Processamento de Texto Utilizando Haskell

Nikolas Lacerda[1]

[1]Pontifícia Universidade Católica do Rio Grande do Sul – Porto Alegre – RS – Brazil

nikolas.lacerda@acad.pucrs.br

Abstract. *This article introduces the implementation of a Haskell function library for processing a text. Starting with the implementation of the functions and finally, showing an example of the application of the library.*

Resumo. *Este artigo apresenta a implementação de uma biblioteca de funções em Haskell para o processamento de um texto. Iniciando com a implementação das funções e por fim, demonstrando um exemplo da aplicação da biblioteca.*

1. Introdução

Através deste artigo será demonstrado a implementação de uma biblioteca de funções em Haskell para o processamento de um texto, as funções da biblioteca serão utilizadas em arquivos de texto (.txt) aplicando sua funcionalidade ao texto contido no arquivo.

2. Objetivo

O objetivo é a implementação de uma biblioteca que seja capaz de aplicar algumas das funções mais comuns existentes no tratamento de textos, como a de alinhar, justificar, substituir palavras e extrair informações de um texto, utilizando a linguagem Haskell para a implementação da biblioteca.

3. Implementação

Na aplicação da nossa biblioteca iremos trabalhar somente com Strings e listas de Strings, para facilitar a organização, iremos definir dois tipos novos, o tipo Word e o tipo Line, o tipo **Word** será uma String que irá representar cada palavra em si, e o tipo **Line** será uma lista de palavras (Word) que irá representar uma linha.

```
type Word = String
type Line = [Word]
```

Percebe-se que quando recebemos o arquivo de texto, o seu conteúdo será uma String, então devemos definir funções para tratar essa String como palavras e linhas (Word e Line) que são os tipos que nós definimos.

A primeira função auxiliar que iremos criar é a **getWord**, que irá receber uma String e retornar a primeira palavra dessa String.

```
getWord :: String -> String
getWord [] = []
getWord (x:xs)
    | elem x whitespace = []
    | otherwise         = x : getWord xs
```

Iremos também criar a função **dropWord** e a **dropSpace**, a primeira função, remove a primeira palavra de uma String e a segunda remove o espaço da frente de uma String.

```
dropWord :: String -> String
dropWord [] = []
dropWord (x:xs)
    | elem x whitespace = (x:xs)
    | otherwise = dropWord xs

dropSpace :: String -> String
dropSpace [] = []
dropSpace (x:xs)
    | elem x whitespace = dropSpace xs
    | otherwise = (x:xs)
```

Para o tratamento das linhas, definiremos o **lineLen** que irá representar o tamanho da linha que queremos, por padrão, na nossa biblioteca, vamos definir o tamanho da linha como 35.

```
lineLen :: Int
lineLen = 35
```

A função auxiliar que iremos criar agora é a **getLine** que irá receber um Int, e uma lista de palavras, e retornar uma linha, essa linha terá o tamanho que foi passado pelo Int.

```
getLine :: Int -> [Word] -> Line
getLine len [] = []
getLine len (x:xs)
    | length x <= len = x : restOfLine
    | otherwise = []
    where
    newlen = len - (length x + 1)
    restOfLine = getLine newlen xs
```

Iremos também criar a função **dropLine** que remove uma linha de uma lista de palavras, a linha que será removida terá o tamanho que foi passado pelo Int.

```
dropLine :: Int -> [Word] -> Line
dropLine len [] = []
dropLine len (x:xs)
    | length x <= len = restOfLine
    | otherwise = (x:xs)
    where
    newlen = len - (length x + 1)
    restOfLine = dropLine newlen xs
```

Agora que já conseguimos tratar o texto como palavras e linhas (Word, Line) podemos aplicar as funções de processamento.

A primeira função é a **joinLine**, iremos pegar uma lista de palavras, que é a nossa Line e transformar em uma String.

```
joinLine :: Line -> String
joinLine [] = ""
joinLine (x:[]) = x
joinLine (x:xs) = x ++ " " ++ (joinLine xs)
```

Essa função é aplicada a apenas uma linha, para aplicar ao texto inteiro, iremos criar a função **joinLines.**

```
joinLines :: [Line] -> String
joinLines [] = ""
joinLines (x:xs) = (joinLine x) ++ "\n" ++ (joinLines xs)
```

A função joinLines, transforma uma lista de linhas, que nada mais é que o nosso texto, em uma String, para isso, a função aplica o joinLine em cada uma das linhas.

3.1. Alinhar

Para criarmos uma função para alinhar ou agrupar um texto devemos criar basicamente duas funções principais, uma para retirar o excesso de espaços em um texto, e a outra, dado o tamanho da linha que queremos, dividir o texto em linhas desse tamanho.

Criaremos a função **split** e **splitWords** que receberão uma String e transforma em uma lista de palavras, retirando os espaços com o auxílio da função dropSpace.

```
split :: String -> [Word]
split [] = []
split st = (getWord st) : split (dropSpace(dropWord st))

splitWords :: String -> [Word]
splitWords st = split (dropSpace st)
```

Para dividir o nosso texto em linhas, iremos definir a função **splitLines**.

```
splitLines :: [Word] -> [Line]
splitLines [] = []
splitLines st = getLine lineLen st : splitLines (dropLine lineLen st)
```

Nossa função de alinhamento, **fill**, aplicará o splitLines e o splitWords e nos retornará uma lista de linhas, com isso, nosso texto já estará dividido em linhas do tamanho que queremos, que foi definido pelo lineLen, e devidamente sem os espaços, aplicando o joinLines na função fill, transformamos essa lista de linhas em um texto alinhado, podendo assim, inserir o resultado em um novo arquivo.

```
fill :: String -> [Line]
fill = splitLines . splitWords
```

3.2. Justificar

Para criarmos uma função para justificar um texto devemos criar uma função para contar o número de espaços necessário para preencher a linha.
A função **spacesCount** será a função responsável por fazer isso.

```
spacesCount :: Line -> Int -> Int
spacesCount _ 0  = 1
spacesCount st n
    | div n (length st) > 1 = 1 + (spacesCount st (n - 1))
    | otherwise             = 1
```

A função **justify** irá receber uma linha e um Int, esse Int sendo a quantidade de espaços faltantes para justifica-la.

```
justify :: Line -> Int -> String
justify [] _      = ""
justify (x:xs) n
    | null xs   = x
    | otherwise = x ++ count ++ (justify xs (n - spaces))
    where
    spaces = spacesCount xs n
    count  = replicate spaces ' '
```

A função **justifyLine** irá justificar uma linha, junto com a função justify, elas serão responsáveis por controlar o número de espaços para preencher a linha até o seu tamanho.

```
justifyLine :: Line -> String
justifyLine [] = ""
justifyLine st = justify st spaces
   where
   spaces = lineLen – lineLength st
```

A função **justifyLines** irá justificar todas as linhas do texto, aplicando a função justifyLine para cada linha.

```
justifyLines :: [Line] -> String
justifyLines []     = ""
justifyLines (x:xs)
    | null xs   = joinLine x
    | otherwise = (justifyLine x) ++ "\n" ++ (justifyLines xs)
```

3.3. Utilitários

A nossa biblioteca de processamento de texto também irá possuir algumas funções uteis para utilizarmos em nossos textos, como funções de substituição, de dados e de comparação de textos.

A função **wc**, irá retornar o número de caracteres, palavras e linhas de um texto.

```
wc :: String -> (Int, Int, Int)
wc s = count caracterCount wordCount 1
    where
    caracterCount = length s
    wordCount = length (words s)
    count :: String -> Int -> Int -> Int -> (Int, Int, Int)
    count [] c w l = (c, w, l)
    count (x:xs) c w l
        | x == '\n' && xs /= []  = count xs c w (l + 1)
        | otherwise              = count xs c w l
```

A função **isPalin**, retorna True caso a String indicada seja um palíndromo.

```
isPalin :: String -> Bool
isPalin st = s1 == s2
    where
    s1 = capitalize(dropPontuations(dropAllSpaces st))
    s2 = reverse s1
```

A função **subst**, recebe um texto, e duas palavras, uma indicando a palavra que será substituída e a outra indicando a palavra que irá substitui-la, a função aplica-se a todas as ocorrências da palavra no texto.

```
subst :: String -> String -> String -> String
subst [] oldSub newSub = []
subst (x:xs) oldSub newSub = substRec (x:xs)
    where
    substRec [] = []
    substRec (x:xs) =
        let (prefix, rest) = splitAt n (x:xs)
        in
          if oldSub == prefix
          then newSub ++ substRec rest
          else x : substRec (xs)
    n = length oldSub
```

4. Validação

Agora que a nossa biblioteca de funções foi criada, iremos fazer algumas demonstrações de sua aplicação

4.1. Exemplo 1

Arquivo de entrada:

```
The heat bloomed      in December
  as the   carnival  season
           kicked into  gear.
Nearly helpless with sun and glare, I avoided Rio's brilliant
sidewalks
   and glittering beaches,
panting in dark    corners
and waiting out the inverted southern summer.
```

Arquivo de saída após aplicar a função fill:

```
The heat bloomed in December as
the carnival season kicked into
gear. Nearly helpless with sun and
glare, I avoided Rio's brilliant
sidewalks and glittering beaches,
panting in dark corners and waiting
out the inverted southern summer.

Dados:
Caracteres: 239
Palavras: 37
Linhas: 7
```

Arquivo de saída após aplicar a função justifyLines:

```
The heat bloomed in December  as
the  carnival  season  kicked  into
gear. Nearly helpless with sun  and
glare, I  avoided  Rio's  brilliant
sidewalks and  glittering  beaches,
panting in dark corners and waiting
out the inverted southern summer.

Dados:
Caracteres: 249
Palavras: 37
Linhas: 7
```

4.2. Exemplo 2

Arquivo de entrada:

```
Haskell é uma
linguagem de programação
puramente funcional, de propósito geral,
 nomeada em homenagem ao
       lógico Haskell Curry.
 Como uma linguagem
funcional, a        estrutura de controle
 primária é a função; a linguagem é baseada
 nas observações de Haskell Curry
 e seus descendentes intelectuais.
```

Arquivo de saída após aplicar a função fill:

```
Haskell é uma linguagem de
programação puramente funcional,
de propósito geral, nomeada em
homenagem ao lógico Haskell Curry.
Como uma linguagem funcional, a
estrutura de controle primária é
a função; a linguagem é baseada
nas observações de Haskell Curry
e seus descendentes intelectuais.

Dados:
Caracteres: 305
Palavras: 43
Linhas: 9
```

Arquivo de saída após aplicar a função justifyLines:

```
Haskell  é  uma  linguagem  de
programação puramente  funcional,
de  propósito  geral,  nomeada  em
homenagem ao lógico Haskell Curry.
Como  uma  linguagem  funcional,  a
estrutura de controle primária  é
a função; a linguagem é  baseada
nas observações de Haskell  Curry
e seus descendentes intelectuais.

Dados:
Caracteres: 321
Palavras: 43
Linhas: 9
```

4.2. Exemplo 3

Arquivo de entrada:

```
Haskell é a linguagem funcional
sobre a qual mais se realizam pesquisas atualmente.
Muito utilizada no meio acadêmico. É uma linguagem nova, elaborada em 1987, derivada de outras
linguagens
 funcionais
 como por
 exemplo Haskell
Miranda e ML.
Ela se baseia em um estilo de programação em que se enfatiza mais o
que deve ser feito (what) em detrimento
 de como deve ser feito (how).
É uma linguagem que possui
          foco no alcance de soluções para
problemas matemáticos, clareza, e de fácil manutenção nos códigos,
e possui uma variedade de aplicações e apesar de simples é muito poderosa.
```

Arquivo de saída após aplicar a função fill:

```
Haskell é a linguagem funcional
sobre a qual mais se realizam
pesquisas atualmente. Muito
utilizada no meio acadêmico. É
uma linguagem nova, elaborada em
1987, derivada de outras linguagens
funcionais como por exemplo Haskell
Miranda e ML. Ela se baseia em um
estilo de programação em que se
enfatiza mais o que deve ser feito
(what) em detrimento de como deve
ser feito (how). É uma linguagem
que possui foco no alcance de
soluções para problemas
matemáticos, clareza, e de fácil
manutenção nos códigos, e possui
uma variedade de aplicações e
apesar de simples é muito
poderosa.

Dados:
Caracteres: 599
Palavras: 97
Linhas: 19
```

Arquivo de saída após aplicar a função justifyLines:

```
Haskell é a linguagem funcional
sobre  a  qual  mais  se  realizam
pesquisas        atualmente.  Muito
utilizada no  meio  acadêmico.  É
uma linguagem  nova,  elaborada  em
1987, derivada de outras linguagens
funcionais como por exemplo Haskell
Miranda e ML. Ela se baseia  em  um
estilo de programação em  que  se
enfatiza mais o que deve ser  feito
(what) em detrimento de  como  deve
ser feito (how). É  uma  linguagem
que  possui  foco  no  alcance  de
soluções          para  problemas
matemáticos, clareza, e de  fácil
manutenção nos códigos, e possui
uma  variedade  de  aplicações  e
apesar       de  simples  é  muito
poderosa.

Dados:
Caracteres: 657
Palavras: 97
Linhas: 19
```

5. Conclusão

Vimos que com a utilização da linguagem Haskell, com algumas poucas definições, fomos capazes de criar uma biblioteca com diversas funções uteis para o tratamento de texto, definimos também funções com grande coesão e de uma certa forma simples, para manipular o texto.

Utilizando a maneira de trabalhar o texto, com palavras e lista de palavras, temos grandes métodos de manipula-lo de forma controlada, podendo assim implementar diversas funções para o tratamento do texto.

Também é importante notar que há várias maneiras de fazer a implementação dessa biblioteca, essa é apenas uma delas, que em um próximo trabalho poderemos aplicar ainda mais funções para o tratamento do texto, fazendo assim, uma biblioteca com um leque maior de opções de manipulação.

Referencias

THOMPSON, Simon. Haskell: the craft of functional programming. Third Edition. Addison-Wesley, 2011.