programadores do paradigma funcional fazem cara feia para quem altera variáveis (eles nem usam esse nome, pois não querem que 'varie'), para quem usa print ou spit, para quem mexe numa coleção. também não gostam de globais variando. para tudo isso, eles criam novas cópias. seu mundo lembra um museu vivo de todas as versões existentes dos dados. pode parecer estranho, mas assim eles se tornam imunes aos males mais cruéis do mundo da programação.

1. Funções puras = ausência de efeitos colaterais = transparência referencial

Programas funcionais têm algo em comum com planilhas eletrônicas. Planilhas eletrônicas possuem campos a serem preenchidos. Para uma dada planilha eletrônica, que contém cálculos prontos dentro dela, os resultados obtidos dependem exclusivamente dos valores inseridos nos campos.

Essa simples propriedade é das mais importantes. Uma função *pura*, no jargão da programação funcional, encontra-se completamente definida por sua assinatura, isto é, pelos parâmetros de entrada e pelo valor de retorno.

A ideia se torna mais clara quando se pensa numa função que não é pura. Quando uma função realiza parte de sua ação por canais que não estão evidentes em sua assinatura, diz-se que ela possui ***efeitos colaterais*** *(side effects)*. Por exemplo, uma função pode alterar o valor de uma variável não local (possivelmente uma global). Esse acontecimento se dá pela "porta dos fundos". Quem lê um trecho de código em que a função é usada, e que não conhece o código fonte da função, não enxerga a ação de alteração da variável não local. Os problemas acontecem também quando há leitura de variáveis não locais. Caso a variável mude de valor, a função por consequência pode resultar em algo diferente para os mesmos valores como parâmetros. Mais uma vez, a leitura de um programa em que a função é usada tem opacidades (isto é, nem tudo está visível; a propriedade de não possuir opacidades é conhecida como *transparência referencial*).

Toda operação de entrada e saída é um tipo de efeito colateral. Imprimir em tela, receber dados digitados, ler e escrever em arquivos, gerar números aleatórios, etc. tornam as funções impuras.

> o ideal supremo da programação funcional é a presença mínima de funções impuras. com isso, o sistema se torna legível, compreensível, previsível e mais propenso a poder ser controlado intelectualmente.

Existem linguagens de programação funcionais que exigem uma disciplina de programação absolutamente pura, como a linguagem Haskell. **Mas o mundo real está repleto de efeitos colaterais**. Programas precisam interagir com usuários, precisam atualizar arquivos e bancos de dados, portanto não há como evitar efeitos colaterais.

Clojure é antes de tudo uma linguagem prática, ela possui mecanismos para a realização de efeitos colaterais. Entre linguagens funcionais puras num extremo e linguagens absolutamente liberais no outro, Clojure (assim como muitas linguagens funcionais) adota um meio termo: efeitos colaterais são permitidos, mas há uma cerimônia para que aconteçam, de modo que se sabe bem onde eles estão (confinamento). Em geral, um sistema deve ser constituído de um núcleo de funções puras e uma casca que concentra os efeitos colaterais (e que usa o núcleo puro).

2. Dados que não mudam = valores (e não têm estado)

Alterações são proibidas. Uma vez iniciada, uma variável sempre conterá o mesmo dado. Chamá-la de *variável* é inapropriado, mais correto seria chamá-la de *valor*. Uma coleção (array, por exemplo) não pode receber valores novos, nem ter valores extraídos ou alterados. Aquela coleção é imutável. No entanto, nada impede que se crie uma nova coleção a partir dela com as alterações desejadas, como se fosse uma nova versão. (Isso é o que acontece com as strings em Java.)

A descrição acima enuncia uma das propriedades mais notáveis da programação funcional: a imutabilidade de dados, que se estabeleceu muito mais a partir da família de linguagens funcionais com raiz em ML (SML, OCaml, Haskell) do que da família Lisp. Clojure, embora um Lisp, é totalmente ML nesse quesito.

Se uma função recebe um dado (pode ser uma coleção) para realizar o que seria uma alteração nele, em Clojure, ela cria um novo dado. O dado original é preservado.

Dados imutáveis não têm estado. Não existe algo como "ele estava assim, agora ele está assado, e daqui a pouco será outra coisa". Não!

Dados imutáveis são valores. `[1,2,3]` é um valor de lista. `[10,2,3]` é outro valor, não pode ser pensado como um dado que foi alterado. Os dois valores estão disponíveis na memória.

Clojure é eficiente. Se uma coleção tem 1.000.000 de inteiros, não vai ser preciso alocar mais 1.000.001 de inteiros para gerar a cópia em que um novo valor é inserido. Como tudo é imutável, o novo valor se liga, em memória, aos dados do valor original.

Dados imutáveis são persistentes. Eles existem para sempre na memória (≠ persistência em disco).

Esse modelo de programação, cuja imagem mental praticamente elimina cálculos sobre estados e em que tudo são como valores matemáticos, propicia mais clareza a respeito do que se passa na execução do programa.

3. Efeitos colaterais em Clojure

Clojure possui diversos mecanismos de manipulação de células de memória (chamados, em geral, de "referências"), mas a discussão aqui se limita a apenas um deles. Um *atom* em Clojure é como uma caixa. Enquanto caixa, ela é imutável: é sempre a mesma caixa. No entanto, como caixa, pode ser aberta e fechada, e um valor pode ser posto ou tirado de lá de dentro.

repl

```
(def power (atom 100))
;=> #'user/power
(deref power)
;=> 100
@power
;=> 100
(reset! power 200)
;=> 200
@power
;=> 200
(swap! power inc)
;=> 201
@power
;=> 201
```

A função `atom` cria o *atom* com um valor inicial. A função `deref`, ou a literal correspondente `@`, abre a caixa para ver o que está lá dentro. `reset!` põe um valor na caixa, independentemente do que havia antes. O sinal de exclamação indica efeito colateral. Toda função que tem uma exclamação pode realizar efeitos colaterais. Isso é uma convenção. (Clojure traz essa convenção da linguagem Scheme, um dos Lisps mais influentes.) `swap!` usa uma função para atualizar o que está na caixa (isto é, ler, calcular e escrever).

Atoms são atômicos (daí o nome), pois, se uma outra *thread* tentar acessar o átomo durante a execução de `reset!`, `swap!` e outras (pesquise sobre `compare-and-set!`) numa situação conflituosa, a linguagem resolve a situação. (A ênfase deste material é a programação funcional; para discussão sobre concorrência em Clojure, confira *Clojure Programming* de Chas Emerick et al., O'Reilly).

3.1 Exercício: função impura

Explique o conceito de função impura a partir da seguinte sessão de REPL.

repl

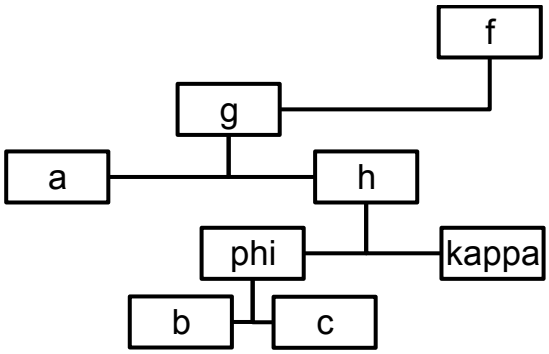
```
(def glob (atom 10))
;=> #'user/glob
(defn f [x]
  (let [v @glob]
    (if (> v 50)
      (inc x)
      (dec x))))
;=> #'user/f
(f 3)
;=> 2
(reset! glob 100)
;=> 100
(f 3)
;=> 4
```

4. Morfologia das funções puras e impuras

Funções puras têm uma forma característica! E funções impuras também!

As funções puras mais básicas são como funções compostas na matemática. O resultado da função é a invocação de uma função. Esta, por sua vez, pode ter vários parâmetros. Cada um deles pode corresponder à invocação de uma função. E, assim, por diante. A composição das funções constitui uma árvore. Mais importante de tudo: essa árvore tem uma só raiz. Ou seja, o corpo de uma função pura é a invocação de uma só função! Não importa que essa função invoque outras nos parâmetros.

```
(def f [a b c]
  (g a (h (phi b c) (kappa))))
```



Os condicionais (`if`, `cond`, etc.) têm papel decisivo na morfologia das funções puras, pois eles nos remetem às funções matemáticas definidas por vários segmentos:

$$f(x,y) = \begin{cases} -7, x = 0 \\ y - x, x < 0 \\ x > 0 \begin{cases} y, y = 2x \\ -y, y \neq 2x \end{cases} \end{cases}$$

```
(defn f [x y]
  (cond
    (= x 0) -7
    (< x 0) (- y x)
    (> x 0) (if (= y (* 2 x))
              y
              (neg y))))
```

Dispor assim, lado a lado, uma função matemática e sua implementação em Clojure ajuda a estabelecer uma correspondência, mas é preciso salientar que as funções em Clojure costumam tratar de strings, dados, *maps*, arquivos, *e-commerce, business*, ...

```
(def web-crawler [urls]
  (when (urls)
    (if (> (count urls) 1000)
      (let [subs (partition urls)]
        (dispatch (load-balance subs)))
      (dispatch urls))))
```

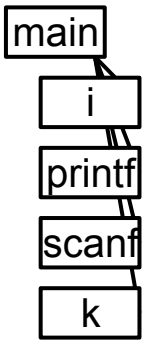
O exemplo acima é inventado, mas verossímil. O que importa é ver as ramificações similares às das funções matemáticas. Esse é o ponto: uma função pura pode estar repleta de ramificações, mas só um galho é executado, e nele há uma invocação de função única, como visto anteriormente. A função é como uma árvore de ramificações de caminhos de escolha (condições), mas sua execução é só o que está em alguma ponta.

Já as funções impuras podem manifestar uma característica marcante: passos. Enquanto a função pura acha uma função única para executar, a função impura executa diversos passos. Cada passo é uma etapa da função, e a ordem é essencial, pois cada passo produz efeitos que podem ser usados pelo seguinte. Tudo nessa programação é estado e efeitos. Na visão de árvore, cada passo é uma raiz.

> Tecnicamente falando, funções puras são apenas *expressões*. Expressões são *avaliadas* (`eval`) e cada uma resulta num valor. Funções impuras possuem *statements*, que produzem efeitos (mudanças nos estados das coisas). *Statements* podem conter expressões (como as aritméticas, por exemplo).

A programação funcional ideal é uma programação sem *statements*, sem efeitos, sem estados, sem passos.

```
void main(int c, char* a[]){
  int i = 0;
  printf("%d", i);
  scanf("%d", &i);
  float k = i * 3.14;
}
```



5. Elementos para execução de efeitos em Clojure

Clojure não é uma linguagem funcional pura. Há diversos elementos na linguagem para a execução de efeitos. Entre eles estão: `do`, `dorun`, `doall`.

Com `do`, é possível definir blocos. Blocos são justamente coleções de passos. Todos são executados para obter efeitos, mas apenas o último passo do bloco define o resultado do bloco. Em outras palavras, o resultado do bloco é o resultado de seu último passo. Portanto, os passos anteriores só valem por seus efeitos.

repl

```
(def power (atom 7))
;=> #'user/power
(do
  (println "hi")
  (swap! power dec)
  (+ @power 10))
;hi
;=> 16
```

As *forms* `fn`, `let` e `defn` chamam `do` implicitamente. Isso quer dizer que é possível escrever "passos" no corpo dessas *forms*.

Sobre `dorun` e `doall`, a sessão de REPL a seguir ilustra as diferenças.

repl

```
(def alpha (atom 1))
;=> #'user/alpha
(defn phi! [x]
  (do
    (println @alpha)
    (swap! alpha inc)
    (+ 100 x)))
;=> #'user/phi
(def m (map phi! [10 30 70]))  ; LAZY !!!
;=> #'user/m
(def n (map phi! [10 30 70]))  ; LAZY !!!
;=> #'user/n
(dorun m)
;1
;2
;3
;=> nil
(doall n)
;4
;5
;6
;=> (110 130 170)
```

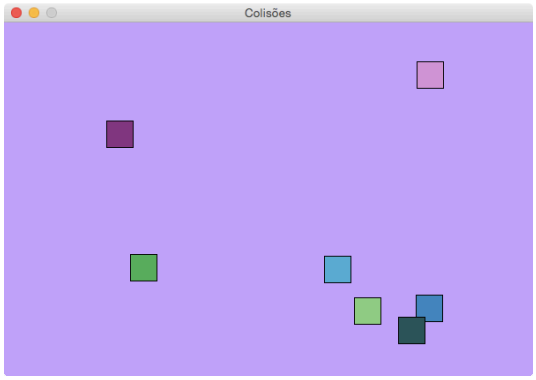
`m` e `n` são coleções obtidas com a função `phi!` aplicada ao vetor `[10 30 70]` através de `map`. `m` e `n` são o mesmo valor. Devido à *lazy evaluation* de `map`, nada acontece no momento em que `m` e `n` são definidas. (Essa intuição é prejudicada com o uso constante do REPL, pois o `print` [o P em REPL] obriga a realização de uma sequência *lazy* criada com `map`, `filter` ou `reduce` para exibi-la.) Para forçar a execução da sequência, existem `dorun` e `doall`. A diferença entre eles é evidente: `dorun` é usado só pelos efeitos, `doall` além dos efeitos, retém os valores. Os termos *run* e *all* têm a ver com o que acontece.

6. Detecção de colisões

Este exemplo-exercício é 99% voltado para programação funcional. A presença do gerenciamento de estado é um detalhe (sem o qual o programa não funcionaria!). O assunto volta a ser compor funções para atingir um objetivo de programação, mas a noção de estado se faz sentir na observação do funcionamento do programa, que depende de um único *atom* (ainda que escondido na biblioteca empregada).

O programa é uma animação gráfica em 2D. Figuras simples se movem na tela ricocheteando nas paredes.

- O primeiro objetivo é entender o programa.
- O segundo objetivo é modificar o programa, incluindo a funcionalidade de detecção de colisões.



O programa usa a biblioteca `quil`[1]. Ela é uma adaptação do sistema Processing, uma ambiente de programação para aplicações gráficas feito em Java. Ele possui uma linguagem de programação própria (um misto de Java e *scripts*) e uma extensa biblioteca de funções voltadas para desenho gráfico, animações e multimídia. Com `quil`, o programador Clojure tem à disposição todo o arsenal da biblioteca Processing. (Atenção! Não é um reimplementação! É o próprio Processing, que é Java, funcionando por baixo.)

Modifique o arquivo `project.clj` do projeto `lab`, para que o Leiningen traga Processing e `quil` para seu sistema. O trecho de dependências deve ficar assim (pesquise as versões atuais no site `clojars`) :

trecho do arquivo project.clj

```
:dependencies [[org.clojure/clojure "1.7.0"]
               [quil "2.2.6"]]
```

6.1 Preâmbulo: novas funções empregadas no programa (`update-in`, `comp`, `for` e `doseq`)

A função `update-in` é especializada em *maps*. Ela recebe um *map*, uma vetor com a *key*, uma função e seus parâmetros. Em seguida, ela lê o valor da *key* no *map* recebido, aplica a função e parâmetros a ela, e cria um novo *map* com esses valores. Ex: `(update-in {:nome "alpino" :preco 4.45} [:preco] + 2)` resulta em `{:nome "alpino", :preco 6.45}` . O vetor da *key* é, na verdade, um *path*. Se o *map* contivesse outros *maps* dentro, essa opção agilizaria a especificação. Ex.: o vetor `[:b :qq :zzz]` permite alterar o valor `0` em `{:a "foo" :b {:pp \i :qq {:zzz 0 :www 1}}}`

`comp` significa *composição de funções*. `(comp f g)` corresponde à expressão matemática $f \circ g$. Uma função definida como `(def mais-2 (comp inc inc))` quando aplicada em `(mais-2 100)` resulta em `102`.

É comum encontrar `doseq` nos exemplos sobre `quil`. Ela é melhor compreendida se apresentada juntamente com a função `for`.

Embora suscite a ideia de laço pela existência de elementos de mesmo nome em outras linguagens, `for` pouco tem a ver com isso. Em matemática, é possível definir conjuntos com base em outros conjuntos. Por exemplo, seja $A = \{1, 3, 5, 17, 23, 328\}$; então é possível definir $B = \{x \in A \mid x < 10\}$. Em inglês, a definição de B é chamada de *set comprehension* (evitar a tentação de traduzir). No contexto da computação, especialmente das linguagens de programação, algo similar tem surgido. Haskell é a linguagem que ficou marcada pela divulgação da ideia, que foi denominada de *list comprehension*. Pois bem, `for` é o mecanismo de

[1] http://quil.info. Destaque para a introdução: http://nbeloglazov.com/2014/05/29/quil-intro.html. Há uma versão de Processing em JavaScript (Processing.js), criada pelo mesmo autor da jQuery. ClojureScript é usada com ela na *web*.

list comprehension em Clojure. Os exemplos matemáticos acima seriam escritos com `for` da seguinte forma: `(def A [1 3 5 17 23 328])`, `(def B (for [x A :when (< x 10)] x))`. Execute num REPL:

repl

```
 (for [a (range 1 10)
       b (range a)
       :when (= 10 (+ a b))]
 [a b (- a b)])
;=> ([6 4 2] [7 3 4] [8 2 6] [9 1 8])
(for [x (range) :while (< x 5) :when (even? x)
      y (range 10) :while (< y x)]
  [x y])
;=> ([2 0] [2 1] [4 0] [4 1] [4 2] [4 3])
```

`for` gera uma sequência *lazy*. doseq tem exatamente os mesmos parâmetros de for, mas ela é mais apropriada para executar operações com efeitos colaterais, como as funções de desenho de `quil`.

repl

```
(doseq [a (range 1 10)
      b (range a)
      :when (= 10 (+ a b))]
  (println (- a b)))
;2
;4
;6
;8
;=> nil
```

6.2 Estrutura de um *sketch* em `quil` com o middleware funcional

Um programa (desenho ou animação) em Processing é chamado de *sketch* (esboço). `quil` não foge à estrutura de um programa em Processing. A macro `defsketch`, localizada no final do programa, contém a configuração. Os campos mais importantes são os nomes das funções de inicialização (*:setup*) e de desenho (*:draw*).

O *middleware* funcional de `quil` é opcional (e pouco usual) e traz mudanças específicas ao *sketch*.

- Internamente, `quil` mantém o estado do programa num *atom* oculto.
- O estado é inicializado pelo resultado de retorno da função configurada em `:setup` de `defsketch`.
- O estado é passado como parâmetro para a função configurada em `:draw`.
- Uma função adicional precisa ser especificada no campo `:update` para atualização do estado antes de cada desenho. A função é invocada automaticamente antes de chamar a função em `:draw`. A função recebe o estado como parâmetro, e o seu resultado é o novo valor do estado.

6.3 O programa collisions.clj

- O programa começa com a criação de constantes como quantidade de sprites, dimensões da tela e margens.
- Uma função auxiliar para geração de números aleatórios num intervalo que começa em um ponto diferente de zero foi criada.
- Sprites são gerados aleatoriamente. Seus dados compreendem posição, direção, cor e velocidade.
- A função `setup` inicia o *sketch*. Ela configura, como efeito colateral, a *frame rate*. Seu resultado é uma sequência de *maps* representando os sprites gerados.
- A função `my-update` recebe a sequência de sprites e atualiza cada um independentemente através de `map`. Ela aplica a composição de duas funções, uma de movimento e outra de rebatida (`bounce`).
- Observe o *destructuring* nas entradas das funções de movimento e rebatida.
- Experimente trocar `bounce` por `pass` e observe o efeito.
- O papel da função que desenha um sprite está nos seus efeitos colaterais. Quando aplicada à coleção, exige o emprego de `dorun`. Crie uma outra versão com `doseq`.

arquivo collisions.clj na pasta lab

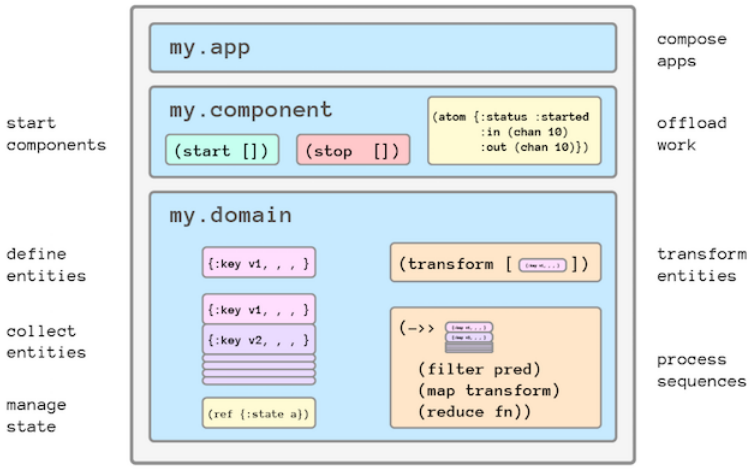
```
(ns lab.collisions
  [:require [quil.core :refer :all]
            [quil.middleware :as m]])
(def quant 7)
(def the-w 600)
(def the-h 400)
(def margin-l (+ 0 50))
(def margin-r (- the-w 50))
(def margin-u (+ 0 50))
(def margin-d (- the-h 50))
(defn rand-lower-upper [lower upper]
  (+ lower (rand (- upper lower))))
(defn generate-sprite []
  (hash-map
    :x (rand-lower-upper margin-l margin-r)
    :y (rand-lower-upper margin-u margin-d)
    :dx (rand-lower-upper -1.0 1.0)
    :dy (rand-lower-upper -1.0 1.0)
    :r (rand-int 256)
    :g (rand-int 256)
    :b (rand-int 256)
    :v (rand 4)))
(defn setup []
  (frame-rate 60)
  (take quant (repeatedly generate-sprite)))
(defn move [{:keys [dx dy v] :as sprite}]
  (-> sprite
      (update-in [:x] + (* v dx))
      (update-in [:y] + (* v dy))))
(defn bounce [{:keys [x y dx dy] :as sprite}]
  (let [fx (if (or (and (< x margin-l) (neg? dx))
                   (and (> x margin-r) (pos? dx))) -1 1)
        fy (if (or (and (< y margin-u) (neg? dy))
                   (and (> y margin-d) (pos? dy))) -1 1)]
    (-> sprite
        (update-in [:dx] * fx)
        (update-in [:dy] * fy))))
; alternate behavior to bounce : when sprite touches wall, it disappears and appears in
the other side
(defn pass [{:keys [x y dx dy] :as sprite}]
  (let [mrgn-l (if (and (< x margin-l) (neg? dx)) (assoc sprite :x margin-r) sprite)
        mrgn-r (if (and (> x margin-r) (pos? dx)) (assoc mrgn-l :x margin-l) mrgn-l)
        mrgn-u (if (and (< y margin-u) (neg? dy)) (assoc mrgn-r :y margin-d) mrgn-r)
        mrgn-d (if (and (> y margin-d) (pos? dy)) (assoc mrgn-u :y margin-u) mrgn-u)]
    mrgn-d))
(defn my-update [state]
  (map (comp move bounce) state))
(defn draw-sprite [{:keys [x y r g b]}]
  (fill r g b)
  (rect (- x 15) (- y 15) 30 30))
(defn draw [state]
  (background 200 150 255)
  (dorun (map draw-sprite state)))
(defsketch collision
           :title "Colisões"
           :size [the-w the-h]
           :setup setup
           :draw draw
           :update my-update
           :middleware [m/fun-mode])
```

6.4 Detectando colisões

Em `my-update`, após mover e rebater, remova os sprites que se chocam. Para isso, primeiramente gere o conjunto de todos os pares de sprites (o *produto cartesiano*). `for` é excelente para isso. Em seguida, filtre (remova) aqueles que tiverem muito próximos entre si. Remova-os dos pares e reúna-os em um conjunto. Devolva os remanescentes como retorno da função.

7. Conclusão

A figura abaixo é encontrada em sites de divulgação do livro *Applied Clojure*, como um resumo do conteúdo (embora o livro em si não contenha a figura).



tasks and structure of a Clojure application

As etapas mostradas na parte inferior foram cobertas de alguma forma:

- Definem-se entidades de domínio criando-se *maps*.
- Reúnem-se entidades de domínio em vetores, outros *maps*, conjuntos, listas.
- Gerencia-se estado com *atoms*.
- Processam-se sequências com `filter`, `map` e `reduce`.

Embora pareça simplório à primeira vista, esse pequeno conjunto de conceitos cobre uma vasta gama de sistemas e aplicações em computação. As ferramentas básicas da escola funcional para empreitadas de desenvolvimento de software estão postas na mesa. Mãos à obra!