02

Listas e suas Funções

Capítulo 3 - Listas e suas Funções

Já está claro que a função `jogo` é a principal do nosso programa e precisará receber mais parâmetros:

- *palavra*: a palavra secreta a qual o jogador deverá descobrir,
- *acertos*: as letras das palavras que o jogador já acertou.

Vamos passar para a implementação dessa função:

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

O símbolo de interrogação é muito comum ao se programar em *Clojure* quando se trata de condições que acarretam em "verdadeiro" ou "falso". Nesse caso o retorno é uma função que avisa que o jogador ganhou. Então, logo abaixo de `perdeu`, fazemos:

```
(defn ganhou [] (print "Você ganhou!"))
```

O que imprimirá a mensagem de vitória. Há ainda o cenário no qual não se ganha nem se perde, ou seja, existe vidas e deve-se chutar uma outra letra:

```
(defn jogo [vidas palavra acertos]
    (if (= vidas 0)
        (perdeu)
        (if (acertou-a-palavra-toda? palavra acertos)
            (ganhou)
            (print "Chuta, amigo!")
        )
    )
)
```

Por enquanto o código está um pouco confuso, mas vamos entender o que se passa:

- Se vidas = 0, o jogador perdeu,
- Caso contrário, se o jogador acertou a palavra toda, ele ganhou,
- Caso contrário, o jogador deve chutar uma letra.

Falta implementar a função `acertou-a-palavra-toda`, a qual receberá dois parâmetros:

```
(defn acertou-a-palavra-toda? [palavra acertos] true)
```

Por motivos de teste, ela retornará sempre verdadeiro (`true`). Agora a função `jogo` recebe mais de um parâmetro: vidas, a palavra e as letras certas. Este último será do tipo *conjunto*, que é declarado com `#{}` :

```
forca.core=> (forca/jogo 5 "MELANCIA" #{"A" "M"})
Você ganhou!nil
```

Obviamente este foi o resultado, pois ainda não implementamos toda a função `acertou-a-palavra-toda` . Testando com `false` no lugar de `true` :

```
forca.core=> (forca/jogo 5 "MELANCIA" #{"A" "M"})
Chuta, amigo!nil
```

E se passarmos vidas = 0:

```
forca.core=> (forca/jogo 0 "MELANCIA" #{"A" "M"})
Você perdeunil
```

Funções que trabalham com conjuntos

Vamos tentar entender melhor a ideia de conjunto e as funções que a cercam. Definemos o seguinte conjunto:

```
forca.core=> (def conjunto #{"L" "M" "A"})
#'user/conjunto

forca.core=> conjunto
#{"L" "M" "A"})
```

contains?

Existe uma função que "pergunta" para o conjunto se um elemento específico está lá dentro. A ela damos o nome de `contains?` e fazemos:

```
forca.core=> (contains? conjunto "L")
true

forca.core=> (contains? conjunto "Q")
false
```

Ou seja, a letra L está contida no conjunto e a letra Q não.

conj e *disj*

A função `conj` adiciona elemento em conjuntos:

```
forca.core=> (conj conjunto "X")
#{"L" "M" "X" "A"}
```

```
forca.core=> conjunto
#{"L" "M" "A"}
```

Perceba que a variável `conjunto` continua a mesma.

A função `disj` subtrai elemento em conjuntos:

```
forca.core=> (disj conjunto "L")
#{"M" "A"}
```

map

Essa função recebe uma outra função que é aplicada em uma lista de números. É o `map` que irá tratar cada um dos elementos. Vejamos um exemplo:

- Definimos um *array* de números:

  ```
  (def nums [1 2 3 4])
  ```

- Definimos uma função que multiplica um número por 2:

  ```
  (defn mult [x] (* x 2))
  ```

- Chamamos a função `map` que recebe a anterior aplicada na lista de números:

  ```
  (map mult nums)
  (2 4 6 8)
  ```

  A função `map` pegou cada um dos elementos da lista e passou para a função `mult` .

Porém, vamos precisar de uma função que remove elementos com características em comum. Vamos ver um exemplo utilizando números pares:

```
defn par [x] (= 0 (rem x 2)))
```

A função `par` é definida pelos elementos cujo resto da divisão por dois seja igual a zero. Usando o mesmo *array* anterior podemos fazer:

```
(remove par nums)
(1 3)
```

De fato, foram removidos os números pares. Perceba que é bem comum chamar funções como parâmetros na linguagem funcional.

Agora já temos conhecimento o suficiente para implementarmos `acertou-a-palavra-toda`

A função *letras-faltantes*

Como saberemos se o jogador acertou a palavra toda? Basta fazermos uma comparação entre a palavra e os chutes. Se todas as letras da palavra estiverem contidas no conjunto de letras chutadas corretamente (**acertos**), então o jogador ganhou.

Se estivéssemos programando em **C**, faríamos um *loop* passando por cada letra. Mas em programação funcional processamos conjuntos de maneira diferente, usando a função **map**

Porém, vamos aplicar nosso conhecimento adquirido sobre como remover elementos de listas numa outra função: **letras-faltantes** :

```
(defn letras-faltantes [palavra acertos]
    (remove (fn [letra] (contains? acertos (str letra))) palavra))
```

A função remove as letras chutadas que estão na palavra. Então, por exemplo,

```
forca.core=> (forca/letras-faltantes "MELANCIA" #{"M" "E"})
(\L \A \N \C \I \A)

forca.core=> (forca/letras-faltantes "MELANCIA" #{"M" "E" "A" "L"})
(\N \C \I)
```

De fato, retornaram as letras faltantes. Agora ficou fácil implementar a função **acertou-a-palavra-toda** :

```
(defn acertou-a-palavra-toda? [palavra acertos]
    (empty? (letras-faltantes palavra acertos))
)
```

Se o conjunto de letras faltantes for vazio, então o jogador acertou todas as letras da palavra. Vamos testar no Terminal:

```
forca.core=> (forca/jogo 5 "MELANCIA" #{"M" "E"})
Chuta, amigo!nil

forca.core=> (forca/jogo 5 "MELANCIA" #{"M" "E" "L" "A" "N" "C" "I"})
Você ganhou!nil
```

De fato, quando não falta nenhuma letra, o programa retorna a frase de vitória. Mas tudo isso começa na função **jogo** que chama a **acertou-a-palavra-toda** que, por sua vez, chama **letras-faltantes** . Dentro desta acontece o processamento da lista, que vê se cada letra da palavra está dentro do conjunto acertos, as que estiverem são removidas.

O mais importante nesta aula é entender que com a linguagem funcional, usando o *Clojure*, conseguimos chamar funções dentro de funções. Vamos ver um outro exemplo para fixar tal ideia. Criamos uma função:

```
    (defn triplica [x] (* x 3))

    (triplica 5)
15
```

Agora a usamos dentro do `map` para que ela seja aplicada dentro do *array* `nums` criado anteriormente:

```
    (map triplica nums)
(3 6 9 12)
```

De fato, o `map` nos ajuda a processar listas utilizando outras funções.