02

Funções recursivas e recursão de cauda

Capítulo 5 - Funções recursivas e recursão de cauda

O básico do nosso jogo de forca está pronto e funcionando. Porém ainda falta dar alguns retoques e melhorá-lo. Vejamos as funções `avalia-chute` e `jogo` :

```
(defn jogo [vidas palavra acertos]
    (if (=vidas 0)
        (perdeu)
        (if (acertou-a-palavra-toda? palavra acertos)
            (ganhou)
            (avalia-chute (le-letra!) vidas palavra acertos)
        )
    )
)

(defn avalia-chute [chute vidas palavra acertos]
    (if (acertou? chute palavra)
        (jogo vidas palavra (conj acertos chute))
        (jogo (dec vidas) palavra acertos)
    )
)
```

Perceba que uma invoca a outra e isto pode ser um problema na linguagem funcional, pois elas vão se empilhando de forma recursiva. Uma forma de solucionar esse problema é inserir o código de uma função dentro do outro. Para tal precisaremos fazer algumas mudanças:

```
(defn jogo [vidas palavra acertos]
    (cond
        (=vidas 0) (perdeu)
        (acertou-a-palavra-toda? palavra acertos) (ganhou)
        :else
        (let [chute (le-letra!)]
            (if (acertou? chute palavra)
                (jogo vidas palavra (conj acertos chute))
                (jogo (dec vidas) palavra acertos)))))
```

O `cond` funciona como *if* e *else if*. O `let` serve para abrirmos um novo bloco de código em que invocamos outras variáveis. Dessa maneira podemos inclusive excluir a função `avalia-chute` .

O próximo passo é implementarmos um sistema de *feedback*. Usaremos a função `do` :

```
(defn jogo [vidas palavra acertos]
    (cond
        (=vidas 0) (perdeu)
        (acertou-a-palavra-toda? palavra acertos) (ganhou)
        :else
        (let [chute (le-letra!)]
```

```
            (if (acertou? chute palavra)
                (do
                    (println "Acertou a letra!")
                    (jogo vidas palavra (conj acertos chute)))
                (do
                    (println "Errou a letra! Perdeu vida!")
                    (jogo (dec vidas) palavra acertos))))))
```

Vendo no Terminal:

```
forca.core=> (forca/jogo 2 "MELANCIA" #{})
Q
Errou a letra! Perdeu vida!
W
Errou a letra! Perdeu vida!
Você perdeunil

forca.core=> (forca/jogo 2 "MELANCIA" #{})
M
Acertou a letra!
E
Acertou a letra!
L
Acertou a letra!
A
Acertou a letra!
N
Acertou a letra!
C
Acertou a letra!
I
Acertou a letra!
Você ganhou!nil
```

Usamos o `println` para haver quebra de linha.

Porém ainda temos que resolver o problema do começo da aula. Dessa vez a função `jogo` invoca ela mesma. Em programação funcional existe um conceito que chamamos de ***recursão de cauda*** e será ela que nos ajudará a melhor implementar a função. Tal conceito diz que se a última linha de código da função chama ela mesma, então o compilador é inteligente o suficiente para não fazer os vários empilhamentos. Usamos `recur` no lugar do nome da função:

```
(defn jogo [vidas palavra acertos]
    (cond
        (=vidas 0) (perdeu)
        (acertou-a-palavra-toda? palavra acertos) (ganhou)
        :else
        (let [chute (le-letra!)]
            (if (acertou? chute palavra)
                (do
                    (println "Acertou a letra!")
                    (recur vidas palavra (conj acertos chute)))
                (do
                    (println "Errou a letra! Perdeu vida!")
                    (recur (dec vidas) palavra acertos))))))
```