Programação Funcional

Lourenço Bogo

26 de dezembro de 2020

Sumário

1 Introdução

Programação funcional foi o primeiro tópico que vimos na matéria.

Trata-se de um paradigma onde as funções são valores, ou seja, podemos passá-las como argumentos e atribuí-las a variáveis. Além disso, o paradigma possui outras propriedades relevantes que irei abordar em seguida.

2 Propriedades Principais

Uma parte extremamente importante de linguagens funcionais é que, em geral, os procedimentos (funções) não têm efeitos colaterais: uma função apenas recebe os seus argumentos, executa algum tipo de operação com eles e no final retorna um novo valor. O procedimento não altera variáveis de outro escopo, inclusive os seus argumentos, que permanecerão inalterados no ambiente do escopo no qual o procedimento foi chamado.

As funções podem ser argumentos de outras funções, o que nos conduz a um ponto crucial do paradigma funcional: as funções de ordem superior. Essas são funções cujos parâmetros são outras funções e, quando combinadas com certas propriedades do paradigma, nos permitem atingir um nível de abstração muito interessante nas linguagens. Com base nesse conceito podemos criar alguns dos procedimentos mais importantes e conhecidos da programação funcional, como o `map`, o `filter` e o `reduce`.

3 Lambdas e Closures

Sendo as nossas funções valores, não é necessário que tenham nomes e, por essa razão, são definidas de maneira anônima. Denominamos funções anônimas como lambdas. Se quisermos dar-lhes um nome, podemos simplesmente atribuí-las a alguma variável, já que elas são valores comuns iguais a `1` ou `"foo"`.

Outro conceito é o fechamento (closure), que é um par `(lambda, ambiente)`. Quando criamos uma nova função usando `lambda`, montamos um fechamendo com a lambda previamente definida e o ambiente atual é estendido pela variável dessa lambda. Isso é feito por dois motivos principais:

1. No momento de fazer a aplicação da função definida, vamos ao seu ambiente e alteramos o valor da variável para o valor passado como argumento na aplicação.

2. Como todo ambiente de fechamento é uma extensão do ambiente no qual esse fechamento foi criado, as funções mais internas poderão acessar os argumentos das funções externas.

Vale relembrar aqui a diferença entre escopo léxico e escopo dinâmico. O que usamos em nossos trabalhos foi o escopo léxico, onde uma variável terá o valor que lhe foi dado localmente, pois expandimos os ambientes da maneira que foi explicada acima. Já no escopo dinâmico, todo identificador terá associada uma pilha global de valores. Quando criamos uma variável cujo nome é esse identificador, inserimos o valor atribuído à variável na pilha do identificador. Ao avaliarmos certa variável, o valor resultante será o que estiver no topo da pilha. O desempilhamento é feito seguindo os escopos.

4 Currying

Outro ponto interessante abordado foi o conceito de `curry` que, de forma sintética e simplista, significa transformar funções de múltiplos parâmetros em cadeias de funções com apenas um parâmetro. Exemplificando:

```
;; Função definida sem currying
(define somaDois
  (lambda (x y)
    (+ x y)))

;; Função definida com currying
(define somaDoisCurry
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

A primeira função só pode ser chamada quando passamos dois argumentos e ela somará os dois. A segunda função recebe um argumento, devolve um fechamento, que também recebe um argumento, e só então soma os dois. Isso é útil, pois torna o código mais reutilizável e sustentável. Por exemplo, se quisermos fazer um procedimento que recebe um argumento e acrescenta 1 a esse argumento, poderíamos reutilizar `somaDoisCurry`, e definir a nova função mais facilmente:

```
(define somaUm
  (somaDoisCurry 1)) ;; Retorna uma lambda de um argumento que soma 1 a esse argumento
```

5 Continuações

Aprendemos também sobre continuações. Ao invés de termos uma função que recebe e retorna valores, adcionamos a essa função outro parâmetro que será um procedimento. O argumento desse procedimento será o valor que a função estava retornando anteriormente. Ou seja:

- Fazíamos $f(x) \to y$
- Passamos a fazer $f(x, g) \to g(x)$

A grande utilidade disso é que nos permite controlar melhor o fluxo do programa e tratar mais facilmente situações não usuais, como por exemplo, sair de recursões sem termos que voltar por todas as chamadas.