Programação Funcional — O problema das palavras mais frequentes

Pedro Vasconcelos, DCC/FCUP

abril 2021

Objetivo

Pretendemos resolver o seguinte problema: dado um texto e um inteiro $N > 0$, determinar as N palavras mais frequentes no texto (i.e. com maior número de ocorrências).

Vamos representar o texto como uma lista de carateres (`String`). A função pretendida terá o seguinte tipo:

```
commonWords :: Int -> String -> [(String, Int)]
```

O primeiro argumento inteiro é o número N de palavras distintas; o segundo argumento é o texto; o resultado é a lista das N palavras mais comuns associadas à contagem das suas ocorrências.

Por exemplo: partindo o texto da *A Cidade e as Serras*, de Eça de Queirós[1] podemos obter as 20 palavras mais comuns com o seguinte programa:

```
main = do
   txt <- readFile "pg18220.txt"
   print (commonWords 20 txt)
```

Primeiros passos

Começamos por importar algumas funções de bibliotecas úteis para processar listas e carateres.

```
import Data.List
import Data.Char
```

Visão global

A ideia geral do algoritmo para este problema será então:

(1) converter todo o texto para minúsculas;
(2) decompor o texto em palavras;
(3) ordenar a lista das palavras;

[1] Disponível no Projeto Gutenberg: http://www.gutenberg.org/ebooks/18220.

(4) agrupar palavras iguais e contar ocorrências;
(5) ordenar por ordem decrescente de contagens;
(6) retirar as primeiras N palavras.

Podemos traduzir as fases deste algoritmo como a composição de funções mais simples:

```
commonWords :: Int -> String -> [(String, Int)]
commonWords n
   = take n . sortRuns . countRuns . sort . words . map toLower
 --     (6)       (5)          (4)       (3)    (2)        (1)
```

Recorde que a composição de funções $f \circ g$ é a função que, a cada x associa $f\,(g\,x)$. Isto quer dizer que devemos ler o encadeamento acima da direita para a esquerda: a primeira operação efetuada será a conversão em minúsculas (1), depois a conversão em palavras (2) e assim sucessivamente.

Em vez de usar o operador `(.)` poderiamos ter escrito a composição de funções explicita e dando um nome `txt` ao texto de entrada:

```
commonWords n txt
   = take n (sortRuns (countRuns (sort (words (map toLower txt)))))
```

A vantagem de usar o operador `(.)` é que evita os parêntesis e torna mais evidente as diferentes fases do algoritmo. Isto é particularmente útil se quisermos acrescentar novas fases mais tarde.

Implementação das fases do algoritmo

Converter em minúsculas

Devemos ignorar a distinição entre maiúsculas e minúsculas ao contar ocorrências de palavras. Por exemplo, ocorrências de "`Jacintho`", "`jacintho`" e "`JACINTHO`" devem ser contabilizadas como a mesma palavra. Podemos garantir isso convertendo todas as letras para uma mesma forma (por exemplo, minusculas) usando a função `toLower :: Char -> Char`. Esta função opera sobre carateres individuais; para converter para minúsculas todo o texto (1) basta fazer `map toLower`.

```
map toLower :: String -> String
```

Note ainda que `toLower` deixa inalterados quaisquer carateres que não sejam letras, pelo que podemos aplicar `map toLower` a todo o texto.

Partir o texto em palavras

Vamos considerar que os espaços, tabulações e mudanças de linha separam as palavras e qualquer sequência de outros carateres constitui uma palavra.

A função `words :: String -> [String]` do prelúdio-padrão parte uma cadeia de texto numa lista de palavras segundo este critério. Por exemplo:

```
   words "---O meu amigo Jacintho nasceu n'um palacio."
== ["---O","meu","amigo","Jacintho","nasceu","n'um","palacio."]
```

Note que consideramos os sinais de pontuação como fazendo parte das palavras. Mais à frente veremos como melhorar esta situação.

Ordenar a lista de palavras

A ordenação pode ser feita pela função genérica `sort` do módulo `Data.List`.

```
sort :: [String] -> [String]
```

Note que `sort` admite um tipo mais geral; o tipo acima é o deste uso concreto para ordenar uma lista de palavras.

Agrupar e contar ocorrências

A função `countRuns` deve agrupar palavras iguais, juntamente com a contagem de ocorrências. Podemos definir usando `takeWhile` e `dropWhile`, uma vez a lista de palavras já está ordenada pelo que quaisquer ocorrências repetidas têm de estar seguidas:

```
countRuns :: [String] -> [(String,Int)]
countRuns [] = []
countRuns (w:ws) = (w, 1+length ws') : countRuns ws''
   where ws' = takeWhile (==w) ws
         ws''= dropWhile (==w) ws
```

Ordenar por ordem decrescente de contagens

A função `sortRuns` deve ordenar a lista resultante por ordem decrescente de contagens. Para isso usamos uma função `sortBy` do módulo `Data.List` — uma generalização de `sort` de listas cujo primeiro argumento é uma função que devolve uma *comparação* (`LT`, `EQ`, `GT`). Vamos definir a comparação de pares usando `compare` das contagens mas pela *ordem inversa*; desta forma vamos obter os pares por ordem decrescente das ocorrências.

```
sortRuns :: [(String,Int)] -> [(String, Int)]
sortRuns = sortBy (\(w1,c1) (w2,c2) -> compare c2 c1)
```

Experimentação

Combinado todas as definições podemos obter as 20 palavras mais comuns no texto de Eça de Queiroz:

```
main = do
   txt <- readFile "pg18220.txt"
   print (commonWords 20 txt)
```

O resultado é:

```
[("de",2431),("e",2426),("o",2275),("a",2136),("que",1560),
("com",968),("um",901),("do",838),("da",816),("os",744),
("uma",715),("para",679),("se",609),("as",593),("em",590),
("na",467),("no",455),("n\227o",409),("meu",407),("por",377)]
```

Melhoramentos

Há alguns pontos que podemos agora melhorar neste programa. O primeiro aspecto é que (previsivelmente) obtemos muitas ocorrências de palavras curtas como artigos e preposições. Podemos eliminar estas casos de forma simples retirando todas as palavras com (digamos) comprimento é inferior a 4. Como o função `commonWords` está definida como uma composição podemos simplesmente insertir um `filter` após decompor em palavras; o resto do programa não precisa de ser modificado:

```
commonWords n
   = take n . sortRuns . countRuns . sort
     . filter ((>3).length) .  words . map toLower
```

O segundo melhoramento é mais subtil: pela forma como `words` funciona, consideramos que as palavras incluem sinais de pontuação e algarismos além de letras. Isto faz com que, por exemplo, "`d'olival`" e "`olival.`" e "`olival?`" sejam consideradas palavras distintas. Para tratar estes casos poderiamos pensar em re-escrever uma função especial semelhante a `words`, mas há uma forma mais simples: basta modificar o texto à entrada convertendo em espaços todos carateres do texto original que não são letras (sinais de pontuação, algarismos e carateres brancos); desta forma podemos continuar a usar `words` e obter o comportamento desejado.

```
commonWords n
   = take n . sortRuns . countRuns . sort
     . filter ((>3).length) .  words . map normalize

normalize :: Char -> Char
normalize x | isLetter x = toLower x
            | otherwise  = ' '
```

Corremos o programa com estes melhoramentos e obtermo a lista revista das 20 palavras mais comuns:

```
[("para",702),("jacintho",477),("como",394),("mais",272),
("elle",227),("sobre",215),("principe",210),("depois",195),
("muito",188),("onde",182),("entre",149),("quando",143),
("todos",140),("ent\227o",138),("ainda",137),("toda",136),
("pela",126),("fernandes",121),("minha",118),("tormes",117)]
```

Epílogo: decomposição funcional

Este exemplo ilustra a metodologia de *decomposição funcional* para a resolução de um problema: decompomos um algoritmo em fases sucessivas que correspondem a funções separadas; exprimimos a solução usando a composição das funções de bibliotecas e do prelúdio. Esta abordagem permitiu não só resolver o problema de forma sucinta mas também fazer modificações modulares, acrescentando novas fases sem ter de alterar as componentes que não foram afetadas.
