Nome: Iago Richard da Anunciação da Silva – R.A : 21038514
Disciplina: Paradigmas de Programação

Compactação de arquivos em Haskell usando codificação de Huffman e LZW

Codificação de Huffman

Para a codificação de Huffman foi estabelecido o seguinte *pipeline:*

- Montar tabela ordenada de frequências para os caracteres do arquivo de entrada;
- Criar lista com nós da árvore de codificação para cada caractere;
- Montar árvore de codificação com os nós da lista;
- Atribuir códigos para cada nó da árvore;
- Filtrar caracteres de interesse;
- Montar dicionário (caractere, código);
- Codificar a string de entrada;

Para a criação da árvore de codificação foi implementado um tipo chamado *Arvore* que é definida como nulo ou como um nó que contém um campo tipo Char (que servirá para armazenar os caracteres do arquivo de entrada e o caractere auxiliar), um campo tipo Int (para armazenar a frequência do caractere), um campo tipo String (para armazenar o código do caractere) e duas *Arvores* (esquerda e direita).

```
data Arvore = Nulo | No Char Int String Arvore Arvore
            deriving Show
```

1 – Montar tabela ordenada de frequências para os caracteres do arquivo de entrada

Implementação da função *contarFreq* que recebe uma string e retorna uma lista de tuplas do tipo [(x, y)], sendo x o caractere o y sua frequência na string. São utilizadas algumas funções auxiliares:

- *count*: conta a ocorrência de um caractere em uma string.
- *nub* (from Data.List) : retorna uma lista sem itens duplicados
- *sortOn* (from Data.List) : ordena uma lista de tuplas pelo primeiro (fst) ou pelo segundo (snd) elemento da tupla.

```
contarFreq :: String -> [(Char, Int)]
contarFreq s = (sortOn snd [(y, count y s) | y <- nub s])

count :: Char -> String -> Int
count c s = length freq
  where
    freq = [y | y <- s, y == c]
```

2 - Criar lista com nós da árvore de codificação para cada caractere

Implementação da função *listTree* que recebe a lista de frequências gerada pela função *contarFreq* e devolve uma lista de árvores, cada uma com um único nó do tipo A*rvore.*

```
listTree :: [(Char, Int)] -> [Arvore]
listTree xs = [arv y | y <- xs]
  where
    arv ls = No (fst ls) (snd ls) "" Nulo Nulo
```

3 - Montar árvore de codificação com os nós da lista

Função *monta_Arv* recebe uma lista de árvores gerada pela função *listTree* e devolve uma única árvore com todos os nós. A montagem da árvore é feita recursivamente. É utilizada a função auxiliar *reorganiza_Arv* que reorganiza a subárvore gerada na lista de árvores.

```
monta_Arv :: [Arvore] -> Arvore
monta_Arv []     = Nulo
monta_Arv (x:[]) = x
monta_Arv xs   = monta_Arv (reorganiza_Arv (monta (take 2 xs)) (drop 2 xs))
  where
     monta ((No c1 n1 a1 esq1 dir1) : (No c2 n2 a2 esq2 dir2): []) = No '☠' (n1 + n2) ""
(No c1 n1 a1 esq1 dir1) (No c2 n2 a2 esq2 dir2)
```

```
reorganiza_Arv :: Arvore -> [Arvore] -> [Arvore]
reorganiza_Arv (No c n a esq dir) []   = [(No c n a esq dir)]
reorganiza_Arv Nulo xs                 = xs
reorganiza_Arv (No c1 n1 a1 esq1 dir1) (No c2 n2 a2 esq2 dir2 : xs)
  | n1 > n2    = [(No c2 n2 a2 esq2 dir2)] ++ reorganiza_Arv (No c1 n1 a1 esq1 dir1) xs
  | otherwise  = [(No c1 n1 a1 esq1 dir1)] ++ ((No c2 n2 a2 esq2 dir2) : xs)
```

4 - Atribuir códigos para cada nó da árvore

Função atribuirCodigos atribui códigos para cada nó da árvore gerada por monta_Arv. Para cada nó esquerdo é concatenado “0” ao campo String do nó, e a cada nó direito é aconcatenado “1”. Retorna uma lista de tuplas [(x,y)] onde x é o caractere e y o código atribuido ao mesmo.

```
atribuirCodigos :: Arvore -> [(Char, String)]
atribuirCodigos Nulo                        = []
atribuirCodigos (No c n a Nulo Nulo)        = [(c,a)]
atribuirCodigos  (No  c  n  a  (No  c1  n1  a1  esq1  dir1)  (No  c2  n2  a2  esq2  dir2))  =
[(c,a)] ++ atribuirCodigos m ++ atribuirCodigos n
  where
    m = (No c1 n1 (a++"0") esq1 dir1)
    n = (No c2 n2 (a++"1") esq2 dir2)
```

5 - Filtrar caracteres de interesse e montar dicionário

Função *filtraCodigos* retira da lista gerada por *atribuirCodigos* o caractere ☠, usado para construção dos nós intermediários da árvore de codificação. Retorna o dicionário [(caractere, codigo)].

```
filtraCodigos :: [(Char, String)] -> [(Char, String)]
filtraCodigos xs = [y | y <- xs, fst y /= '☠']
```

6 - Codificar a string de entrada

A função *codificar1* executa todas as funçẽs anteriores, ou seja, recebe a string a ser codificada e devolve o dicionário com códigos para cada caraceter. A função *codificar2* recebe a string a ser codificada e o dicionario montado por *codificar1* e devolve a codificação da string.

```
codificar1 :: String -> [(Char, String)]
codificar1 xs = filtraCodigos
              $ atribuirCodigos
              $ monta_Arv
              $ listTree
              $ contarFreq xs
```

```
codificar2 :: [(Char, String)] -> String -> String
codificar2 _ [] = []
codificar2 dic (x:xs) = retornaCodigo x dic ++ codificar2 dic xs
```

As funções seguintes servem para transformar a string binária que foi formada nas funções anteriores em uma lista de inteiros:

```
--verifica se é multiplo de 8
lenMult8 :: String -> Int
lenMult8 s = (length s) `mod` 8
```

```
--adiciona zeros se necesário
addZeros :: String -> String
addZeros s | lenMult8 s == 0 = s
           | otherwise       = s ++ ['0' | i <- [1..lenMult8 s]]
```

```
--binario para decimal
binstr2dec :: String -> Int
binstr2dec x = aux x 0 ((length x)-1)
  where
    aux x n m
      | null x        = n
      | head x == '0' = aux (tail x) n (m-1)
      | otherwise     = aux (tail x) (n+ 2^m) (m-1)
```

```
--converte toda a String
convertBin :: String -> [Int]
convertBin s = converte (addZeros s)
  where
    converte xs
      | null xs = []
      | otherwise = [binstr2dec (take 8 xs)] ++ converte (drop 8 xs)
```

Codificação de Lempel-Ziv-Welch (LZW)

Para a codificação LZW foram definidas as seguintes funções:

- *inicialDic*

Recebe uma string e retorna uma lista de tuplas [(x,y)] . Essa lista será o dicionário inicial e conterá todas os caracteres que aparecem na string a ser codificada, sem repetição.

```
inicialDic:: String -> [(String, Int)]
inicialDic xs   = [y | y <- zip zs ns]
  where
    zs          = [toString g | g <- nub xs]
    ns          = [1..length xs]
    toString js = [ws | ws <- [js]]
```

- *contem*

Essa função recebe uma string e um dicionário. Retorna “0” se a string não está no dicionário, caso contrário retorna o seu código correspondente.

```
contem :: String -> [(String, Int)] -> Int
contem _ []     = 0
contem xs (y:ys)
  | xs == fst y = snd y
  | otherwise   = contem xs ys
```

- *addDic*

Recebe uma string e a adiciona ao fim do dicionário.

```
addDic :: String -> [(String, Int)] -> [(String, Int)]
addDic st ys = ys ++ [(st, m)]
  where
```

```
    m = n + 1
    n = (snd (head (reverse ys)))
```

- *codifica*

Utiliza as funções anteriores para fazer a codificação da string original. Recebe a string original e o dicionário inicial e retorna a string codificada.

```
codifica :: String -> String -> [(String, Int)] -> [Int]
codifica [] x zs               = [contem x zs]
codifica (x:xs) ys zs
  | contem (ys++[x]) zs /= 0   = codifica xs (ys++[x]) zs
  | otherwise                  = [contem ys zs] ++ codifica xs [x] (addDic (ys++[x]) zs)
```

A função main recebe como argumento o caminhao do arquivo a ser codificado e aplica os algoritmos acima citados

```
main :: IO ()
main =    do args <- getArgs
             s <- readFile (head args)

             --Huffman
             let hdic = codificar1 s
             let hcod = codificar2 hdic s
             let a = convertBin hcod
             let hbin = runPut (mapM_ putWord16le (int2w16 a))

             BS.writeFile "codificado_huffman.bin" hbin

             --LZW
             let ldic = inicialDic s
             let lcod = codifica s "" ldic
             let lbin = runPut (mapM_ putWord16le (int2w16 lcod))

             BS.writeFile "codificado_lzw.bin" lbin
```

A saída para a codificação de Huffman é “codificado_huffman.bin” e para Lzw é “codificado_lzw.bin”