DO REPL ÀS FUNÇÕES

Série “Programação Funcional Desde O Básico”

Linguagens de Programação Não Convencionais

Unesp >< IGCE

Do REPL às Funções

- Ponto de partida
  - Uma sessão de REPL de exploração de ideias
- Alvo
  - Código da função ou funções
- Método
  - Considere fazer mais de uma função
    - Sessões longas costumam ter padrões que se repetem
  - Expresse o fluxo de dados em um formato conveniente
    - *One-liner*
    - Encadeamento por variáveis locais (*form* `let`)
    - Encadeamento por *threading macros* (`->` ou `->>`)

Exemplo

- Enunciado
  - Criar a função `numbered-seq`
    - Entrada: uma sequência qualquer
    - Saída: a sequência numerada em pares
  - Ilustração

```
(numbered-seq [:a :b :c])
;=> [[1 :a] [2 :b] [3 :c]]
```

- Projeto (fluxo de dados)
  - Interlaçar a sequência com outra sequência de *range*
  - Particionar o resultado em pares

Exemplo

- Experimentação (sessão de REPL)

```
[:a :b :c]
;=> [:a :b :c]
(range 1 (inc (count *1)))
;=> (1 2 3)
(interleave *1 *2)
;=> (1 :a 2 :b 3 :c)
(partition 2 *1)
;=> ((1 :a) (2 :b) (3 :c))
```

Lembre-se:

- `*1` é o **último** resultado obtido no REPL **em relação à linha atual**
- `*2` é o penúltimo
- `*3` é o antepenúltimo

One-liner

$$f(g(x))$$

- É o encadeamento naturalmente encontrado em programação e matemática
- Todo encadeamento pode ser escrito em uma linha
- Idealmente, um programa funcional é composto de muitas pequenas funções *one-liner*
- Na prática, dificulta a leitura da função
- Só é recomendável para encadeamentos muito pequenos

Exemplo

- Sessão de REPL

```
[:a :b :c]
;=> [:a :b :c]
(range 1 (inc (count *1)))
;=> (1 2 3)
(interleave *1 *2)
;=> (1 :a 2 :b 3 :c)
(partition 2 *1)
;=> ((1 :a) (2 :b) (3 :c))
```

Cores mostram a relação entre os resultados do REPL e os passos subsequentes, e também a função final

Na função final, obviamente `*1`, `*2` e `*3` não aparecem

- Função no formato *one-liner*

```
(defn numbered-seq [xs]
  (partition 2 (interleave (range 1 (inc (count xs))) xs)))
```

OFF: Tirando Proveito de *Laziness*

- Mudando de assunto ligeiramente…
- A função de exemplo pode ser simplificada tirando proveito de *laziness*
- *Laziness* não combina bem com o REPL por causa do Print
  - O P em REPL
- Função original

```
(defn numbered-seq [xs]
  (partition 2 (interleave (range 1 (inc (count xs))) xs)))
```

- Função otimizada

```
(defn numbered-seq [xs]
  (partition 2 (interleave (rest (range)) xs)))
```

Form `let`

$$a = g(x), f(a)$$

- Encadeamento por variáveis locais
- Cada nova variável local é definida a partir das anteriores
- Os nomes das variáveis ajudam a visualizar as etapas
  - Por outro lado, nomes sobrecarregam a apresentação da função
- Rotinas executadas uma única vez
  - `(aaa (bb 8 12) (c (bb 8 12)))  ; 2 execuções`
  - `(let [b (bb 8 12)] (aaa b (c b))) ; 1 execução`

Exemplo

- Sessão de REPL

```
[:a :b :c]
;=> [:a :b :c]
(range 1 (inc (count *1)))
;=> (1 2 3)
(interleave *1 *2)
;=> (1 :a 2 :b 3 :c)
(partition 2 *1)
;=> ((1 :a) (2 :b) (3 :c))
```

- Função na *form* `let`

```
(defn numbered-seq [xs]
  (let [numbers (range 1 (inc (count xs)))
        mixed (interleave numbers xs)]
    (partition 2 mixed)))
```

Anatomia da *form* `let`

```
(defn numbered-seq [xs]
  (let [numbers (range 1 (count xs))
        mixed (interleave numbers xs)]
    (partition 2 mixed)))
```

1. Comece identificando quem desempenhou o papel de "entrada" do processo na sessão de REPL
   - Esses são os parâmetros
2. O corpo da função é **uma única** *form* let, não importa quão grande ela seja: comece digitando `(let [`
3. Diversas variáveis locais podem ser criadas dentro do bloco delimitado pelos colchetes
4. Depois dos colchetes, **uma única** função (*one-liner*, *threading macro* ou outra *form* `let`) cria o resultado do `let`, e portanto da função; nela as variáveis locais são usadas

Threading macros

- Macros são transformações de código fonte

```
(macroexpand-1 '(->> (a 1) (b 2) c (d 3 4)))
;=> (d 3 4 (c (b 2 (a 1))))
(macroexpand-1 '(-> (a 1) (b 2) c (d 3 4)))
;=> (d (c (b (a 1) 2)) 3 4)
```

- O resultado de um elemento é inserido no elemento seguinte
  - Macro -> insere na segunda posição
  - Macro ->> insere na última posição
- A apresentação da função fica muito mais leve
- Funções que não seguem o padrão atrapalham
- Só há passagem de dados entre etapas subsequentes

Exemplo

- Sessão de REPL

```
[:a :b :c]
;=> [:a :b :c]
(range 1 (inc (count *1)))
;=> (1 2 3)
(interleave *1 *2)
;=> (1 :a 2 :b 3 :c)
(partition 2 *1)
;=> ((1 :a) (2 :b) (3 :c))
```

- Função com *threading macros*

```
(defn numbered-seq [xs]
  (->> xs
       (interleave (rest (range)))
       (partition 2)))
```

Resumo

	One-liner	`let`	`-> / ->>`
Tamanho da função	Pequeno, na prática	Qualquer	Qualquer
Minimiza repetições	Não	Sim	Não
Fluxo entre etapas “afastadas”	Com repetições	Sim	Não
Posição do parâmetro	Qualquer	Qualquer	Sempre a mesma
Legibilidade	Péssima para funções médias e grandes	Boa	Excelente

Guia de Decisão

- O sessão de REPL é muito pequena?
  - Sim: *one-liner*
- Na sessão de REPL, há subexpressões que acontecem muitas vezes?
  - Sim: `let`
- Na sessão de REPL, todas as funções recebem parâmetros do passo anterior e na mesma posição?
  - Sim: `->` / `->>`
  - Não: `let`
- Na prática, é comum misturar os formatos e contornar pequenos problemas de posição etc. com “jeitinhos” para usar `->` / `->>`