6. Aquecimento

Os exercícios a seguir são para ajudar a rememorar a aula anterior:

1) Função `my-max`
Parâmetros:
`coll` – uma sequência de números (lista ou vetor)
Retorno:
o maior elemento da elementos da sequência
Caso:
```
(my-max '(1 7 3 17 11))
;=> 17
```

2) Função `append`
Parâmetros:
`coll1` – uma sequência
`coll2` – uma sequência
Retorno:
uma sequência obtida a partir da concatenação de `coll1` e `coll2` (= `concat` em Clojure)
Caso:
```
(append '(2 6 18 54) '(1 3 9 18 27 81))
;=> (2 6 18 54 1 3 9 18 27 81)
```
Dica:
A recursão é feita em apenas uma das sequências. Esse é um exempo interessante em que o caso base não resulta em uma lista nula.

7. A técnica do parâmetro-acumulador

Em muitos casos, programas recursivos admitem versões iterativas. Uma técnica muito empregada é usar parâmetros extras que servem para passar adiante os resultados parciais do programa.

Veja o exemplo do programa fatorial nas duas versões

programa fatorial, versão recursiva
```
(defn fat-rec [n]
    (if (zero? n)
      1
      (* n (fat-rec (dec n)))))
```

programa fatorial, versão iterativa
```
(defn fat-iter [n fat]
    (if (zero? n)
      fat
      (fat-iter (dec n) (* n fat)))
```

Observe a comparação entre as execuções das versões no cálculo do fatorial de 4:

```
(fat-rec 4) =
(* 4 (fat-rec 3)) =
(* 4 (* 3 (fat-rec 2))) =
(* 4 (* 3 (* 2 (fat-rec 1)))) =
(* 4 (* 3 (* 2 (* 1 (fat-rec 0))))) =
(* 4 (* 3 (* 2 (* 1 1)))) =
(* 4 (* 3 (* 2 1))) =
(* 4 (* 3 2)) =
(* 4 6) =
24
```

```
(fat-iter 4 1) =
(fat-iter 3 (* 4 1)) =
(fat-iter 2 (* 3 4)) =
(fat-iter 1 (* 2 12)) =
(fat-iter 0 (* 1 24)) =
24
```

A execução da versão iterativa parece ser mais ágil. De fato, com a TCO (*tail call optimization*), como não há nada pendente a ser feito na volta da recursão (no caso da versão iterativa), ganha-se em desempenho.

Versão iterativa para `length`

```
(defn length2-aux [coll len]
    (if (empty? coll)
        len
        (length2-aux (rest coll) (inc len))))
```

É comum criar uma função de fachada, que não possui o parâmetro extra. Isso torna o uso da função mais discreto.

```
(defn length2 [coll]
    (length2-aux coll 0))
```

Exemplo completo da execução:

```
(length2 '(10 20)) =
(length2-aux '(10 20) 0) =
(length2-aux '(20) 1) =
(length2-aux '() 2) =
2
```

Observe que a função de fachada não possui o parâmetro extra. A função iterativa é quem faz o trabalho de fato.

8. Mais exercícios
Crie versões iterativas para funções dos itens 5 e 6:
`sum`
`my-max`
`stammer` (dica: use `reverse` no final)

9. Ainda mais exercícios
1) Função `remq`
Parâmetros:
`el` – um elemento
`coll` – uma sequência
Retorno:
uma sequência similar, mas com todas as ocorrências do elemento removidas
Caso:

```
(remq 'b '(a b c b d e))
;=> (a c d e)
```

2) Função `remove-k-first`
Parâmetros:
`el` – um elemento
`k` – quantidade de repetições do elemento a serem removidas
`coll` – uma sequência
Retorno:
uma sequência similar, mas com k ocorrências do elemento removidas
Caso:

```
(remove-k-first 'a 2 '(1 2 a 3 4 a 5 6 a 7))
;=> (1 2 3 4 5 6 a 7)
```

3) Função `longest`
Parâmetros:
`coll` – uma sequência de sequências
Retorno:
a subsequência mais longa
Caso:

```
(longest '((-1 3) (a b c d) () (a b)))
;=> (a b c d)
```

4) Função `first-occurence`
Parâmetros:
`coll` – uma sequência
Retorno:
os elementos da sequência sem repetição, na ordem em que aparecem na sequência
Caso:
```
(first-occurence '(a a a b b a a a c c))
;=> (a b c)
```

5) Função `substitute`
Parâmetros:
`old` – um elemento a ser substituído
`new` – o elemento que substitui `old`
`coll` – uma sequência
Retorno:
uma sequência com todas ocorrências de `old` substituídas por `new`
Caso:
```
(substitute 'c '(a b) '(a c 3 c b))
;=> (a (a b) 3 (a b) b)
```

6) Função `follow`
Parâmetros:
`el` – elemento
`coll` – uma sequência
Retorno:
uma sequência dos elementos que ocorrem depois de `el` (se for o último, considera-se que a lista vazia o segue)
Casos:
```
(follow ‘b ‘(a b c b d e))
;=> (c d)
(follow ‘b ‘(a b c b d b))
;=> (c d ())
(follow ‘b ‘(a b b d b e))
;=> (b d e)
```

7) Função `pairlis`
Parâmetros:
`coll1` – uma sequência
`coll2` – uma sequência
Retorno:
uma sequência de vetores; cada vetor contém os elementos de mesma posição nas duas listas
Caso:
```
(pairlis '(a b c) '(1 2 3))
;=> ([a 1] [b 2] [c 3])
```

8) Função `pos+`
Parâmetros:
`coll` – uma sequência de números
Retorno:
uma sequência que contém os elementos de `coll` somados com as respectivas posições
Caso:
```
(pos+ '(7 5 1 4))
;=> (7 6 3 7)
```

9) Função `is-suffix?`
Parâmetros:
`coll1` – uma sequência
`coll2` – uma sequência
Retorno:
`true` – se `coll1` é sufixo de `coll2` sufixo, isso significa dizer que `coll1` pode ser obtida a partir da aplicação repetida de `rest` sobre `coll2`
Casos:

```
(is-suffix? '(1 2 3) '(4 3 2 1 2 3))
;=> true
(is-suffix? '(1 2 3) '(4 3 2 1 2))
;=> false
```

10) Função `common-suffix?`
Parâmetros:
`coll1` – uma sequência
`coll2` – uma sequência
Retorno:
a maior sub-lista que é um sufixo comum entre `coll1` e `coll2`
Casos:

```
(common-suffix '(73 45 3 4 56 5 6 8) '(3 4 5 6 8))
;=> (5 6 8)
(common-suffix '(73 45 3 4 56 5 6 8) '(3 4 5))
;=> ()
```