a forma mais categórica de programar funcionalmente envolve o uso de recursão. programas assim escritos manifestam as propriedades essenciais do paradigma funcional: ausência de efeitos colaterais e semântica declarativa. um curso tradicional de programação funcional começa por recursão. map, filter e reduce podem ser definidos recursivamente. a chave para entender a técnica de programação funcional com recursão em listas é 'pensar como a máquina'.

1. Pensar como a máquina

O número 7 pertence à coleção (4 0 -2 3 9 6 20 -4 7 13 8) ?

Para um ser humano, essa operação é de uma certa qualidade. Seres humanos possuem grandes poderes cognitivos.

Para uma máquina, a história é diferente. (Máquina pode ser um aplicativo, um interpretador/compilador, um processador, etc.) A máquina só "vê" um número de cada vez. Seria algo como

Dentro de uma função recursiva sobre lista o elemento visível da vez seria `(first coll)` . Sobre esse elemento pode-se operar (fazer comparações, operações aritméticas, operações em listas, etc.)

O que fazer com os demais elementos, que não estão visíveis ainda? Antes de responder, é preciso dizer que esses elementos aparecem como `(rest coll)`, ou seja, o "resto da lista". **O "pulo do gato" da programação com recursão é "fazer de conta" que a função já foi aplicada a esse resto e o resultado disto está disponível agora na função.**

Anatomia de uma função recursiva em listas (primeiro esboço):

```
(defn function [coll (e outros)]
    ,,,
    ,,, (first coll) ,,,
    ,,,
    ,,, (function (rest coll)),,,
)
```

Segundo ponto da programação recursiva em listas: a recursão precisa parar em algum momento. O mais comum é que isso aconteça quando a lista for nula. Em Scheme, o teste é feito assim: `(if (null? col) ,,,)`. Em Clojure: `(if (empty? col) ,,,)`.

Anatomia de uma função recursiva em listas (segundo esboço):

```
(defn function [coll (e outros)]
    (if (empty? coll)
        (,,, retorno do caso base, quando a lista é nula*, ,,,)
        (,,, retorno do caso geral* ,,,)))
```

* ambos podem conter `(first coll)` ou `(function (rest coll))`, dependendo da lógica da função.

O terceiro ponto da programação recursiva em listas é observar o tipo do retorno da função. Ele define os *combinadores* usados na função.

Tipo de Retorno da Função	Retorno do Caso Base	Combinadores do Caso Geral
Booleano	`true false`	`and or ...`
Número	`0 1 ...`	`+ - * / ...`
Lista	`() [] ...`	`conj int cons ...`
String	`""`	`str join ...`

2. Exemplos

Função `member`
Parâmetros:
`x` – número
`coll` – sequência de números (lista ou vetor)
Retorno:
`true` – se `x` está em `coll` (pertence; em inglês: *member*)
Casos:
```
(member 3 '(4 6 0 8 -2 3 7 5))
;=> true
(member 1 '(4 6 0 8 -2 3 7 5))
;=> false
```

Discussão:
Temos dois pontos a analisar: `(first coll)` e `(member x (rest coll))`
Duas possibilidades: ou `x` é o `(first coll)`, ou ele está no `(rest coll)`
Em outras palavras, ou `(= x (first coll))` é `true`, ou `(member x (rest coll))` é `true`
Ou `x` é o primeiro elemento da lista, que a máquina pode examinar agora, ou ele está no resto da lista.

Caso base. Se a recursão chegar até a lista nula, foi porque não encontrou o `x`. Portanto, o resultado tem de ser `false`.

Resposta:
```
(defn member [x coll]
   (if (empty? coll)
      false
      (or (= x (first coll)) (member x (rest coll)))))
```

Função `length`
Parâmetros:
`coll` – sequência
Retorno:
quantidade de elementos em `coll`
Casos:
```
(length '(4 6 0 8 -2 3 7 5))
;=> 8
(length '())
;=> 0
```

Discussão:
`(first coll)` não é analisado em seu valor, basta contar 1.
`(length (rest coll))` corresponde ao tamanho do resto da lista. Deve ser considerado como pronto.
Assim, o resultado é obtido somando-se 1 a `(length (rest coll))` .

Caso base. A lista nula mede 0.

Resposta:
```
(defn length [coll]
   (if (empty? coll)
      0
      (+ 1 (length (rest coll)))))
```

Função `twice`[1]
Parâmetros:
`coll` – sequência de números (lista ou vetor)
Retorno:
sequência cujos respectivos elementos em relação a `coll` são dobrados
Casos:
```
(twice '(4 6 0 8 -2 3 7 5))
;=> (8 12 0 16 -4 6 14 10)
```

Discussão:
`(twice (rest coll))` já traz a sequência pronta para o resto da lista.
Então, basta inserir na frente o dobro de `(first coll)`.
Para inserir na frente de uma sequência, usar `cons`.

Caso base. A lista nula deve gerar a lista nula.

Resposta:
```
(defn twice [coll]
   (if (empty? coll)
      '()
      (cons (* 2 (first coll)) (twice (rest coll)))))
```

Função `remove-first`
Parâmetros:
`e` – elemento a ser removido
`coll` – sequência
Retorno:
sequência similar a `coll`, exceto pela supressão da primeira ocorrência de `e`
Casos:
```
(remove-first 'b '(a b c b d e))
;=> (a c b d e)
```

Discussão:
É preciso comparar o primeiro elemento `(first coll)` com o valor a ser removido `e`.
Se forem iguais, o resultado é o resto da lista
Se não forem iguais, faça de conta que `(remove-first e (rest coll))` é uma lista que atende o enunciado. Então, o resultado deve ser a lista que contém `(first coll)` na frente do resultado de `(remove-first e (rest coll))`

Caso base. A lista nula significa que o elemento não foi encontrado. O resultado deve ser a lista nula.

Resposta:
```
(defn remove-first [e coll]
   (if (empty? coll)
      '()
      (if (= e (first coll))
         (rest coll)
         (cons (first coll) (remove-first e (rest coll))))))
```

[1] Vários exemplos e exercícios nestas notas foram adaptados de MOREAU, L., QUEINNEC, C., RIBBENS, D., SERRANO, M. **Recueil de petits problèmes en Scheme.** Berlin: Springer, 1999.

3. Observações

Em Clojure, muitas vezes é mais conveniente usar `next` em vez de `rest`. Pesquise sobre a diferença.

Quando se diz que só se pode ver um elemento de cada vez, na verdade são alguns. Ou seja, além de `(first coll)`, outras funções como `(second coll)`, `(nth coll index)`, etc. podem ser usadas para puxar mais elementos da lista numa mesma execução da função.

Atente, nesse caso, para quantos elementos são suprimidos no chamado da recursão. Pesquise sobre `nnext`, `nthnext`.

Num Lisp qualquer, chamadas recursivas podem ser otimizadas pelo interpretador com base na ideia de *tail-call optimization* (TCO). Devido a limitações da JVM com respeito a TCO, Clojure não conta com a mesma facilidade. Para contornar essa limitação, Clojure oferece a função `recur`. Pesquise.

4. Mais um "pulo do gato"

Cada chamada recursiva é um sub-problema independente!!!

Por exemplo, para calcular o fatorial de 7 recursivamente, é preciso calcular o fatorial de 6, que, por sua vez, pede o fatorial de 5, e, assim, sucessivamente. Ou seja, calcular o fatorial de 7 levou a resolver outros 6 problemas.

O ponto é: o fatorial de 6, calculado para obter o fatorial de 7, não tem nada a ver com o fatorial de 7, é um problema separado. O fatorial de 6 poderia ter sido o alvo principal da chamada da função fatorial. Eu queria calcular o fatorial de 6. Ou ele pode ter sido chamado para calcular outro fatorial. Dá na mesma.

Dito de outra forma, quando o fatorial de 7 chama o fatorial de 6, é como se o fatorial de 7 já tivesse feito a sua parte e terminado. Não era esse o "faz de conta"? Que o fatorial de 6 já estava pronto? Quando o fatorial de 7 chama o fatorial de 6, ele vai esperar o resultado, mas do ponto de vista do fatorial de 6, ele não sabe disso. É um novo problema começando.

5. Exercícios

1) Função `all-odd` (*todos ímpares*)
Parâmetros:
`coll` – sequência
Retorno:
true – se todos elementos na sequência forem ímpares
Casos:
```
(all-odd '(1 7 3 17 11))
;=> true
(all-odd '(1 7 3 17 11 2))
;=> false
```

2) Função `and-map`
Parâmetros:
`pred` – predicado
`coll` – sequência
Retorno:
true – se todos elementos na sequência satisfizerem o predicado
Casos:
```
(and-map odd? '(1 7 3 17 11))
;=> true
(and-map char? '(\z \w \i 9))
;=> false
```
Dica:
Adapte `all-odd`

3) Função `my-some`
Parâmetros:
`pred` – predicado
`coll` – sequência
Retorno:
true – se um elemento na sequência satisfizer o predicado
Casos:
```
(my-some even? '(1 7 3 17 11))
;=> false
(my-some number? '(\z \w \i 9))
;=> true
```

4) Função `sum`
Parâmetros:
`coll` – uma sequência de números (lista ou vetor)
Retorno:
a soma dos elementos da sequência
Casos:
```
(sum '(1 7 3 17 11))
;=> 39
```

5) Função `my-count`
Parâmetros:
`e` – um elemento
`coll` – uma sequência
Retorno:
quantas vezes e ocorre na sequência
Casos:
```
(my-count 'b '(a b c b d b e))
;=> 3
```

6) Função `my-map`
Parâmetros:
`f` – uma função
`coll` – uma sequência
Retorno:
uma sequência similar, em que cada elemento `e` é substituido por `(f e)`
Casos:
```
(my-map inc '(1 2 3 4))
;=> (2 3 4 5)
(my-map rand-int '(10 100 1000))
;=> (3 26 103)
(my-map inc '())
;=> ()
```
Dica:
Adapte `twice`

7) Função `my-filter`
Parâmetros:
`pred` – um predicado
`coll` – uma sequência
Retorno:
uma sequência similar, mas que contém apenas os elementos `e` em que `(pred e)` é `true`
```
(my-filter odd? '(1 2 3 4))
;=> (1 3)
(my-filter odd? '())
;=> ()
```
Dica:
Se o predicado falha, não precisa fazer um `cons`

8) Função `stammer`
Parâmetros:
`coll` – uma sequência
Retorno:
uma sequência similar, mas com os elementos repetidos
Casos:
```
(stammer '(a b c d))
;=> (a a b b c c d d)
```
Dica:
`cons` deve ser usada duas vezes: `(cons ,,, (cons ,,, ))`

9) Função `list-ref`
Parâmetros:
`coll` – uma sequência
`pos` – um número significando uma posição na lista (0 = primeiro elemento)
Retorno:
o elemento na posição `pos`
Casos:
```
(list-ref '(a b c d e) 0)
;=> a
(list-ref '(a b c d e) 3)
;=> d
```
Dica:
Na recursão, a sequência e a posição devem "diminuir" juntas. Lembre-se: o único elemento visível é o primeiro.

10) Função `list-tail`
Parâmetros:
`coll` – uma sequência
`pos` – um número significando uma posição na lista (0 = primeiro elemento)
Retorno:
o resto da sequência a partir da posição `pos`
Casos:
```
(list-tail '(a b c d e) 0)
;=> (a b c d e)
(list-tail '(a b c d e) 3)
;=> (d e)
```
Dica:
Basta uma pequena mudança em `list-ref`

11) Função `alternate`
Parâmetros:
`coll` – uma sequência
Retorno:
uma sequência similar cujos elementos estão invertidos na ordem, de dois em dois
Casos:
```
(alternate '(a b c d e f))
;=> (b a d c f e)
(alternate '(a b c d e f g))
;=> (b a d c f e g)
```
Dica:
Acesse os dois primeiros elementos. Na recursão, diminua a sequência em duas posições. Inclua mais um caso de base além da lista nula: a lista com um só elemento.

12) Função `my-sorted?`
Parâmetros:
`coll` – uma sequência de números
Retorno:
`true` – se a sequência é ordenada
Casos:
```
(my-sorted? '(10 13 45))
;=> true
(my-sorted? '(12 15 3 20))
;=> false
(my-sorted? '())
;=> true
```
Dica:
Acesse os dois primeiros elementos. Na recursão, diminua a sequência em uma posição.

13) Função `odds`
Parâmetros:
`coll` – uma sequência
Retorno:
uma sequência similar cujos elementos nas posições pares são removidos
Casos:
```
(odds '(a b c d e f))
;=> (a c e)
(odds '(a b c d e f g))
;=> (a c e g)
```
Dica:
Adapte `alternate`.

14) Função `unique`
Parâmetros:
`coll` – uma sequência
Retorno:
uma sequência similar cujos elementos repetidos em sequência não mais se repetem
Casos:
```
(unique '(a a a b b a a a c c))
;=> (a b a c)
```
Dica:
Crie uma função auxiliar que possui mais um parâmetro. Esse parâmetro deve conter o elemento suprimido na recursão anterior. A função auxiliar é a real função que responde o problema. `unique` é só uma fachada para ela, como uma forma limpa de invocá-la (limpa quer dizer sem o parâmetro a mais poluindo a visão do usuário).

15) Função `my-merge`
Parâmetros:
`coll1` – uma sequência numérica ordenada
`coll2` – uma sequência numérica ordenada
Retorno:
uma sequência ordenada obtida a partir da junção de `coll1` e `coll2`
Casos:
```
(my-merge '(2 6 18 54) '(1 3 9 18 27 81))
;=> (1 2 3 6 9 18 18 27 54 81)
```
Dica:
Os dois elementos iniciais são visíveis. Qual deve ser inserido na sequência resultado? Quem deve diminuir na recursão?