Recursividade

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Nós vimos como a definição de tipos de dados adequadas é importante no projeto de programas.

Agora vamos explorar como a forma da definição do tipo de dado pode nos ajudar a escrever o corpo das funções.

Considere a seguinte definição de número natural:

- 0 é um número natural;
- Se n é um número natural, então $n + 1$ é um número natural.

O que esta definição tem de diferente?

No segundo caso, um número natural é definido em termos de outro número natural! Como isso é possível!?

Definições recursivas

Como é possível definir uma coisa em termos dela mesmo?

Este tipo de definição é chamada de **definição recursiva** (ou definição indutiva) e é muito utilizada na computação e matemática.

Para ser válida, uma definição recursiva precisa de

- Pelo menos um caso base (que não depende da própria definição)
- Pelo menos um caso com autorreferência (que depende da própria definição para elementos “menores”)

A partir dos casos base, os outros elementos são definidos de forma indutiva pelos casos com autorreferência.

O número 4 é natural?

Vamos verificar

- Como 4 não é zero, para ele ser natural, o 3 tem que ser natural
- Como 3 não é zero, para ele ser natural, o 2 tem que ser natural
- Como 2 não é zero, pare ele ser natural, o 1 tem que ser natural
- Como 1 não é zero, pare ele ser natural, o 0 tem que ser natural
- Por definição, 0 é natural

Portanto, o 4 é natural.

Note que foi preciso decompor o 4 até chegar no caso base.

Assim como temos definições recursivas, também podemos ter funções recursivas.

Uma **função recursiva** é aquela que chama a si mesmo.

Assim como para definições recursivas, para estar bem formada uma função recursiva precisa de

- Pelo menos um caso base (o valor da função é calculado diretamente)
- Pelo menos um caso com chamada recursiva (depende do valor da função para entradas menores)

Como projetar funções recursivas?

Existem várias técnicas de projeto de funções recursivas, nós vamos explorar uma delas, chamada de diminuição e conquista.

A ideia é diminuir o tamanho do problema original, conquistar o problema menor, e estender a solução do problema menor para o problema original.

No início, para diminuir o problema original, nós vamos explorar a relação entre autorreferência na definição do tipo de dado e a chamada recursiva na função que processa o tipo de dado.

Exercício - soma

Projete uma função recursiva que some todos os números naturais menores ou iguais que um determinado *n*.

Especificação

```
// Soma todos os número naturais menores
// ou iguais que n.
//
// Requer que n >= 0.
int soma_naturais(int n)
{
    return 0;
}

examples
{
    check_expect(soma_naturais(0), 0);
    check_expect(soma_naturais(1), 1);
    check_expect(soma_naturais(2), 3);
    check_expect(soma_naturais(3), 6);
    check_expect(soma_naturais(4), 10);
}
```

Como a definição de número natural tem dois casos, vamos começar a nossa função com dois casos.

```
if (n == 0) {
    ...
} else {
    n ...
}
```

Como o segundo caso da definição de número natural tem uma autorreferência, vamos colocar uma chamada recursiva no segundo caso da função.

```
if (n == 0) {
    ...
} else {
    n ... soma_naturais(n - 1);
}
```

Implementação

```
examples
{
    check_expect(soma_naturais(0), 0);
    check_expect(soma_naturais(1), 1);
    check_expect(soma_naturais(2), 3);
    check_expect(soma_naturais(3), 6);
    check_expect(soma_naturais(4), 10);
}
```

```
// Soma todos os número naturais menores
// ou iguais que n.
int soma_naturais(int n)
{
    int soma;
    if (n == 0) {
        // Qual é a soma dos naturais até n == 0?
        soma = ...;
    } else {
        // Tendo a soma dos naturais até n - 1 e o natural n,
        // como obter a soma para os naturais até n?
        soma = n ... soma_naturais(n - 1);
    }
    return soma;
}
```

Implementação

```
examples
{
    check_expect(soma_naturais(0), 0);
    check_expect(soma_naturais(1), 1);
    check_expect(soma_naturais(2), 3);
    check_expect(soma_naturais(3), 6);
    check_expect(soma_naturais(4), 10);
}
```

```
// Soma todos os número naturais menores
// ou iguais que n.
int soma_naturais(int n)
{
    int soma;
    if (n == 0) {
        soma = 0;
    } else {
        soma = n + soma_naturais(n - 1);
    }
    return soma;
}
```

Exercício - exponencial

Projete uma função recursiva que receba como entrada um número $a \neq 0$ e um número natural n e calcule o valor a^n.

Especificação

```
// Calcula a elevado a n.
// Requer que a != 0 e n >= 0.
double potencia(double a, int n)
{
    return 0.0;
}

examples
{
    check_expect(potencia(2, 0), 1);
    check_expect(potencia(2, 1), 2);
    check_expect(potencia(2, 2), 4);
    check_expect(potencia(2, 3), 8);

    check_expect(potencia(3, 0), 1);
    check_expect(potencia(3, 1), 3);
    check_expect(potencia(3, 2), 9);
    check_expect(potencia(3, 3), 27);
}
```

Como a definição de número natural tem dois casos, vamos começar a nossa função com dois casos.

```
if (n == 0) {
    a ...
} else {
    a ... n ...
}
```

Como o segundo caso da definição de número natural tem uma autorreferência, vamos colocar uma chamada recursiva no segundo caso da função.

```
if (n == 0) {
    a ...
} else {
    a ... n ... potencia(a, n - 1);
}
```

Especificação

```
examples
{
    check_expect(potencia(2, 0), 1);
    check_expect(potencia(2, 1), 2);
    check_expect(potencia(2, 2), 4);
    check_expect(potencia(2, 3), 8);

    check_expect(potencia(3, 0), 1);
    check_expect(potencia(3, 1), 3);
    check_expect(potencia(3, 2), 9);
    check_expect(potencia(3, 3), 27);
}
```

```
// Calcula a elevado a n.
// Requer que a != 0 e n >= 0.
double potencia(double a, int n)
{
    double pot;
    if (n == 0) {
        // Qual o valor de a^n quando n == 0?
        pot = a ...;
    } else {
        // Tendo a potência a^(n - 1), o valor
        // de a e n, como calulcar a^n?
        pot = a ... n ... potencia(a, n - 1);
    }
    return pot;
}
```

Especificação

```
examples
{
    check_expect(potencia(2, 0), 1);
    check_expect(potencia(2, 1), 2);
    check_expect(potencia(2, 2), 4);
    check_expect(potencia(2, 3), 8);

    check_expect(potencia(3, 0), 1);
    check_expect(potencia(3, 1), 3);
    check_expect(potencia(3, 2), 9);
    check_expect(potencia(3, 3), 27);
}
```

```
// Calcula a elevado a n.
// Requer que a != 0 e n >= 0.
double potencia(double a, int n)
{
    double pot;
    if (n == 0) {
        pot = 1;
    } else {
        pot = a * potencia(a, n - 1);
    }
    return pot;
}
```

Exercício - fatorial

Projete uma função recursiva que calcule o fatorial de n, isto é, o produto dos n primeiros números naturais maiores que 0.

Especificação

```
// Calcula o fatorial de n, isto é,
// o produto dos n primeiros números
// naturais maiores que 0.
//
// Requer que n >= 0.
int fatorial(int n)
{
    return 0;
}

examples
{
    check_expect(fatorial(0), 1);
    check_expect(fatorial(1), 1);
    check_expect(fatorial(2), 2);
    check_expect(fatorial(3), 6);
    check_expect(fatorial(4), 24);
}
```

Como a definição de número natural tem dois casos, vamos começar a nossa função com dois casos.

```
if (n == 0) {
    ...
} else {
    n ...
}
```

Como o segundo caso da definição de número natural tem uma autorreferência, vamos colocar uma chamada recursiva no segundo caso da função.

```
if (n == 0) {
    ...
} else {
    n ... fatorial(n - 1);
}
```

Especificação

```
examples
{
    check_expect(fatorial(0), 1);
    check_expect(fatorial(1), 1);
    check_expect(fatorial(2), 2);
    check_expect(fatorial(3), 6);
    check_expect(fatorial(4), 24);
}
```

```
// Calcula o fatorial de n, isto é,
// o produto dos n primeiros números
// naturais maiores que 0.
//
// Requer que n >= 0.
int fatorial(int n)
{
    int fat;
    if (n == 0) {
        // Qual é o produto dos primeiros n == 0
        // naturais maiores que 0?
        fat = ...;
    } else {
        // Tendo o fatorial de n - 1 e o natural n,
        // como obter o fatorial de n?
        fat = n ... fatorial(n - 1);
    }
    return fat;
}
```

Especificação

```
examples
{
    check_expect(fatorial(0), 1);
    check_expect(fatorial(1), 1);
    check_expect(fatorial(2), 2);
    check_expect(fatorial(3), 6);
    check_expect(fatorial(4), 24);
}
```

```
// Calcula o fatorial de n, isto é,
// o produto dos n primeiros números
// naturais maiores que 0.
//
// Requer que n >= 0.
int fatorial(int n)
{
    int fat;
    if (n == 0) {
        fat = 1;
    } else {
        fat = n * fatorial(n - 1);
    }
    return fat;
}
```

Aspectos importantes no projeto de funções recursivas

Quando estamos projetando funções recursivas, temos que considerar alguns aspectos:

- A chamada recursiva deve ser feita para um entrada “menor”, dessa forma, temos a certeza que o caso base será alcançado e a função está bem definida.
- Devemos confiar que a chamada recursiva produz a resposta correta e nos preocuparmos apenas em como utilizar essa resposta para calcular o resultado da função.

Podemos projetar funções recursivas que operam em arranjos de forma similar a funções que operam com números naturais.

A ideia é associar um número natural com o arranjo que determina o seu tamanho. Com isso podemos criar funções recursivas que funcionam de forma indutiva no tamanho do arranjo.

Vamos ver alguns exemplos.

Exercício - soma

Projete uma função recursiva que some os elementos de um arranjo.

Especificação

```
// Soma os primeiros n elementos de valores.
// Requer que 0 <= n <= valores.size()
int soma(const vector<int> &valores, int n)
{
    return 0;
}

examples
{
    check_expect(soma({5, 1, 4, 3}, 0), 0);
    check_expect(soma({5, 1, 4, 3}, 1), 5);
    check_expect(soma({5, 1, 4, 3}, 2), 6);
    check_expect(soma({5, 1, 4, 3}, 3), 10);
    check_expect(soma({5, 1, 4, 3}, 4), 13);
}
```

Como a definição de número natural (tamanho do arranjo) tem dois casos, vamos começar a nossa função com dois casos.

```
if (n == 0) {
    ...
} else {
    valores[n - 1] ...
}
```

Como o segundo caso da definição de número natural (tamanho do arranjo) tem uma autorreferência, vamos colocar uma chamada recursiva no segundo caso da função.

```
else {
    valores[n - 1] ... soma(valores, n - 1);
}
```

Implementação

```
examples
{
    check_expect(soma({5, 1, 4, 3}, 0), 0);
    check_expect(soma({5, 1, 4, 3}, 1), 5);
    check_expect(soma({5, 1, 4, 3}, 2), 6);
    check_expect(soma({5, 1, 4, 3}, 3), 10);
    check_expect(soma({5, 1, 4, 3}, 4), 13);
}
```

```
// Soma os primeiros n elementos de valores.
// Requere que 0 <= n <= valores.size()
int soma(const vector<int> &valores, int n)
{
    int s;
    if (n == 0) {
        // Qual é a soma dos n == 0 primeiros
        // elementos de valores?
        s = ...;
    } else {
        // Sabendo a soma dos n - 1 primeiros
        // elementos de valores e valores[n - 1],
        // como obter a soma dos primeiros n
        // elementos de valores?
        s = valores[n - 1] ... soma(valores, n - 1);
    }
    return s;
}
```

Implementação

```
examples
{
    check_expect(soma({5, 1, 4, 3}, 0), 0);
    check_expect(soma({5, 1, 4, 3}, 1), 5);
    check_expect(soma({5, 1, 4, 3}, 2), 6);
    check_expect(soma({5, 1, 4, 3}, 3), 10);
    check_expect(soma({5, 1, 4, 3}, 4), 13);
}
```

```
// Soma os primeiros n elementos de valores.
// Requere que 0 <= n <= valores.size()
int soma(const vector<int> &valores, int n)
{
    int s;
    if (n == 0) {
        s = 0;
    } else {
        s = valores[n - 1] + soma(valores, n - 1);
    }
    return s;
}
```

Exercício - contagem

Projete uma função recursiva que conte quantas vezes um determinado valor aparece em um arranjo.

Especificação

```
// Conta quantas vezes val aparece nos
// primeiros n elementos de valores.
// Requer que 0 <= n <= valores.size()
int freq(int val,
         const vector<int> &valores,
         int n)
{
    return 0;
}

examples
{
    check_expect(freq(1, {5, 1, 4, 1}, 0), 0);
    check_expect(freq(1, {5, 1, 4, 1}, 1), 0);
    check_expect(freq(1, {5, 1, 4, 1}, 2), 1);
    check_expect(freq(1, {5, 1, 4, 1}, 3), 1);
    check_expect(freq(1, {5, 1, 4, 1}, 4), 2);
}
```

Como a definição de número natural (tamanho do arranjo) tem dois casos, vamos começar a nossa função com dois casos.

```
if (n == 0) {
    ...
} else {
    valores[n - 1] ...
}
```

Como o segundo caso da definição de número natural (tamanho do arranjo) tem uma autorreferência, vamos colocar uma chamada recursiva no segundo caso da função.

```
else {
    valores[n - 1] ... freq(val, valores, n - 1);
}
```

Implementação

```
examples
{
    check_expect(freq(1, {5, 1, 4, 1}, 0), 0);
    check_expect(freq(1, {5, 1, 4, 1}, 1), 0);
    check_expect(freq(1, {5, 1, 4, 1}, 2), 1);
    check_expect(freq(1, {5, 1, 4, 1}, 3), 1);
    check_expect(freq(1, {5, 1, 4, 1}, 4), 2);
}
```

```
// Conta quantas vezes val aparece nos
// primeiros n elementos de valores.
// Requer que 0 <= n <= valores.size()
int freq(int val,
         const vector<int> &valores,
         int n) {
    int cont;
    if (n == 0) {
        // Quantas vezes val aparece nos n == 0
        // primeiros elementos de valores?
        cont = ...;
    } else {
        // Sabendo a quantidade de vezes que val
        // aparece nos n - 1 primeiros elementos
        // de valores e valores[n - 1], como obter
        // a quantidade de vezes que val aparece
        // nos primeiros n elementos de valores?
        valores[n - 1] ... freq(val, valores, n - 1);
    }
    return cont; }
```

Implementação

```
examples
{
    check_expect(freq(1, {5, 1, 4, 1}, 0), 0);
    check_expect(freq(1, {5, 1, 4, 1}, 1), 0);
    check_expect(freq(1, {5, 1, 4, 1}, 2), 1);
    check_expect(freq(1, {5, 1, 4, 1}, 3), 1);
    check_expect(freq(1, {5, 1, 4, 1}, 4), 2);
}
```

```
// Conta quantas vezes val aparece nos
// primeiros n elementos de valores.
// Requer que 0 <= n <= valores.size()
int freq(int val,
         const vector<int> &valores,
         int n) {
    int cont;
    if (n == 0) {
        cont = 0;
    } else {
        if (valores[n - 1] == val) {
            cont = 1 + freq(val, valores, n - 1);
        } else {
            cont = freq(val, valores, n - 1);
        }
    }
    return cont;
}
```

Exercício - ordem

Projete uma função recursiva que verifique se os elementos de um arranjo estão em ordem não decrescente.

Especificação

```
// Verifica se os primeiros n elementos de
// valores estão em ordem não-decrescente.
// Requer que 0 <= n <= valores.size()
int ordenado(const vector<int> &valores, int n)
{
    return 0;
}

examples
{
    check_expect(ordenado({1, 2, 3, 2}, 0), true);
    check_expect(ordenado({1, 2, 3, 2}, 1), true);
    check_expect(ordenado({1, 2, 3, 2}, 2), true);
    check_expect(ordenado({1, 2, 3, 2}, 3), true);
    check_expect(ordenado({1, 2, 3, 2}, 4), false);
}
```

Como a definição de número natural (tamanho do arranjo) tem dois casos, vamos começar a nossa função com dois casos.

```
if (n == 0) {
    ...
} else {
    valores[n - 1] ...
}
```

Como o segundo caso da definição de número natural (tamanho do arranjo) tem uma autorreferência, vamos colocar uma chamada recursiva no segundo caso da função.

```
else {
    valores[n - 1] ... ordenado(valores, n - 1);
}
```

Implementação

```
examples
{
    check_expect(ordenado({1, 2, 3, 2}, 0), true);
    check_expect(ordenado({1, 2, 3, 2}, 1), true);
    check_expect(ordenado({1, 2, 3, 2}, 2), true);
    check_expect(ordenado({1, 2, 3, 2}, 3), true);
    check_expect(ordenado({1, 2, 3, 2}, 4), false);
}
```

```
// Verifica se os primeiros n elementos de
// valores estão em ordem não-decrescente.
// Requer que 0 <= n <= valores.size()
int ordenado(const vector<int> &valores, int n)
{
    bool ord;
    if (n == 0) {
        // Os primeiros n == 0 elementos de
        // valores estão em ordem?
        ord = ...;
    } else {
        // Sabendo que os n - 1 primeiros elementos
        // de valores estão em ordem, como determinar
        // se os primeiros n elementos de valores
        // estão em ordem?
        valores[n - 1] ... ordenado(valores, n - 1);
        ord = ...;
    }
    return ord;
}
```

Implementação

```
examples
{
    check_expect(ordenado({1, 2, 3, 2}, 0), true);
    check_expect(ordenado({1, 2, 3, 2}, 1), true);
    check_expect(ordenado({1, 2, 3, 2}, 2), true);
    check_expect(ordenado({1, 2, 3, 2}, 3), true);
    check_expect(ordenado({1, 2, 3, 2}, 4), false);
}
```

```
// Verifica se os primeiros n elementos de
// valores estão em ordem não-decrescente.
// Requer que 0 <= n <= valores.size()
int ordenado(const vector<int> &valores, int n)
{
    bool ord;
    if (n == 0) {
        ord = true;
    } else {
        if (n == 1) {
            ord = true;
        } else {
            ord = valores[n - 2] <= valores[n - 1] &&
                  ordenado(valores, n - 1);
        }
    }
    return ord;
}
```

Implementação

```
examples
{
    check_expect(ordenado({1, 2, 3, 2}, 0), true);
    check_expect(ordenado({1, 2, 3, 2}, 1), true);
    check_expect(ordenado({1, 2, 3, 2}, 2), true);
    check_expect(ordenado({1, 2, 3, 2}, 3), true);
    check_expect(ordenado({1, 2, 3, 2}, 5), false);
}
```

```
// Verifica se os primeiros n elementos de
// valores estão em ordem não-decrescente.
// Requer que 0 <= n <= valores.size()
int ordenado(const vector<int> &valores, int n)
{
    bool ord;
    if (n <= 1) {
        ord = true;
    } else {
        ord = valores[n - 2] <= valores[n - 1] &&
              ordenado(valores, n - 1);
    }
    return ord;
}
```

Implementação

```
examples
{
    check_expect(ordenado({1, 2, 3, 2}, 0), true);
    check_expect(ordenado({1, 2, 3, 2}, 1), true);
    check_expect(ordenado({1, 2, 3, 2}, 2), true);
    check_expect(ordenado({1, 2, 3, 2}, 3), true);
    check_expect(ordenado({1, 2, 3, 2}, 4), false);
}
```

```
// Verifica se os primeiros n elementos de
// valores estão em ordem não-decrescente.
// Requer que 0 <= n <= valores.size()
int ordenado(const vector<int> &valores, int n)
{
    return n <= 1 ||
               valores[n - 2] <= valores[n - 1] &&
                   ordenado(valores, n - 1);
}
```

Esta técnica sempre funciona? Ou seja, se eu aplicar a ideia de diminuir e conquistar eu consigo projetar um algoritmo para resolver qualquer problema?

Não!

Mas então, quando podemos aplicar a técnica de projeto diminuição e conquista?

- Quando conseguimos resolver o caso base
- Quando a solução do problema menor pode ser estendida para a solução do problema original

Se quisermos encontrar todos os divisores de um número n, podemos aplicar essa técnica?

Conseguimos resolver o caso base? Sim.

Tendo os divisores de $n - 1$, podemos encontrar os divisores de n? Sabendo os divisores de $9(1, 3, 9)$, podemos determinar os divisores de $10(1, 5, 10)$? Não!

Então essa técnica não é adequada para esse problema.

Variantes

Podemos diminuir os arranjos de outra forma que não seja pelo seu tamanho (fim)?

Sim, podemos avançar o seu início.

Também podemos aumentar o início e diminuir o final simultaneamente.