Bits & Bytes: Programação Funcional

Fabio Galuppo

Programação Funcional...sem enrolação

Se você abrir qualquer livro de Matemática Discreta ou até mesmo um de Matemática dos primeiros anos do colegial, lá irá encontrar coisas sobre conjuntos, domínios, contradomínios, imagens, funções, relações, entre outros assuntos. Pois bem, a familiaridade com estes elementos já é um bom ponto de partida para entender e começar aplicar **programação funcional** de um forma consciente.

Aqui vamos ilustrar alguns exemplos de conjuntos e funções:

$A = \{1,2,3,4,6\}$	Conjunto formado pelos números inteiros positivos entre 1 e 6 denominado de A
$\|A\| = 6$	Cardinalidade do conjunto A é 6
$B = \{1,2,3\}$ $C = \{4,5\}$ $B \times C - \{(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)\}$	Produto cartesiano dos conjuntos B e C resulta em um conjunto de pares ordenados
$D = \{x \in A : x \text{ é um número par}\} = \{2,4,6\}$	Seleciona os elementos pares do conjunto A
$f : B \to D$ e $g : D \to B$ $f(x) = 2x$ e $g(x) = x/2$ $f(1) = 2, f(2) = 4$ e $f(3) = 6$	Funções bijetora f e g, onde g é a inversa de f ou vice-versa

Caso tenha alguma familiaridade com Python, deve saber que há um ambiente interativo por linha de comando denominado de Read-Eval-Print-Loop (REPL). A maioria das linguagens funcionais possuem um REPL para experimentação. Usando listas, uma das estruturas elementar na programação funcional, e funções, apresentamos os conjuntos e as operações da tabela anterior na versão computacional em F#:

```
fsi
> let A = [1;2;3;4;5;6];;

val A : int list = [1; 2; 3; 4; 5; 6]

> ;;
> List.length A;;
val it : int = 6
> ;;
> let B = [1;2;3];;

val B : int list = [1; 2; 3]

> let C = [4;5];;

val C : int list = [4; 5]

> List.collect (fun b -> List.zip [for i in 1 .. (List.length C) -> b] C) B;;
val it : (int * int) list = [(1, 4); (1, 5); (2, 4); (2, 5); (3, 4); (3, 5)]
> ;;
> let D = List.filter (fun a -> a % 2 = 0) A;;

val D : int list = [2; 4; 6]

> let f x = 2 * x;;

val f : x:int -> int

> List.map f B;;
val it : int list = [2; 4; 6]
> ;;
> List.map f B = D;;
val it : bool = true
```

A programação funcional, é um estilo que enfatiza o uso de funções e a imutabilidade das "variáveis" do programa. A programação decorre de um modo declarativo, onde se determina o que (*what*) deve ser feito através de expressões ou declarações de alto nível. Em contraste, a programação imperativa, principal estilo das linguagens de programação tradicionais, tais como Java, C# ou C++, onde são determinados os detalhes de como (*how*) a computação deve ocorrer, bem como, as mudanças do estado do programa (máquina de estado). A um certo tempo este tipo de programação declarativa vem influenciando linguagens tradicionais como C# com a introdução do Language Integrated Query (LINQ), ou até mesmo o Java 8 que suporta um estilo funcional para expressar código com Lambda Expressions.

Bits & Bytes: Programação Funcional

Fabio Galuppo

A seguir um exemplo com LINQ demonstrando a seleção dos elementos pares da lista A:

```
var A = new List<int>() { 1, 2, 3, 4, 5, 6 };
var D = (from a in A where a % 2 == 0 select a).ToList();
```

D Count = 3
[0] 2
[1] 4
[2] 6

Uma das características da programação funcional é o uso de funções de alta ordem (*higher-order functions*). Elas são funções que recebem uma ou mais funções como argumentos ou retornam uma função com resultado, em alguns casos ambos ocorrerão. As funções para filtrar ou para mapear elementos de uma lista (ou de um conjunto) são funções de alta ordem, ambas recebem uma função como argumento. Note isso nos trechos a seguir escritos em F# e C# com LINQ através do seu modelo de objetos.

```
> A;;
val it : int list = [1; 2; 3; 4; 5; 6]
> B;;
val it : int list = [1; 2; 3]
> let D = List.filter (fun a -> a % 2 = 0) A;;

val D : int list = [2; 4; 6]

> List.map (fun b -> b * 2) B;;
val it : int list = [2; 4; 6]
```

```
var A = new List<int>() { 1, 2, 3, 4, 5, 6 };
var D = A.Where(a => a % 2 == 0).ToList();

var B = new List<int>() { 1, 2, 3 };
var E = B.Select(b => b * 2).ToList();
```

E Count = 3
[0] 2
[1] 4
[2] 6

As funções atribuídas a map (Select) ou filter (Where) nos exemplos são denominadas funções anônimas por não serem nomeadas e estarem definidas localmente. Imagine, isso como um bloco de código que é passado para (ou retornado de) uma função como um valor, simples assim! Inclusive, na linguagem de programação Objective-C da Apple isso é denominado *blocks*. Um outro nome muito comum, com um certo *hype*, para esta funcionalidade é *lambda expression* ou somente ***lambda***.

Recursividade, ao invés de iteração, é como as linguagens funcionais expressam *loops*. Como exemplo a seguir, a computação do fatorial na linguagem Erlang.

```
1  -module(test).
2  -export([factorial/1]).
3
4  % non-tail-recursive implementation
5  %factorial (0) -> 1;
6  %factorial (1) -> 1;
7  %factorial (2) -> 2;
8  %factorial (N) -> N * factorial(N-1).
9
10 % tail-recursive implementation
11 factorial(0, _)   -> 1;
12 factorial(1, Acc) -> Acc;
13 factorial(N, Acc) -> factorial(N-1, Acc * N).
14
15 factorial (N) -> factorial(N, 1).
16
17
18
```

Erlang
File Edit Options View Help

```
17>
17> c(test).
{ok,test}
18> test:factorial(5).
120
19> test:factorial(0).
1
20> test:factorial(20).
2432902008176640000
```

As linguagens funcionais fazem uso de um recurso especial conhecido como *pattern matching*. Ele permite a decomposição de valores (como 0, 1, e N do exemplo anterior), listas e outras estruturas da linguagem para verificação ou reconhecimento de um padrão.

O próximo exemplo é uma implementação em Erlang da função de alta ordem *filter* utilizando decomposição de listas, onde X em [X | Xs] representa o primeiro item da lista e Xs o restante, assim como [] representa uma lista vazia:

Bits & Bytes: Programação Funcional

Fabio Galuppo

```
-module(test).
-export([filter/2]).

filter(_, [])       -> [];
filter(Predicate, [X | Xs]) ->
    Answer = Predicate(X),
    case Answer of
        true -> [X | filter (Predicate, Xs)];
        _    -> filter (Predicate, Xs)
    end.
```

```
Erlang
File  Edit  Options  View  Help

58> A.
[1,2,3,4,5,6]
59> test:filter(fun(X) -> X rem 2 == 0 end, A).
[2,4,6]
```

O que foi apresentado até aqui são alguns pequenos exemplos dos elementos que são encontrados nas linguagens de programação funcional. Aprender qualquer uma delas entre Haskell, Scala, F#, Erlang ou Clojure vai enriquecer suas habilidades de programação e deixar seus programas mais claros e expressivos, mesmo que (infelizmente) você não venha trabalhar com elas diretamente. No entanto, se quiser aplicar tais técnicas com Java, JavaScript, C# e/ou C++ uma vez que estas linguagens incorporam cada vez mais estas características funcionais aqui vão algumas boas referências para iniciar:

- Real-World Functional Programming in F# and C#, que está disponível parcialmente no MSDN.
- Functional Programming in Java // Functional Programming in C++ // Functional Programming in JavaScript

A programação funcional também é essencial para o bom entendimento e aplicação das técnicas da programação reativa, como é demonstrado aqui na conexão entre *lambdas* e *streams*:

```
Subscription sub0 = observable
        .filter((i) -> {
            return isEven(i);
        })
        .subscribe(onNext, onError, onCompleted);

Subscription sub1 = observable
        .filter((i) -> {
            return isOdd(i);
        })
        .buffer(10)
        .map(xs -> xs.get(0))
        .subscribe(onNext, onError, onCompleted);

sub1.unsubscribe();
sub0.unsubscribe();
```

A programação funcional não é nenhuma bala de prata, fazer uso de muitas funções anônimas encadeadas poderá levar a problemas de desempenho e/ou tornar seu código incompreensível. Porém, se usada da forma correta ela trará excelentes vantagens tais como diminuir a quantidade de linhas de código e trazer expressividade aos seus programas. O artigo da InfoWorld traz 14 excelentes razões para F# ser adotado. Também é notório o uso fortemente dela na indústria em Inside Erlang, The Rare Programming Language Behind WhatsApp's Success.

Exemplos não faltam, inclusive na área financeira: "*At Credit Suisse, we've been using F# to develop quantitative models for financial products*.". Experimente F# no seu *browser*: "*F# is used by some of the world's largest financial institutions as a code-oriented financial modeling and engineering language.*"

Se desejar consultar mais exemplos e outros recursos da programação funcional, minha apresentação sobre este assunto num MeetUp em 2015 está disponível no GitHub. Boa diversão!