Implementação de um Corretor Ortgoráfico em Haskell

11049114 - Richard Anemam Da Costa ,
11076415 - Juliane Dias Cunha

15 de Agosto de 2018

1 Introdução

Este projeto consiste na implementação de um algoritmo de distância de edição para correção ortográfica.
O banco de palavras foi construído a partir das obras de Machado de Assis, limitando a ação do corretor à Língua Portuguesa.

2 Tutorial

O código-fonte pode ser encontrado no link https://github.com/richardanemam/paradgmas-de-programa-ao.
Inicialmente, para implementar o corretor é necessário construir o dicionário a ser utilizado como base, e para isso é necessário escolher um texto base de onde se irá extrair essas palavras. Neste dicionário, as palavras devem estar normalizadas, sendo então necessário remover toda a pontuação existente, além de transformar todas as letras para minúsculas ou para maiúsculas.
Nesta implementação, considere que foi escolhido deixar todas as letras minúsculas. Como segue o código abaixo, foi utilizado o método toLower, encontrado no módulo Data.Char importado previamente, para retornar a mesma String de entrada, mas com todas as letras minúsculas.

```
lowerWords :: String -> String
lowerWords word = map toLower word
```

Para retirar a pontuação presente no texto, foi utilizado o recurso de compreensão de lista, onde a partir de uma String de entrada, foi definido que todo caractere que não pertence à ",.?!-:;¨ ´"não faz parte da String resultante.

```
-- | Remove punctuation from text String.
removePunc :: String -> String
removePunc word = [w | w <- word, not (w `elem` ",.?!-:;\"\'")]
```

Para então construir o vetor de String contendo as palavras do dicionário, o método a seguir utiliza words, que quebra a String de entrada em uma lista de String contendo as palavras delimitadas pelo espaço. Este é um método encontrado no Prelude (módulo padrão do Haskell)

```
wordsList :: String -> [String]
wordsList word = words word
```

Neste algoritmo de distância, foi considerado que as edições viáveis são aquelas que exigem no máximo duas edições para fazer a correspondência entre uma palavra do dicionário e a palavra de entrada. Deste modo, existe a necessidade de um método que trate impasses, para casos onde existam mais de uma palavra possível para a correção. Considere então que a palavra de maior probabilidade dentre o grupo de possíveis correções, deve ser a palavra escolhida. Assim, associamos a probabilidade de cada palavra do texto base à palavra em si para consulta posterior.
Essa próxima função recebe uma lista de Strings, coloca essa lista em ordem alfabética, utilizando o sort do módulo Data.List. Aplica então o group, gerando uma lista de listas. Cada uma das listas geradas contém o mesmo valor, deste modo todas as palavras iguais foram agrupadas. São mapeadas as palavras e o número de repetições de cada uma delas, resultando em uma lista de tuplas, onde o primeiro valor é a palavra, e o segundo sua frequência no texto.

```
-- | maps how many times a word appears in the file
countWords :: [String] -> [(String,Int)]
countWords xs = map (\w -> (head w, length w)) $ group $ sort xs
```

A partir da contagem acima, é calculada a probabilidade de cada palavra. Para isso, o método a seguir recebe uma palavra e o dicionário, que é mapeado de de um alista de tuplas [(k,a)] para uma tupla (k,a). Aplicando o

Map.lookup na palavra recebida k e na tupla (k,a) obtém-se a, retornando então quantas vezes a palavra k se repete no dicionário. O erro é tratado com Maybe. A aplicação do fromIntegral transforma para um inteiro, para então ser aplicado (/n), que é a divisão pelo total de palavras do dicionário. O map.fold' faz a soma dos valores para retornar o total de palavras.

```
-- | Probability of word.
probability :: String -> [(String, Int)] -> Double
probability word dictOfWords = (/ n) $ fromIntegral $ fromMaybe 0 (Map.lookup word (Map.fromList dictOfWords) ::
    Maybe Int)
  where
    n = fromIntegral $ Map.foldl' (+) 0 (Map.fromList dictOfWords)
```

O helper ajuda a conter um possível impasse. No caso de duas ou mais respostas, o helper devolve a palavra de maior probabilidade calculada de acordo com a sua frequência no texto base, utilizando recursividade para percorrer a lista de candidatos possíveis, de modo que a cada chamada o primeiro item da lista é comparado com o seguinte, e aquele de maior probabilidade passa a ser o primeiro e o outro é removido da lista. Essa chamada ocorre recursivamente até que reste apenas uma palavra, que é a de maior probabilidade dentre as palavras da lista inicial.

```
helper :: [String] -> [(String, Int)] -> String
helper (x:xs) dictOfWords | xs == [] = x
                          | otherwise = if p2 > p
                                        then helper xs dictOfWords
                                        else helper (x:(drop 1 xs)) dictOfWords
                                            where p2 = probability (xs !! 0) dictOfWords
                                                  p  = probability x dictOfWords
```

Para obter todos os candidatos possíveis é necessário mapear então todas as edições possíveis da palavra de entrada que também se encontram no dicionário. No método a seguir é feito o mapeamento que retorna as palavras que também constam no dicionário, utilziando o Map.member.

```
known :: [String] -> [(String, Int)] -> [String]
known words' dictOfWords = [ w | w <- words', Map.member w (Map.fromList dictOfWords)]
```

Neste método foi utilizado compreensão de listas para compor uma lista de String com todas os erros ortográficos aceitáveis.
A partir da palavra de entrada, é construída uma lista com todas as possíveis omissões de até um caractere (deletes), com a inversão de duas letras seguidas (transposes), erro de até um caractere em qualquer posição da palavra (replaces) e inserção de um caractere a mais em qualquer posição da palavra (inserts).

```
-- | All edits that one two edits away from word.
edits1 :: String -> [String]
edits1 word = deletes ++ transposes ++ replaces ++ inserts
  where
    letters    = "abcdefghijklmnopqrstuvwxyz"
    splits     = [ splitAt i word                    | i <- [1 .. length word] ]
    deletes    = [ l ++ tail r                       | (l,r) <- splits, (not . null) r ]
    transposes = [ l ++ r !! 1 : head r : drop 2 r   | (l,r) <- splits, length r > 1 ]
    replaces   = [ l ++ c : tail r                   | (l,r) <- splits, (not . null) r, c <- letters ]
    inserts    = [ l ++ c : r                        | (l,r) <- splits, c <- letters]
```

A seguir foi definido a lista de todas as edições possíveis que permitem a correção da palavra com até duas edições. O número de edições pode ser limitado para até uma edição pela chamada de edits1 na palavra de entrada, e para até duas pela segunda chamada a edits1 sobre o resultado obtido anteriormente.

```
-- | All edits that are two edits away from @word@.
edits2 :: String -> [String]
edits2 word = [ e2 | e1 <- edits1 word, e2 <- edits1 e1 ]
```

Afinal, a lista com os candidatos à correção podem ser gerados, filtrando todas as opções não nulas da lista de edições possíveis que constam no dicionário.

```
candidates :: String -> [(String, Int)] -> [String]
candidates word dictOfWords = head $ filter (not . null) s
```

```
  where
    s = [ known [word] dictOfWords
        , known (edits1 word) dictOfWords
        , known (edits2 word) dictOfWords
        , [word]
        ]
```

Este método recebe a palavra a ser corrigida, o dicionário já construído e chama a função helper na lista de candidatos a correção, retornando a palavra mais viável dentre eles.

```
correction :: String -> [(String, Int)] -> String
correction word dictOfWords = helper list dictOfWords
  where list = candidates word dictOfWords
```

Por fim, no método principal, é feita a leitura do texto que será utilizado como base para o dicionário, a normalização desta entrada, e em seguida é recebida a palavra a ser corrigida, aplicado o correction nela para retornar a melhor opção para correção, que será então impressa na tela.
Observe que nesta implementação é simples fazer a adaptação para outros idiomas de alfabeto romano,uma vez que apenas substituindo o arquivo de entrada, o dicionário será construído a partir deste novo arquivo.

```
main :: IO ()
main = do

  word <- readFile "Todas-as-obras-Machado-de-Assis.txt"
  let dictOfWords = countWords(wordsList(lowerWords(removePunc (word))))

  wordToCorrect <- getLine
  let rightWordBe = correction wordToCorrect dictOfWords

  print (rightWordBe)
```