1. Clojure, assim como qualquer Lisp, favorece o paradigma de programação funcional. Isso quer dizer que um programa é uma coleção de funções, e que cada função é uma combinação de outras funções. A vantagem do paradigma funcional é uma relativa separação das responsabilidades. Um bom programa funcional contém funções específicas para cada objetivo e sub-objetivo referentes ao programa, procurando misturar ao mínimo aqueles de propósitos diferentes. A partir disso, as função são combinadas de maneira apropriada a atender as especificações do programa.

Além disso, é desejado que as funções demonstrem o máximo possível características de funções matemáticas. A característica suprema nessa direção é a transparência referencial, segundo a qual poder-se-ia substituir uma aplicação da função por sua fórmula (código), feitas as devidas substituições dos parâmetros. Ou seja, se uma função `f` em C é definida como `int f(int x) {return x + 10;}`, ela pode ser substituída em `int i = f(2);` (aqui temos uma aplicação de `f`) sem problemas: `int i = 2 + 10;` Uma função com transparência referencial é totalmente compreensível a partir do seu próprio código.

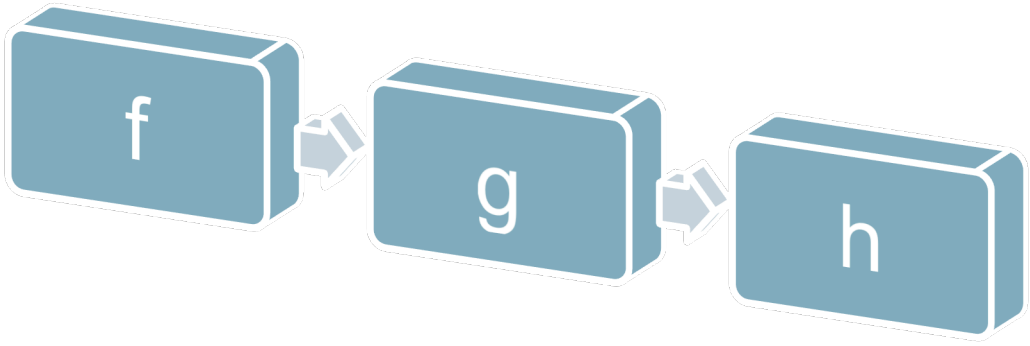
Uma função não possui transparência referencial, por exemplo, quando ela manifesta efeitos colaterais. Uma função com transparência referencial deve concentrar todos os seus resultados no valor de retorno. Mas, em computação, isso nem sempre acontece. Funções podem alterar valores de variáveis globais ou realizar operações de I/O, como imprimir valores em tela ou salvar arquivos. Em todos esses casos, a função realiza algum efeito que não está evidente no seu valor de retorno. Esses são chamados efeitos colaterais. Quando a função realiza efeitos colaterais, o seu sentido não fica evidente pelo código, pois os valores das variáveis globais e dos estados de I/O não estão presentes no código em si mesmo.

No jargão da programação funcional, funções sem efeitos colaterais são chamadas *puras*. Existem linguagem radicais no sentido de exigir que todas funções sejam puras, como é o caso da linguagem Haskell. Clojure, assim como qualquer Lisp, não é pura. É possível realizar efeitos colaterais em Clojure, no entanto, enconraja-se que isso se limite a uns poucos pontos do programa.

A promessa da programação funcional é permitir que o programador tenha maior controle intelectual sobre o programa que ele mesmo criou, ou seja, ele deve conseguir formar uma imagem mental mais clara do funcionamento do programa. Nem sempre, em cursos introdutórios, o aluno consegue perceber essa característica. Uma exposição abrangente sobre as vantagens do paradigma funcional na percepção de programadores Java pode ser encontrada no curto livro (150 p.) de Neal Ford, **Functional Thinking**: Paradigm Over Syntax, O'Reilly, 2014. Outra referência é **Functional JavaScript**, de Michael Fogus, O'Reilly, 2013.

A construção de programas no paradigma funcional é qualitativamente diferente em comparação com linguagens usuais. A "lógica de programação" é outra. Aprender a programar no paradigma funcional, de alguma forma, exige que se esqueça o que se sabe a respeito de programação, como se a pessoa voltasse a ser um iniciante.

2. Fogus apresenta uma interessante imagem para um sistema construído segundo os princípios funcionais: uma linha de montagem que recebe matéria-prima numa ponta e entrega produtos montados na outra ponta. Ao longo do caminho, ela atravessa por estágios intermediários que realizam etapas específicas do processo. Cada estágio é uma função. A cena remete também à composição de funções na matemática.



$$h\Big(g\big(f(x)\big)\Big) = (h \circ g \circ f)(x)$$

3. Se se submete um número a um REPL, ele devolve o número. O mesmo acontece com strings. Se se submete um símbolo a um REPL, ele devolve um valor associado a um símbolo. Exemplos[1]:

[1] A notação usada para representar a interação é inspirada em **The Joy of Clojure**, 2.ed., de Michael Fogus e Chris Houser, editado pela Manning em 2014. Ela facilita o "copiar e colar" direto no REPL.

```
3.4
;-> 3.4

"oi"
;-> oi
```

4. A forma de agregar dados em Lisp é usar listas. A sintaxe literal de uma lista em Lisp começa e termina com parênteses, tendo entre eles os elementos separados por espaço em branco.

```
(1 "a" 3.14 7 100)
```

5. Lisp é uma linguagem *homoicônica*, isto é, os dados e os programas em Lisp têm a mesma representação. Um programa em Lisp é um conjunto de listas.

A operação mais básica em Lisp é a aplicação[2] de uma função. E justamente ela é representada por uma lista cujo primeiro elemento é a função e os demais, se houver, são parâmetros da função. A função, quase sempre, é representada por um símbolo. Exemplo:

```
(inc 3)
;-> 4
```

A execução da instrução acima se dá da seguinte forma: o avaliador reconhece uma lista e, imediatamente, se prepara para executar uma função. Em seguida ele avalia os elementos da lista. O primeiro é um símbolo (`inc`), cujo valor associado é uma função. O segundo é 3, que vale 3 mesmo. Por fim, ele invoca a função passando o parâmetro 3 e retorna o resultado.

Comparando a aplicação de uma função em C e em Clojure/Lisp :

```
f(2,3)   // C

(f 2 3)  ; Clojure
```

6. Exemplos da aritmética ajudam a entender e fixar os conceitos:

```
(+ 2 3)
;-> 5

(* 4 (- 12 5))
;-> 28
```

A operação de adição (primeiro exemplo) não tem nada de diferente em relação a qualquer outra função. O caracter "+" é um símbolo como qualquer outro, e, como símbolo, está associado a uma função (nesse caso, a função de soma). E assim é com as demais operações.

No segundo exemplo, existe uma lista dentro de outra. A regra continua a mesma: o avaliador aplica a função depois de avaliar os parâmetros; como um dos parâmetros é uma lista, ele o avalia da mesma forma (o primeiro elemento/símbolo é uma função e o restante são parâmetros para ela).

Em outras palavras, enquanto em linguagens de programação em geral, a aritmética é um caso especial tratado pelo compilador/interpretador, em Clojure/Lisp ela faz parte da biblioteca de funções e mais nada. Tudo funciona perfeitamente bem com as regras que já existem para funções em geral.

7. Surge, então, um problema: se o avaliador enxerga qualquer lista, mesmo dentro de outra lista, como uma aplicação de uma função, como fazer para passar uma lista como dado em vez de aplicação? Exemplo:

```
(cons 3 (10 17 100))
```

A função `cons` é usada para inserir um elemento numa lista. A intenção da instrução acima era inserir o elemento `3` na lista `(10 17 100)`. No entanto, o REPL responde com erro quando recebe essa instrução. Isso porque ele tenta executar a lista `(10 17 100)` como uma aplicação de função. Mas, logo de início, quando

[2] Isto é, o *uso* de uma função. O termo "aplicação" é uma influência do jargão da matemática. A programação funcional é, às vezes, denominada de *programação aplicativa*.

ele tenta avaliar o símbolo inicial (`10`), não obtém uma função. A questão é que a lista está na instrução não como uma aplicação de função, mas um dado puro.

A solução em Clojure/Lisp é "desligar" o avaliador no trecho em que a lista está. Para isso existe o mecanismo `quote` (citação, ou, no popular, "entre aspas").

```
(cons 3 (quote (10 17 100)))
;-> (3 10 17 100)
```

Esse recurso é tão usado, que foi criado um atalho para ele: basta colocar um apóstrofo (aspas simples) na frente da lista. É sempre um apóstrofo só. Não precisa fechar.

```
(cons 3 '(10 17 100))
;-> (3 10 17 100)
```

8. Existe um mecanismo para criar símbolos, chamado `def`. Ele requer o símbolo sendo definido e o valor associado ao símbolo. O sistema adiciona esse símbolo à sua base de dados ("caderneta") de símbolos.

```
(def x 15)
;-> #'user/x

x
;-> 15

'x
;-> x
```

Observe que o `quote` desliga o avaliador também no caso dos símbolos.

9. Nem tudo parece obedecer ao modelo de avaliação apresentado. Por exemplo:

```
(if (> 3 2) (+ 10 20) (1 2 3))
;-> 30
```

A presença da lista `(1 2 3)` deveria causar um erro no avaliador. Mas isso não acontece, o que é um sinal de que ela não foi avaliada, contrariando o modelo. (A propósito, troque > por < e observe o resultado.)

A explicação para isso é que o recurso `if` não é uma função. Em outras palavras, nem tudo que vem entre parênteses é recebido como aplicação de função. Existem, pelo menos, dois outros casos: as macros e as *special forms*.

Macros são transformações de código. Uma forma de entender a diferença de uma macro para uma função é pensar em que momento cada uma é processada. Macros não são executadas. Elas operam antes da "compilação" do programa, transformando o código naquilo que vai ser efetivamento executado pelo avaliador. Por exemplo, a macro `when` é examinada abaixo com o recurso `macroexpand-1`:

```
(macroexpand-1 '(when (> 2 1) (- 100 10)))
;-> (if (> 2 1) (do (- 100 10)))
```

Special forms são recursos tão básicos que não poderiam ser implementados como funções na própria linguagem, portanto eles são implementados "em *assembly*"[3] (isto é, já estão embutidos prontos no avaliador).

Como as macros e as *special forms* não seguem o modelo de avaliação de função, quando se usa alguma delas nem sempre se manifesta o problema de uma lista ser avaliada como aplicação de função, e, portanto, nem sempre é exigido o `quote`.

Como saber se uma lista será avaliada como função, macro ou *special form*? O avaliador sempre começa pelo primeiro elemento da lista, que costuma ser um símbolo. Esse símbolo é avaliado na base de dados (caderneta), e ali consta o tipo do valor associado (isto é, função, macro ou *special form*). O programador pode consultar essa informação na documentação, usando a macro `doc`.

```
(doc if)
```

[3] Nesse caso, seria mais correto dizer "em Java" ou "em *bytecodes* Java" do que "em *assembly*".

```
;->
;-> -------------------------
;-> if
;->   (if test then else?)
;-> Special Form
;->   Evaluates test. If not the singular values nil or false,
;->   evaluates and yields then, otherwise, evaluates and yields else. If
;->   else is not supplied it defaults to nil.
;->
;->   Please see http://clojure.org/special_forms#if
;-> nil

(doc when)
;-> -------------------------
;-> clojure.core/when
;-> ([test & body])
;-> Macro
;->   Evaluates test. If logical true, evaluates body in an implicit do.
;-> nil
```

Há uma indicação explícita dos tipos dos comandos (destacados em amarelo). (Observação: se `doc` não fosse uma macro, o seu uso correto seria `(doc 'if)`, `(doc 'when)`.)

Experimente:

```
(find-doc "sort")
```

Troque `"sort"` por qualquer string do seu interesse.

10. Em termos de sintaxe, Lisp não tem muito mais a acrescentar do que as poucas regras vistas até aqui. Isso explica porque se diz que Lisp é uma linguagem "sem sintaxe" ou com uma "sintaxe mínima". Não há, por exemplo, regras de precedência de operadores em Lisp, pois não é preciso. Por outro lado, qualquer expressão de médio porte possui muitos parênteses.

Clojure manifesta de alguma maneira essas propriedades do Lisp, mas Clojure possui um pouco mais de sintaxe especial, a começar pelas literais para vetores, tabelas hash (*maps*), keywords, conjuntos, funções anônimas, etc. como será visto.

11. Lisp admite funções anônimas, muitas vezes chamadas de lambda, em referência ao cálculo lambda. Funções anônimas podem ser pensadas como o essencial de uma função: seus parâmetros e sua fórmula (código). A notação **em Clojure** é exemplificada assim:

```
(fn [x y] (+ x y))
```

A função acima pode ser aplicada conforme as regras de aplicação de uma função (observe que função anônima ocupa a primeira posição da lista):

```
((fn [x y] (+ x y)) 10 37)
;-> 47
```

No entanto, a maneira mais usual de declarar uma função é através da macro `defn`:

```
(defn sum [x y]
  (+ x y))
;-> #'user/sum
```

A expansão da macro mostra que ela é basicamente uma combinação das *special forms* `def` e `fn`.

```
(macroexpand-1 '(defn sum [x y] (+ x y)))
;-> (def sum (clojure.core/fn ([x y] (+ x y))))
```

A macro `defn` possui outros recursos, como string de documentação, pré e pós-condições, etc. Confira a documentação.

Clojure, diferentemente de outros Lisps, possui uma literal de função anônima: `#(+ %1 %2)`, `#(cons % '())`. No primeiro caso, temos uma função anônima com dois parâmetros, representados como `%1` e `%2`. No segundo caso, há um único parâmetro, representado como `%`.

12. *Namespaces* (espaços de nomes). O problema da colisão de nomes de variáveis e funções em linguagens de programação costuma ser tratado com alguma forma de modularização (módulos, pacotes, etc.). Em Clojure, isso é chamado de *namespace*.

Há vários recursos para manipulação de *namespaces*. Um dos mais usados consiste em iniciar o arquivo com o programa com a macro `ns`. Além da declaração do nome do *namespace*, a macro ns comporta a declaração de dependência de outros *namespaces* ou pacotes Java.

```
(ns application
  (:require [foo.bar]
            [foo.baz :as baz]))
```

Na expressão acima, o *namespace* `application` é ativado caso ela já exista, ou, senão, é criado. Além disso, dentro deles são carregados os *namespaces* `foo.bar` e `foo.baz`. Para usar a função hipotética `ff` do primeiro namespace, é preciso escrever seu nome qualificado (completo): `foo.bar/ff`. Já temos visto nomes qualificados nos resultados de `def` e `defn`: `user/sum` (nome `sum` no *namespace* `user`; `user` é o *namespace* default no REPL). Para usar a função hipotética `hh` no segundo *namespace*, pode-se opcionalmente usar o apelido `baz` definido por `:as : baz/hh .`

Uma alternativa para `:require` é `:use`. Com ela, todas as funções do *namespace* importado podem ser usadas diretamente, isto é, sem o nome qualificado. Pesquise sobre as opções `:only`, `:refer`, `:exclude`. Para importar pacotes Java, existe a opção `:import`.

No REPL, após carregar o arquivo com o *namespace* `application`, use a *special form* `in-ns` para começar a usar as funções do *namespace*: `(in-ns 'application)`. Observe que `in-ns` não carregaria o próprio Clojure se usada para criar o *namespace* (isto é, sem o uso prévio do `ns`).

Digite `*ns*` no REPL. Nesse caso, `*ns*` é uma variável global. É comum em vários Lisps, denotar variáveis globais com asteriscos no começo e no fim do nome da variável.

13. Interoperabilidade com Java. Existem sintaxes especiais em Clojure para criar objetos Java de qualquer biblioteca carregada na máquina virtual e no *namespace*. Por default, o pacote `java.lang` está disponível. Por exemplo, para criar um objeto da classe `java.util.Date`, basta usar a *special form* `new`, ou, alternativamente, escrever o nome da classe no início da lista, terminando-o com um ponto:
`(new java.util.Date)` ou `(java.util.Date.)`

Além disso, é possível invocar qualquer método em qualquer objeto com a seguinte sintaxe: escrever o nome do método precedido por um ponto no começo da lista, seguido do objeto e dos parâmetros:

```
(def v "oi")
(.toUpperCase v)
;-> "OI"
```

O exemplo acima funciona porque as strings em Clojure também são strings em Java.

14. Estruturas de dados. Em Lisp, a estrutura de dados suprema é a lista. Isso não quer dizer que não existam outros tipos, porém eles entram pela porta do fundo da linguagem. Existem vetores em Lisp? Sim, mas, para usá-los, deve-se digitar muita coisa, assim como acontece em Java com respeito às classes List, Vector, etc. Em Clojure, há quatro tipos diferentes de estruturas de dados de fácil utilização:

Tipos →	Lista	Vetor	Hash Map	Conjunto
Notação	(1 2 3)	[1 2 3]	{:a 1 :b 2}	#{1 2 3}
Objetivo	Programas	Dados	Tabelas	Sem repetição

Todos os tipos de estruturas de dados possuem uma sintaxe literal em Clojure.