o paradigma de programação funcional requer uma reviravolta do pensamento acerca da programação de computadores. é um "clique" na mente, que pode ocorrer quando menos se espera. quem está procurando chegar lá, começa sua jornada tateando as ideias por diversos lados. metáforas, exemplos e exercícios ajudam.

1. Uma visão diferente sobre a programação

Bird (2015, p. 1) resume: "programação funcional é um método de construção de programas que enfatiza funções e aplicação[1] de funções em vez de comandos e sua execução".

Nas palavras de Fogus (2013, p. 4), "uma imagem que se pode fazer de um sistema construído com princípios funcionais é a de uma linha de montagem, em que matérias-primas brutas entram em uma ponta e, paulatinamente, se transformam, até saírem como produtos acabados pela outra ponta". O autor salienta que as funções são as unidades de abstração com as quais se constroem os sistemas. Em outras palavras, funções são tijolos básicos, e programar é combiná-los.

2. Exemplo: números perfeitos

Um clássico do ensino de programação de qualquer tipo: o programa dos números perfeitos. Um número perfeito é igual à soma dos seus divisores, excluindo o próprio número e incluindo o 1. Por exemplo, 6 é perfeito pois 6 = 3 + 2 +1. 28 também é perfeito: 28 = 14 + 7 + 4 + 2 + 1. Por outro lado, 12 não é perfeito pois $12 \neq 6 + 4 + 3 + 2 + 1 = 16$.

Para gerar uma lista de números perfeitos, basta criar um predicado `perfeito?` e usá-lo como filtro na sequência dos números inteiros: `(filter perfeito? (rest (range)))` Obviamente, a sequência *lazy* assim gerada é infinita, e, para obter valores dela (não se recomenda pedir mais que 4), deve-se aplicar `take`: `(take 4 (filter perfeito? (rest (range))))`

O trabalho se resume, então, a definir `perfeito?` . Esse, por sua vez, também resulta de uma filtragem, mas dessa vez removendo os não divisores. Um predicado `divisor?` é trivial: `(= (rem n d) 0)` , ou seja, se o resto da divisão inteira de n por d for 0, então d é divisor de n. Os valores que permanecem após a filtragem pode ser somados com `(reduce + ,,,)` . O programa, portanto, fica assim:

editor ou repl

```
(defn perfeito?
  [n]
  (let [divisor? (fn [d] (= (rem n d) 0))
        divisores (filter divisor? (range 1 n))
        soma (reduce + divisores)]
    (= soma n)))
```

Essa versão com variáveis locais é muito didática, pois a etapas das aplicações ficam mais visíveis e documentadas com os nomes das variáveis. A programação funcional se mostra como uma cadeia de aplicações de funções. Não há laços (eles estão embutidos em `map`, `filter` e `reduce`) nem condicionais (em `filter`), nem variáveis auxiliares de acesso a índices de *arrays* (isso importantíssimo, pois não é fácil captar de primeira a ideia de um algoritmo se ele envolve muitos índices). Em vez disso, há funções para tudo e elas são combinadas claramente.

A versão *inline* (todas as aplicações de funções embutidas nos parâmetros de outras funções) para a mesma função é:

editor ou repl

```
(defn perfeito?
  [n]
  (= n (reduce + (filter #(= (rem n %) 0) (range 1 n)))))
```

[1] O sentido da palavra *aplicar* é bem específico, como na matemática ("Ele *aplicou* a função ao resultado da operação."), e **não** deve ser confundido com sentidos mais comuns ("Ele aplicou a teoria na sala de aula.", "A indústria aplicou a tecnologia."). A programação funcional também é conhecida como *programação aplicativa*.

3. Exemplo: palavras mais usadas[2]

Quais são as 100 palavras que mais aparecem em *Dom Casmurro*? Quais são as 50 palavras que mais aparecem na tradução de *O Senhor dos Anéis*?

Vamos construir uma função que resolve questões desse tipo. Suas entradas são a quantidade de palavras no resultado, e o texto da obra na forma de string. Sua saída é uma string que mostra o resultado de forma tabulada quando passada para uma função de impressão. Exemplo:

ideia

```
(palavras+usadas 2 "Era uma vez, na aurora de nossa era, uma jovem orgulhosa do que era.")
;=> "   era:  3\n   uma:  2\n"
(print *1)
;   era:  3
;   uma:  2
;=>nil
```

A construção da função é *top-down*.[3] Antes de começar a sessão de programação no REPL, cria-se um exemplo (para teste no REPL):

editor ou repl

```
(def exemplo "Era uma vez, na aurora de nossa era, uma jovem orgulhosa do que era.")
```

O primeiro passo é definir a função na linguagem de programação. A lacuna aguarda a programação de fato:

ideia

```
(defn palavras+usadas
  "n palavras mais frequentes em texto"
  [n texto]
  ___ )
```

A função será um encadeamento de outras funções (uma composição de funções, tecnicamente falando), as lacunas (ainda não se sabe quantas) aguardam as etapas do processo:

ideia

```
(defn palavras+usadas
  "n palavras mais frequentes em texto"
  [n texto]
  (->> texto
       ____
       ____
       ,,, )
```

As etapas internas antevistas para a função são:

1. Uniformizar a string de texto como caracteres minúsculos
2. Remover sinais de pontuação da string de texto
3. Separar a string de texto em strings de palavras
4. Ordenar as strings de palavras
5. Contar as strings repetidas, registrando os pares quantidade-string
6. Ordenar os pares quantidade-string em ordem decrescente
7. Pegar os n primeiros pares quantidade-string
8. Aplicar aos pares quantidade-string uma função de formatação para impressão de cada par
9. Concatenar as strings de impressão de cada par numa string final

Alguns dos passos traçados têm uma implementação imediata em Clojure, outros requerem a criação de funções auxiliares (a numeração tem relação com a lista anterior):

1. A classe `String` (Java) dispõe do método `toLowerCase` , que em Clojure pode ser usado assim: `(.toLowerCase exemplo)`
2. A classe `Character` (Java) dispõe dos métodos **de classe** `isLetter` e `isSpace` que podem ser combinados com `filter` (que resulta numa sequência de caracteres, que pode ser convertida em string novamente com `str`) :

repl

```
(filter #(or (Character/isLetter %)(Character/isSpace %)) exemplo)
(apply str *1)
```

[2] Esse exemplo foi adaptado de Bird (2015), p. 3.

[3] Além de ilustrar o processo *top-down*, essa construção mostra o desenvolvimento interativo com apoio do REPL: ideias são testadas no REPL e incorporadas em seguida no programa.

3. O namespace `clojure.string` dispõe da função `split` :
   `(clojure.string/split exemplo #"\s+")`
4. A função `sort` de Clojure:

repl
```
(sort ["um" "dois" "um" "dois" "um"])
;=> ("dois" "dois" "um" "um" "um")
```

5. É necessário criar uma função (vamos chamá-la `contar-vezes`) cujo efeito seria assim:

ideia
```
(contar-vezes ("dois" "dois" "um" "um" "um"))
;=> ([2 "dois"] [3 "um"])
```

   Aparentemente, não há uma função exatamente como esta em Clojure. A discussão sobre a construção é, em si, um excelente exemplo de composição de funções, e é feita adiante.
6. A função `sort-by` permite ordenar os pares quantidade-string usando o campo quantidade como critério (o campo quantidade é o primeiro do par, por isso pode ser acessado com a função `first`). Para obter uma ordenação decrescente, a função `sort-by` (assim como a função `sort`) possui um parâmetro opcional para receber uma função de comparação:

repl
```
(sort-by first > [[2 "dois"] [3 "um"]])
;=> ([3 "um"] [2 "dois"])
```

7. A função `take` atende aos requisitos:

repl
```
(take 2 [[20 "foi"] [17 "era"] [4 "pois"] [3 "mas"]])
;=> ([20 "foi"] [17 "era"])
```

8. Basta criar uma função que recebe um par e retorna a string pretendida:
   `#(str "   " (second p) ":   " (first p) "\n")`

repl
```
 (map #(str "   " (second %) ":   " (first %) "\n") [[20 "foi"] [17 "era"]])
;=> ("   foi:   20\n" "   era:   17\n")
```

9. A função `join` de `clojure.string` .

A função `contar-vezes` pode ser resolvida se usarmos a função `partition-by`. Ela separa uma sequência em subsequências que possuem um mesmo valor para uma propriedade (é preciso que haja uma função para calcular a propriedade). Por exemplo, na sequência a seguir, palavras vizinhas têm o mesmo número de letras (a função `count` aplicada a uma string conta letras). Sempre que há uma mudança de propriedade, `partition-by` cria uma nova subsequência no resultado:

```
(partition-by count ["ic1" "ic2" "org" "lm" "ed1" "ed2" "clp" "lpnc"])
;=> (("ic1" "ic2" "org") ("lm") ("ed1" "ed2" "clp") ("lpnc"))
```

No caso da função `contar-vezes`, a intenção é agrupar strings iguais, portanto a propriedade do item de lista a ser avaliada é o valor do próprio item, que é acessado por intermédio da função identidade. A sequência assim gerada permite contar as repetições (com `count` e `map`). No final, deve-se montar os pares. A sessão de REPL abaixo é um ensaio dessas ideias.

repl
```
(partition-by identity ["ic1" "ic1" "ic2" "ic2" "ic2"])
;=> (("ic1" "ic1") ("ic2" "ic2" "ic2"))
(map count *1)
;=> (2 3)
(map first *2)
;=> ("ic1" "ic2")
(interleave *2 *1)
;=> (2 "ic1" 3 "ic2")
(partition 2 *1)
;=> ((2 "ic1") (3 "ic2"))
```

A seguir, edita-se a função.

editor
```
(defn contar-vezes
 [palavras]
 (let [agrupadas (partition-by identity palavras)]
   (partition 2 (interleave (map count agrupadas) (map first agrupadas)))))
```

Voltando para a função `palavras+usadas`, já é possível ensaiar as ideias no REPL:

repl

```
(.toLowerCase exemplo)
;=> "era uma vez, na aurora de nossa era, uma jovem orgulhosa do que era."
(apply str (filter #(or(Character/isLetter %)(Character/isSpace %)) *1))
;=> "era uma vez na aurora de nossa era uma jovem orgulhosa do que era"
(clojure.string/split *1 #"\s+")
;=> ["era" "uma" "vez" "na" "aurora" "de" "nossa" "era" "uma" "jovem" "orgulhosa" "do" "que" "era"]
(sort *1)
;=> ("aurora" "de" "do" "era" "era" "era" "jovem" "na" "nossa" "orgulhosa" "que" "uma" "uma" "vez")
(contar-vezes *1)
;=> ((1 "aurora") (1 "de") (1 "do") (3 "era") (1 "jovem") (1 "na") (1 "nossa") ,,,
(sort-by first > *1)
;=> ((3 "era") (2 "uma") (1 "aurora") (1 "de") (1 "do") (1 "jovem") (1 "na") ,,,
(take 2 *1)
;=> ((3 "era") (2 "uma"))
(map #(str "   " (second %) ":   " (first %) "\n") *1)
;=> ("   era:   3\n" "   uma:   2\n")
(clojure.string/join *1)
;=> "   era:   3\n   uma:   2\n"
```

A função final:

editor

```
(defn palavras+usadas
 "n palavras mais frequentes em texto"
 [n texto]
 (->> (.toLowerCase texto)
      (filter #(or(Character/isLetter %)(Character/isSpace %)) )
      (apply str )
      (#(split % #"\s+") )
      sort
      contar-vezes
      (sort-by first > )
      (take n )
      (map #(str "   " (second %) ":   " (first %) "\n") )
      join))
```

A função `split` recebe a string numa posição inconveniente para *threading macro* `->>` , por isso o artifício: `(split x #"\s+")` = `( #(split % #"\s+") x )` .

3.1 Observações

Os processos de construção de `contar-vezes` e `palavras+usadas` mostram uma técnica fundamental: a visualização dos resultados intermediários entre uma função e outra. Eles são explicitamente observados na sessão do REPL. Contudo, quando se analisa uma função pronta, é útil, como técnica, tentar visualizar os dados intermediários entre as aplicações de funções. E, também, especialmente, ao se **projetar** uma função.

Em programação funcional, manipular as funções é mais prático do que em C ou Java. Os passos intermediários do processo estão separados na cadeia de chamadas. É fácil inserir um passo a mais, modificar ou remover um já existente. Isso é similar à passagem da lógica de saltos em linguagem de máquina para as expressões lógicas em linguagens de alto nível: modificar a lógica era penoso (envolvia uma complexa operação nos saltos) e passou a ser trivial em expressões como `!x&&(wait()||SIGNAL)`. As sub-funcionalidades das funções acima aparecem separadas e combinadas; em C ou Java, elas estariam embaraçadas em laços e condicionais aninhados.

Java

```
int f(List<Integer> x) {
  List<Integer> y = new ArrayList<>();
  List<Integer> z = new ArrayList<>();
  int soma = 0;
  for (Integer a : x)
    if (a < 100)
      y.add(a);
  for (Integer b : y)
      z.add(b + 10);
  for (Integer c : z)
    soma += c;
  return soma;
}
```

Clojure

```
(defn f
  [x]
  (->> x
       (filter #(< % 100) )
       (map #(+ 10 %) )
       (reduce + )))
```

O comparativo acima é uma versão simplificada da afirmação. Considere o caso de as funções `filter`, `map` e `reduce` ali terem funções muito mais complexas do que `#(< % 10)`, `#(+ 10 %)` e + respectivamente.

Vê-se em tudo isso o princípio reforçado por Fogus: funções são as unidades de abstração.

Em Lisp, combinações corriqueiras de funções ganham identidade e passam a compor a "linguagem". O exemplo da numeração de uma *seq*:

repl

```
(defn number-seq
  [seq]
  (partition 2 (interleave (rest (range)) seq)))
;=> #'user/number-seq
(number-seq [10 100 1000])
;=> ((1 10) (2 100) (3 1000))
```

Explore mais conhecendo as funções `mapcat`, `map-indexed`, `mapv`, `if-not`, `let-if`.

3.2 Exercícios sobre um dicionário de anagramas[4]

Considere um conjunto de palavras da língua portuguesa reunidos em uma lista em Clojure.

Construa uma função, chamada `anagramas`, que recebe como parâmetros um número e o conjunto mencionado acima. A função deve gerar uma string que, ao ser impressa, mostra de forma tabulada os anagramas encontrados no conjunto com a quantidade de letras igual ao número fornecido, da seguinte forma:

```
(def palavras (clojure.string/split-lines (slurp "<path>/palavras.txt")))
(print (anagramas 6 palavras))
Palavras com 6 letras
---------------------
aabcot: abacto, batoca, tabaco, taboca
aabdlo: badalo, balado, bolada, oblada
...
```

Variações:

Modifique a função `anagramas` , criando a função `conta-anagramas`, com os mesmos parâmetros, que informa a quantidade de grupos de anagramas com n letras. No exemplo acima, abacto, batoca, tabaco e taboca pertencem a um mesmo grupo de anagramas.

Modifique a função `anagramas` , criando a função `max-anagramas`, com os mesmos parâmetros, que informa a maior quantidade de anagramas de um grupo de anagramas com n letras. No exemplo acima, ambos os grupos possuem 4 elementos.

Crie a função `histograma`, que tem como único parâmetro uma função que aceita um argumento inteiro, e retorna uma sequência de valores da função na faixa de 4 a 16. Aplique-a com `conta-anagramas` e `max-anagramas` para visualizar o comportamento dos anagramas para vários tamanhos de palavras.

Crie a função `plot` para imprimir barras horizontais para os valores fornecidos por `histograma`.

[4] Esse exercício foi adaptado de Bird (2015, p. 15, 18).

4. Exemplo: títulos[5]

Títulos de trabalhos científicos seguem uma regra: as letras iniciais de todas as palavras devem ser maúsculas. Há controvérsias a respeito, porém, para os fins deste exemplo, essa é uma regra interessante. O objetivo é criar uma função `modernizar` que adequa qualquer título à regra citada. Por exemplo:

ideia

```
(modernizar "As concomitantes paripotéticas das macérias hirundinas")
;=> "As Concomitantes Paripotéticas Das Macérias Hirundinas"
```

É claro que o problema se resolve completamente no nível da palavra como unidade de processamento. O restante consiste em quebrar o título em palavras, aplicar a solução a todas as palavras individualmente, e reunir os resultados em um título final.

Os passos antevistos para tratar uma palavra são:

1. Obter a palavra com as letras minúsculas
2. Obter a letra inicial e transformá-la em maiúscula
3. Reunir a letra inicial e as demais

O programa correspondente é:

editor

```
(defn corrigir
  [p]
  (let [minusc (.toLowerCase p)
        prim (Character/toUpperCase (first minusc))
        resto (rest minusc)]
    (apply str (conj resto prim))))

(defn modernizar
  [s]
  (join " " (map corrigir (split s #"\s+"))))
```

5. Exemplo: planilha

Este exemplo ilustra mais uma vez o processo de desenvolvimento baseado em REPL. Além disso, nele se encontra uma situação em que a natureza dinâmica da linguagem Clojure proporciona flexibilidade e agilidade.

Uma simples planilha eletrônica encontra alternativas convenientes de implementação em Clojure.

Item	Estoque	Vendidos	Devolvidos
Caneta	1000	600	15
Borracha	5000	750	3
TOTAL	6000	1350	18

O primeiro passo é representar a planilha em si. Uma possibilidade é desenhar um *map* para linhas, e alojar diversos *maps* assim construídos em um vetor.

editor ou repl

```
(def planilha [{:desc "caneta" :est 1000 :vend 600 :dev 15}
               {:desc "borracha" :est 5000 :vend 750 :dev 3}])
```

Para totalizar os valores nas colunas, extraem-se os valores de uma dada coluna numa *seq*.

repl

```
(map :est planilha)
;=> (1000 5000)
```

Somam-se os valores e o resultado é armazenado em um *map*.

repl

```
(reduce + *1)
;=> 6000
(hash-map :est *1)
;=> {:est 6000}
```

[5] Esse exemplo foi adaptado de Bird (2015), p. 38.

Para finalizar, repetem-se os passos para as demais colunas. Ops! Trabalho repetitivo à vista! E se a planilha tivesse 15 colunas? E se a planilha sofresse alterações ao longo dos meses? Em outras palavras, automatizar o trabalho com as colunas é possível??? A resposta: graças às propriedades dinâmicas da linguagem Clojure, SIM!! Os campos de um *map*, representados por suas *keys*, podem ser vistos como uma *seq* qualquer!!!

A planilha deve conter apenas colunas numéricas. Planilhas com colunas de texto são retomadas adiante. O primeiro passo é obter as *keys*, o que se faz aplicando a função `keys` a qualquer *map* (linha) da planilha.

repl

```
(def planilha [{:est 1000 :vend 600 :dev 15}
               {:est 5000 :vend 750 :dev 3}])
;=> #'lab.planiha/planilha
(first planilha)
;=> {:est 1000, :vend 600, :dev 15}
(keys *1)
;=> (:est :vend :dev)
```

Para cada *key* nessa *seq*, basta repetir o procedimento inicial, ou seja, extrair os valores correspondentes em `planilha` e somá-los. Continuando a sessão no REPL:

*repl (**cont.**)*

```
(map #(map % planilha) *1)
;=> ((1000 5000) (600 750) (15 3))
(map #(reduce + %) *1)
;=> (6000 1350 18)
```

No final, para reunir os resultados, monta-se um *map* com as mesmas *keys* de `planilha`.

*repl (**cont.**)*

```
(keys (first planilha))
;=> (:est :vend :dev)
(interleave *1 *2)
;=> (:est 6000 :vend 1350 :dev 18)
(apply hash-map *1)
;=> {:vend 1350, :est 6000, :dev 18}
```

O algoritmo definido na sessão de REPL pode ser apreendido numa função, como segue. O parâmetro `p` significa "planilha". As *keys*, por serem usadas mais de uma vez, são manipuladas através da variável local `ks` definida com `let`.

editor

```
(defn processa
  [p]
  (let [ks (keys (first p))]
    (->> (map #(map % p) ks)
         (map #(reduce + %) )
         (interleave ks )
         (apply hash-map ))))
```

Voltando à questão dos campos que contêm valores não numéricos, e, que, portanto, propiciariam erro na aplicação da soma, eles podem fazer parte da planilha, desde que sejam filtrados antes da totalização. A chave é visualizar o *map* de linha como uma *seq* de *keys* e valores. Os valores podem ser testados quanto ao tipo (e, portanto, filtrados); as *keys* resultantes são o ponto de partida para o corpo da função `processa`.

Antes de editar a função, é recomendável experimentar as ideias no REPL:

repl

```
(def planilha [{:desc "caneta" :est 1000 :vend 600 :dev 15}
               {:desc "borracha" :est 5000 :vend 750 :dev 3}])
;=> #'lab.planiha/planilha
(first planilha)
;=> {:desc "caneta", :est 1000, :vend 600, :dev 15}
(seq *1)
;=> ([:desc "caneta"] [:est 1000] [:vend 600] [:dev 15])
(filter #(number? (second %)) *1)
;=> ([:est 1000] [:vend 600] [:dev 15])
(map first *1)
;=> (:est :vend :dev)
```

`filter` contém uma chamada a `seq` embutida, portanto é dispensável invocá-la. Aqui ela favorece a visualização. A função modificada fica assim:

editor

```
(defn processa
  [p]
  (let [ks (->> (first p)
                (filter #(number? (second %)) )
                (map first ))]
    (->> (map #(map % p) ks)
         (map #(reduce + %) )
         (interleave ks )
         (apply hash-map ))))
```

É curioso notar que essa solução funcionaria com outras demandas, bastando trocar a soma por outra função. Essa possibilidade é fácil de implementar com a inclusão de um parâmetro a mais. Experimente executar `(processa max planilha)` no REPL em seguida.

editor

```
(defn processa
  [f p]
  (let [ks (->> (first p)
                (filter #(number? (second %)) )
                (map first ))]
    (->> (map #(map % p) ks)
         (map #(reduce f %) )
         (interleave ks )
         (apply hash-map ))))
```

Clojure possui a peculiar capacidade de, numa única definição de função, acomodar diversas aridades. Observe nas funções populares do próprio Clojure. Num REPL, digite `(source range)` . O corpo da *special form* `defn` aceita várias listas, cada uma correspondendo a uma aridade diferente (os vetores de parâmetros logo nos inícios indicam isso). Com esse mecanismo, pode-se criar uma invocação default para `(processa planilha)`, que, no caso, seria a soma.

editor

```
(defn processa
  ([p]
    (processa + p))
  ([f p]
   (let [ks (->> (first p)
                 (filter #(number? (second %)) )
                 (map first ))]
     (->> (map #(map % p) ks)
          (map #(reduce f %) )
          (interleave ks )
          (apply hash-map )))))
```

5.1 Exercício: *keys* informativas

Modifique a função `processa` de modo que o *map* resultante possua *keys* mais informativas. As *keys* devem incorporar também o nome de `f`. Exemplo:

```
(processa + planilha)
;=> {:+-est 6000 :+-vend 1350 :+-dev 18}
(processa max planilha)
;=> {:max-est 5000 :max-vend 750 :max-dev 15}
```

6. Referências

BIRD, RICHARD. **Thinking Functionally with Haskell**. Cambridge University Press, 2015.

FOGUS, MICHAEL. **Functional JavaScript**. O'Reilly, 2013.