress :: ByteString -> ByteString
gzipCompress =
    BSL.toStrict .
    Z.compressWith
        (Z.defaultCompressParams
         { compressLevel = Z.bestCompression
         , compressMemoryLevel = Z.maxMemoryLevel
         }) .
    BSL.fromStrict
```

Author: Volkhov Mikhail, M3338
Created: 2016-12-18 Sun 17:00
Validate