Andrei de A. Formiga

Programação Funcional

DECEMBER 2018

Universidade Federal da Paraíba

Introdução

Lorem ipsum.

Programação Recursiva

Um dos primeiros problemas de quem está começando na programação funcional, mas tem experiência com programação imperativa, é saber como é possível escrever programas sem poder alterar o valor de variáveis. A solução para isso passa pela programação recursiva, que aqui usamos no sentido de resolver problemas de programação usando funções recursivas.

Através da Teoria da Computação sabemos que vários formalismos para descrever algoritmos são equivalentes (a chamada hipótese de Church-Turing), e entre eles temos a Teoria das Funções Recursivas e as Máquinas de Turing. As Máquinas de Turing podem representar facilmente algoritmos imperativos, onde valores da memória (que é a fita da máquina) podem ser alterados livremente.

Já na Teoria das Funções Recursivas todos os algoritmos devem ser expressos como combinações de funções recursivas, e isso significa (pela equivalência com as Máquinas de Turing) que qualquer algoritmo que pode ser escrito de maneira imperativa pode ser escrito como uma combinação de funções recursivas.

O problema é que normalmente não praticamos muito a programação recursiva, que é pouco explorada no ensino tradicional de programação. Alguns professores chegam a dizer que não se deve usar recursividade, pois pode tornar o programa menos eficiente (uma afirmação que iremos investigar mais tarde).

Entretanto, saber como programar processos recursivos é essencial, não só na programação funcional mas também em outros paradigmas e ferramentas da programação. Por isso, neste capítulo vamos sanar essa deficiência estudando (e praticando) como escrever funções usando recursividade.

A estrutura das funções recursivas primitivas

Existem várias maneiras de escrever funções recursivas, aqui vamos começar estudando uma das formas mais simples e diretas para tarefas de programação.

Essa forma simples é possível quando a função deve processar um

valor de entrada que tem uma estrutura recursiva. A máxima que seguimos aqui é:

A estrutura do código (da função recursiva) deve seguir a estrutura dos dados que ela recebe.

Inicialmente vamos considerar dois tipos de dados que têm estrutura recursiva: números naturais e listas.

Números naturais

Apesar dos números naturais (aqueles que aparecem "naturalmente" quando contamos) e sua aritmética terem sido a base da matemática desde sempre, até o final do Século XIX não havia uma formalização satisfatória desses conceitos.

O matemático italiano Giuseppe Peano (1858–1932) foi o responsável por colocar os números naturais e sua aritmética em bases sólidas, elaborando um conjunto de axiomas que são usados até hoje.

Desses axiomas podemos derivar uma definição recursiva dos números naturais.

Definição: (Número natural) Um número natural satisfaz uma das duas condições:

1. é zero; ou
2. é o sucessor de outro número natural n, em símbolos $S(n)$

Essa definição pode ser usada como uma receita para construir todos os números naturais: comece com zero, depois aplique a operação $S(\cdot)$ para obter o sucessor de zero (o número um), depois o sucessor deste, e por aí vai.

Quando escrevemos uma função que deve processar um número natural, usamos a definição no sentido inverso, como uma forma de "desconstruir" um número: um número natural ou é zero, ou é o sucessor de algum número menor. Se escrevemos uma função recursiva em um parâmetro que é um número natural, o caso do parâmetro igual a zero normalmente é a base da recursão, na qual a função pode dar o valor diretamente; caso contrário, o parâmetro é o sucessor de um número menor, e o valor da função depende da chamada recursiva passando o número menor como parâmetro. A chamada recursiva muitas vezes precisa ser combinada de alguma forma com o valor atual do parâmetro para obter a resposta final.

Vejamos um exemplo: digamos que precisamos escrever uma função para somar dois números naturais, n e m, usando apenas operações de obter o sucessor (incremento) e o predecessor (decremento) de um número. Em uma linguagem imperativa como C podemos

usar operadores de incremento e decremento para alterar diretamente o valor de uma variável:

```
int soma(int m, int n) {
  for (int i = n; i > 0; i--) {
    m++;
  }
  return m;
}
```

A mesma função pode ser escrita de maneira recursiva em Racket, usando as funções `add1` (incremento) e `sub1` (decremento), sem alterar o valor de nenhuma variável:

```
(define (soma m n)
  (if (zero? n)
    m
    (add1 (soma m (sub1 n)))))
```

Esta função segue exatamente a estrutura dos números naturais: se $n = 0$, sabemos que a soma $m + 0$ é igual a m, essa é a base da recursão. Caso contrário, o resultado é o sucessor de $m + (n - 1)$, que é a chamada recursiva.

Avaliação como substituição

A avaliação de uma função puramente funcional como *soma* pode ser feita como fazemos na manipulação de expressões na álgebra: simplesmente substituindo partes da expressão que são equivalente, até chegar no resultado. Vamos ver como é feita a avaliação da expressão (soma 5 3):

```
(soma 5 3) ≡ (if (zero? 3) 5 (add1 (soma 5 2)))
           ≡ (add1 (soma 5 2))
           ≡ (add1 (if (zero? 2) 5 (add1 (soma 5 1))))
           ≡ (add1 (add1 (soma 5 1)))
           ≡ (add1 (add1 (if (zero? 1) 5 (add1 (soma 5 0)))))
           ≡ (add1 (add1 (add1 (soma 5 0))))
           ≡ (add1 (add1 (add1 (if (zero? 0) 5 (add1 (soma 5 -1))))))
           ≡ (add1 (add1 (add1 5)))
           ≡ (add1 (add1 6))
           ≡ (add1 7)
           ≡ 8
```

É interessante verificar nessa avaliação que cada vez que o segundo parâmetro é decrementado, aparece mais uma chamada a `add1` na expressão. Isso acumula os incrementos ao resultado da base, que é o primeiro parâmetro.

Informalmente, sabemos que essa função sempre termina se o segundo parâmetro for um número inteiro maior ou igual a zero, pois em cada chamada recursiva o segundo parâmetro é reduzido em 1 e a cadeia de chamadas recursivas termina quando este segundo parâmetro for igual a zero; com os decrementos sucessivos, eventualmente chega-se a zero. Assim como acontece com *loops* em linguagens imperativas, também é preciso cuidado com as condições de progresso e parada de uma função recursiva; um erro nesses pontos pode causar um *loop* infinito.

Exercícios

Exercício 1 Cria uma função `(mult m n)` que multiplica os dois números naturais `m` e `n`, usando apenas a operação de soma.

Exercício 2 Crie uma função `(sub m n)` que calcula a subtração de `m` por `n`, usando apenas as operações de incremento (`add1`) e decremento (`sub1`). Você pode assumir que $m \geq n$, mas não é muito mais difícil funcionar mesmo quando $m < n$.

Exercício 3 Crie uma função `(par n)` que retorna `#t` (verdade) se `n` é par, e `#f` caso contrário. Defina também uma função `(impar n)` que retorna `#t` apenas se `n` for ímpar. Podemos considerar zero como sendo um número par. Opcional: se uma das duas funções (`par` ou `impar`) já estiver definida, fica mais fácil definir a outra. Escreva uma função usando a outra. Depois pense se as duas podem ser definidas uma em função da outra, conjuntamente (*recursão mútua*).

Listas

As listas são uma estrutura de dados fundamental na programação funcional, primeiro por motivos históricos: a própria linguagem LISP original (uma sigla que significava *LISt Processor*) foi criada para trabalhar com listas. Mas as listas também são importantes porque permitem agregação de dados em uma estrutura naturalmente recursiva, e por mais outros motivos relacionados a conceitos mais avançados[1].

[1] A manipulação de listas também é a base da *metaprogramação* nas linguagens Lisp.

Podemos definir as listas recursivamente da seguinte forma: uma lista ou é a lista vazia, ou é obtida (*construída*) a partir da adição de um novo elemento na frente de outra lista já existente. Ou seja:

Definição (*Lista*) Uma lista satisfaz uma das seguintes condições:

1. é vazia cuja notação é `'()`; ou
2. é da forma `(cons x l)` onde `x` é um elemento e `l` é uma lista.

Assim como vimos com os números naturais, essa definição pode ser usada para *construir* listas, começando com a lista vazia podemos criar, por exemplo:

- A lista que contém apenas o número 1: `(cons 1 '())`; esta lista está correta porque `'()` é uma lista (regra 1) e adicionar um elemento na frente de uma lista constrói outra lista (regra 2).
- Uma lista com os números 2 e 1, nesta ordem: `(cons 2 (cons 1 '()))`; mais uma vez, usamos a regra 2 duas vezes, e a regra 1 para justificar a lista vazia.

Dessa forma podemos ver como construir qualquer lista, embora de maneira primitiva.

Exercício 4 Usando apenas `cons` e a lista vazia, construa a lista com os números de 1 a 5, em ordem crescente.

A notação usando repetidas ocorrências de `cons` não é muito prática, e por isso existe uma alternativa:

- `(cons 1 '())` pode ser escrita `'(1)`
- `(cons 2 (cons 1 '()))` é o mesmo que `'(2 1)`

e por aí vai. A segunda lista, na verdade, é `(2 1)` apenas, o apóstrofo antes é um detalhe que será explicado mais tarde e tem a ver com a forma que a linguagem Racket avalia expressões.

Uma outra forma de construir listas é usando a função `list`, que simplesmente retorna uma lista com todos os elementos passados como parâmetro, por exemplo `(list 1 2 3)` retorna a lista `'(1 2 3)`.

Exercício 5 Escreva novamente a lista do exercício anterior (números de 1 a 5, em ordem crescente) usando a notação com o apóstrofo, depois repita a mesma lista usando a função `list`.

Sabemos como construir listas, mas para criar funções recursivas para processar listas é necessário sabe como *desconstruir* ou *quebrar* essas listas.

O processo é o contrário da construção: construímos uma nova lista adicionando um elemento à frente de uma lista já existente; para desconstruir, separamos uma lista (se não for vazia) no elemento da frente (o primeiro elemento) e no resto da lista depois do primeiro. Para isso usamos as funções `first` e `rest`. Por exemplo:

- `(first '(1 2 3))` é igual a `1`
- `(rest '(1 2 3))` é igual a `'(2 3)`
- `(first '(2 3))` é igual a `2`
- `(rest '(2 3))` é igual a `'(3)`
- `(first '(3))` é igual a `3`
- `(rest '(3))` é igual a `'()`
- chamar `first` ou `rest` para a lista vazia é um erro

Com essas duas funções temos nossa receita para funções recursivas que devem processar listas: a base da recursão é quando a lista é vazia; nesse caso a função deve dar o resultado diretamente. Se a lista não é vazia, ela é composta por um primeiro elemento (acessado com `first`) e um resto (acessado com `rest`); a chamada recursiva geralmente usa o resto, e o resultado da chamada recursiva pode ser combinada com o primeiro elemento.

Como um exemplo, vamos escrever a função `tamanho`, que calcula o tamanho de uma lista. A ideia é simples: para a base da recursão, a lista vazia tem tamanho zero. Quando a lista não é vazia, ela é composta por um primeiro item e um resto; o tamanho da lista nesse caso vai ser um a mais que o tamanho do resto, que é obtido pela chamada recursiva. Podemos testar se uma lista é vazia chamando a função `empty?`. Em Racket, a função `tamanho` pode ser escrita da seguinte forma:

```
(define (tamanho l)
  (if (empty? l)
      0
      (add1 (tamanho (rest l)))))
```

Note que não importa processar os elementos da lista, por isso nunca precisamos usar a função `first`. Naturalmente, já existe na linguagem Racket uma função para calcular o tamanho de uma lista, e é chamada de `length`. Mas é interessante refazer essas funções básicas para aprender.

Exercício 6 Crie uma função recursiva que calcula a soma dos números em uma lista de números. Qual deve ser a soma de uma lista vazia?

Exercício 7 Crie uma função recursiva que calcula o produto dos números em uma lista de números. Podemos considerar que o produto de uma lista vazia é igual a 1.

Exercício 8 Crie uma função recursiva que, dado uma lista de números (maiores ou iguais a zero), retorna o maior valor entre eles. Assuma que o máximo de uma lista vazia é 0.

Exercício 9 Muitas vezes precisamos transformar os elementos de uma lista de uma mesma maneira. Por exemplo, escreva uma função que, dada uma lista de números como parâmetro, retorna uma outra lista, contendo os elementos da lista original elevados ao quadrado. Ou seja, `(quadrado-lista (list 1 2 3))` deve retornar `'(1 4 9)`.

Exercício 10 Agora vamos selecionar itens em uma lista. Crie uma função que, dado uma lista de números inteiros, retorna uma outra lista, contendo apenas os números pares da lista original.