

Curso Introdutório de Programação  
(título provisório)

Rosane Minghim e Guilherme P. Telles

24 de Agosto de 2005

# Capítulo 11

## Recursão

### 11.1 Considerações Iniciais

A Recursão é uma característica, comum em matemática, de definir elementos com base em versões mais simples deles mesmos.

Este capítulo apresenta a forma de especificar características recursivas em algoritmos, e ilustra o desenvolvimento de algoritmos recursivos em pseudo-código bem como sua implementação em Pascal.

Como exemplo do conceito de recursão, podemos apresentar duas definições diferentes de *caminhada*.

**Definição não recursiva:** Uma caminhada é uma sequência de passos. O tamanho da caminhada é o número de passos.

**Definição recursiva:** Uma caminhada de tamanho 1 é um passo. Uma caminhada de tamanho  $n$  é uma caminhada de tamanho  $n-1$  seguida (ou precedida) de um passo.

Como exemplo de definição matemática recursiva, temos qualquer declaração indutiva, do tipo:

**Fatorial:**  $0! = 1; n! = n * (n - 1)!$

**Multiplicação por um valor b natural:**  $a * 1 = a; a * b = a + a * (b - 1), b > 1$

**Exponenciação natural:**  $a^1 = a; a^b = a * a^{(b-1)}$

**Exercício Sugerido 39** *Escrever a versão não-recursiva das três definições acima*

## 11.2 Algoritmos recursivos

Em programação, a forma mais comum de recursão é um subprograma que chama a ele próprio.

O exemplo abaixo implementa um subprograma em pseudo-código recursivo para o cálculo do fatorial definido na Seção 11.1.

### Exemplo 11.1 *Fatorial Recursivo*

Subprograma `fatorial(n):inteiro`

`e: n: inteiro {número do qual será calculado o Fatorial}`

`r: inteiro fatorial de n`

pré-condição: `n>0`

```
início
  se n = 1 então
    retorne (1)
  senão
    retorne (n*fatorial(n-1))
  fim se
fim
```

Um subprograma recursivo possui as seguintes características:

- Uma condição de parada, isto é, algum evento que encerre a auto-chamada consecutiva. No caso do fatorial, isso ocorre quando a função é chamada com parâmetro (**n**) igual a 1. Um algoritmo recursivo precisa garantir que esta condição será alcançada.
- Uma mudança de 'estado' a cada chamada, isto é, alguma 'diferença' entre uma chamada e a próxima. No caso do fatorial, o parâmetro **n** é decrementado a cada chamada. Essa mudança

Quando um subprograma recursivo é acionado, a sequência de ações do programa é a mesma de uma chamada convencional. Uma cópia do código do subprograma é executada com os parâmetros estabelecidos. Ou seja, a pilha de passagem de parâmetros cresce com os novos parâmetros e as novas variáveis locais, se existirem. Além disso, o contexto muda, isto é, para

cada chamada, existe um novo conjunto de parâmetros e variáveis, independente das chamadas anteriores. É como se existissem várias cópias do mesmo subprograma.

### Percorrendo um algoritmo recursivo

Para entender o funcionamento da chamada recursiva, imagine que um outro algoritmo irá acionar o subprograma fatorial através do seguinte comando:

```
escreva('fatorial de 5 = ',fatorial(5))
```

A sequência de chamadas ao subprograma é dada por:

```
fatorial(5)
  {n=5}
  fatorial(4)
    {n=4}
    fatorial(3)
      {n=3}
      fatorial(2)
        {n=2}
        fatorial(1)
          {n=1}
          retorna (1)
        retorna (2*1)
      retorna(3*2)
    retorna(4*6)
  retorna(5*24)
```

saída impressa:

```
fatorial de 5 = 120
```

Como pode ser visto, cada chamada à função fatorial gera uma nova execução da função, com um novo valor de  $n$ , independente das demais chamadas. Quando fatorial é chamado com  $n=1$  (`fatorial(1)`), acontece o retorno da função com resultado 1, finalizando o processo de chamadas recursivas. A partir deste retorno, a função `fatorial(2)` retorna também. Seu resultado é 2 multiplicado pelo valor retornado por `fatorial(1)` e assim sucessivamente até o último retorno da função. O valor final (120) é impresso em seguida.

**Exercício Sugerido 40** *Desenvolva subprogramas recursivos para implementar a multiplicação e exponenciação conforme expressas nas definições do início do capítulo.*

A seguir é fornecido mais um exemplo de algoritmo recursivo elaborado a partir de uma definição indutiva.

**Exemplo 11.2** *Soma de vetores recursiva.*

*O algoritmo iterativo para somar os elementos de um vetor é bastante simples. Basta consecutivamente somar os elementos, variando um índice que percorre todos os elementos a serem somados.*

*Indutivamente, pode-se definir a soma dos elementos das  $k$  primeiras posições de um vetor ( $s_k$ ) como a soma dos elementos das  $k-1$  primeiras posições mais o elemento da posição  $k$  do vetor ( $v_k$ ), isto é:*

*Assumindo 1 como o índice da primeira posição do vetor e  $n$  como o número de elementos do vetor, temos:*

$$s_k = s_{k-1} + v(k), 1 \leq k \leq n$$

*e:*

$$s_0 = 0$$

*O algoritmo para o cálculo recursivo da soma dos elementos do vetor é então dado por:*

```

Subprograma soma_vet(v,n):real

e: v:vetor_real
n:inteiro

r: soma dos elementos do vetor v

início
  Se n=0 então
    retorne 0
  senão
    retorne (v[n] + soma(v,n-1))
  fim se
fim

```

*O tipo vetor\_real é dado por:*

*tipo vetor\_real = real [1..MAX]*

O exercício abaixo fornece um outro exemplo de solução recursiva para um problema.

### Exercício Resolvido 11 *Permutações de cadeias de caracteres.*

*Enunciado: Gerar todas as permutações dos caracteres de uma cadeia de tamanho  $n$ .*

*Existem várias abordagens para a resolução deste problema.*

*Uma possibilidade, adotada aqui, é fixar um caracter da cadeia (por exemplo, o último) e permutar os demais. Em seguida trocar este caracter fixado com um outro e repetir o processo, até que ele seja trocado com todos os demais elemento da cadeia, um a um. Cada permutação da subcadeia restante (após fixar um elemento) pode ser feita da mesma maneira.*

*Por exemplo, seja a cadeia: 'abcd'*

*De acordo com o procedimento descrito acima, seriam realizados os seguintes passos:*

*1. fixe 'd' na última posição.*

*gere todas as permutações de 'abc' mantendo 'd' no final.*

*2. Na cadeia inicial 'abcd' troque 'd' com 'a', obtendo 'dbca'. Fixe 'a' na última posição.*

*gere todas as permutações de 'dbc' mantendo 'a' no final.*

*3. Na cadeia 'abcd' troque 'd' com 'b', obtendo 'adcb'. Fixe 'b' na última posição.*

*gere todas as permutações de 'adc' mantendo 'b' no final.*

*4. Na cadeia 'abcd' troque 'd' com 'c', obtendo 'abdc'. Fixe 'c' na última posição.*

*gere todas as permutações de 'abd' mantendo 'c' no final.*

*As permutações de sub-cadeias mencionadas acima podem ser realizadas da mesma maneira, isto é, 'revezando' o elemento da última posição da subcadeia e permutando os anteriores. Ou seja, o processo é recursivo até que a subcadeia atinja tamanho 1.*

*A tabela 11.1 mostra a sequência de permutações. Nela, as cadeias grifadas indicam os passos de fixação de caracteres, e as cadeias não grifadas são as permutações de fato. Os procedimentos 1 a 4 acima correspondem às quatro colunas da tabela.*

*Assim algoritmo recursivo para isso propaga o procedimento de recursão para as subcadeias até que ela tenha tamanho 1. em seguida há a troca e nova chamada da função de permutação. A versão do algoritmo abaixo implementa este procedimento.*

- 1 Subprograma `permute(cad,n)`
- 2 `e`: cadeia: a cadeia de caracteres

1	2	3	4
<u>abcd</u>	<u>dbca</u>	<u>adcb</u>	<u>abdc</u>
<u>abcd</u>	<u>dbca</u>	<u>adcb</u>	<u>abdc</u>
<u>abcd</u>	<u>dbca</u>	<u>adcb</u>	<u>abdc</u>
<u>abcd</u>	<u>dbca</u>	<u>adcb</u>	<u>abdc</u>
<u>bacd</u>	<u>bdca</u>	<u>dacb</u>	<u>badc</u>
<u>cbad</u>	<u>cdba</u>	<u>cdab</u>	<u>dabc</u>
<u>cbad</u>	<u>cdba</u>	<u>cdab</u>	<u>dabc</u>
<u>cbad</u>	<u>cdba</u>	<u>cdab</u>	<u>dabc</u>
<u>cbad</u>	<u>cdba</u>	<u>cdab</u>	<u>dabc</u>
<u>bcad</u>	<u>dcba</u>	<u>dcab</u>	<u>adbc</u>

Tabela 11.1: Tabela de fixações e permutações de caracteres

```

3  n:inteiro : o número de elementos da sub-cadeia a ser permutada
4
5  {este programa imprime todas as permutações dos n primeiros
6  caracteres da cadeia cad}
7
8  início
9      se n=1 então
10         escreva(cad,n)
11     senão
12         fixe o caracter da posição n e permuta a sub-cadeia de n-1 elementos
13         permute(cad,n-1)
14     fim se
15     para i de 1 até n
16         troque o caracter da posição n com um outro da cadeia
17         troque cad[i] com cad[n]
18         fixe o caracter da posição n e permuta a sub-cadeia de n-1 elementos
19         permute(cad,n-1)
20     fim para
21 fim
22

```

Observe que no algoritmo do exercício anterior necessariamente os parâmetros precisam ser passado por valor (i.e., como parâmetro de entrada apenas), pois em cada nível a troca do  $i$ -ésimo caracter pelo último é feito com base na cadeia inicial (veja primeira linha da Tabela ?? e pseudo-código como exemplo) e não na cadeia modificada pelas permutações das subcadeias. Isto é, a chamada da linha 19 ocorre sobre a cadeia `cad` original. A chamada

da linha 13, então não deve permitir mudança nesta variável. Isso é garantido pela passagem por valor. Por consequência dessa grande quantidade de passagens de cadeias de caracteres por valor, aquele subprograma é um caso de operação recursiva que implica num grande gasto de memória. Por outro lado, o processo de permutação em si é bastante lento devido ao grande número de operações, o que torna um programa de recursão para uma cadeia grande inviável mesmo na sua versão não recursiva. Se necessário, entretanto, submeter uma cadeia muito grande ao processo de permutação, é possível que a versão não recursiva seja mais aconselhável.

**Exercício Sugerido 41** *Desenvolver uma versão não-recursiva do algoritmo da permutação do exercício 11.*

Diversas operações encontram na recursão uma solução plausível e sem prejuízos com relação à versão iterativa. Por exemplo, a busca binária em vetores ordenados, desenvolvida no Exemplo ??, não altera o vetor, e também possui uma definição apropriada para uso em recursão.

Na busca binária sobre um vetor ordenado, a cada passo, a busca é reduzida a uma fração do vetor que é aproximadamente metade do vetor anterior. O elemento do meio é comparado com o elemento procurado (x). Se for menor do que o elemento, ele deve ser procurado à esquerda do elemento. Se for maior, só pode estar à direita do elemento comparado. Se for igual, o elemento foi encontrado.

Um subprograma recursivo para resolver a busca binária é dado abaixo.

**Exemplo 11.3** *Busca binária recursiva*

Subprograma `buscabin(v,i,j,x,pos)`

`e:v:vetor` vetor de elementos simples  
`x:elemento` elemento a ser procurado no vetor  
`i:índice inferior` vetor  
`j:índice superior` do vetor

`s: pos: {posição onde o elemento foi encontrado (ou -1 se não for)}`

variável  
`med :inteiro;`



```

início
  Se  $i \leq j$  então
    med  $\leftarrow$  quociente( $(i+j), 2$ )
    se  $v[\text{med}] = x$  então
      pos  $\leftarrow$  med
    senão
      se  $v[\text{med}] < x$  então
        buscabin( $v, \text{med}+1, j, x, \text{pos}$ )
      senão
        buscabin( $v, i, \text{med}-1, x, \text{pos}$ )
      fim se
    fim se
  senão
    pos  $\leftarrow$  -1
  fim se
fim

```

Observe no exemplo acima que o tipo vetor não foi totalmente especificado, uma vez que o algoritmo é o mesmo para qualquer vetor de tipos simples, que possam ser comparados com operadores relacionais.

### Recursão de Cauda

Observe os algoritmos do próximo exemplo, ambos responsáveis por imprimir os elementos de um vetor. Procure percorrê-los e descobrir o resultado antes de prosseguir no texto.

#### Exemplo 11.4 *Impressão recursiva de vetores*

```

1 Subprograma impvet( $v, i, n$ )
2
3 e:  $v$ :vetor
4    $i$ :inteiro posição atual dentro do vetor
5    $n$ :inteironúmero de elementos do vetor
6
7 {subprograma recursivo para impressão do vetor - imprime
8 elementos de  $i$  a  $n$ }
9
10 início
11   Se  $i \leq n$  então

```

```
12     escreva(vet[i])
13     impvet(v,i+1,n)
14   fim se
15 fim

1 Subprograma impvet(v,i,n)
2
3 e: v:vetor
4 i:inteiro posição atual dentro do vetor
5 n:inteironúmero de elementos do vetor
6
7 {subprograma recursivo para impressão do vetor - imprime
8 elementos de i a n}
9
10 início
11   Se  $i \leq n$  então
12     impvet(v,i+1,n)
13     escreva(vet[i])
14   fim se
15 fim
```

Os subprogramas acima têm definições que seriam idênticas não fosse pelas linhas 12 e 13, que estão invertidas. Qual é o efeito dessa 'troca' de linhas. No primeiro algoritmo, o  $i$ -ésimo valor é escrito e então o subprograma é chamado para os elementos subsequentes do vetor. No segundo algoritmo a chamada ocorre antes da impressão. Como consequência, o primeiro algoritmo tem como resultado a impressão dos elementos do vetor na ordem em que se encontram armazenados no vetor, e o segundo imprime os valores na ordem inversa, isto é, o primeiro valor a ser impresso é  $v[n]$ . Percorra a recursão para verificar.

A recursão do primeiro algoritmo acima é chamada de **recursão de cauda**. Nela, existe uma única chamada recursiva e ela é a última coisa que ocorre no algoritmo. A tarefa do algoritmo para os valores atuais de parâmetro já foi concluída, e a chamada seguinte não é dependente deste resultado. No caso do exemplo, já foi feita a impressão, e as demais impressões decorrentes da chamada recursiva são independentes uma da outra. Na recursão de cauda, a memória está sendo preenchida com novas chamadas aninhadas de subprogramas, e uma vez que a última chamada (no exemplo  $\text{impvet}(v,n+1,n)$ ) retorna, ocorre o retorno em sequências de todas as demais chamadas, uma a uma.

Já na recursão do segundo algoritmo (impressão dos elementos do vetor em ordem inversa), a execução do comando da linha 13 precisa que a execução

de todas as demais chamadas do procedimento sejam finalizadas antes que possa ser executada. Existe uma dependência entre a chamada atual e as demais.

A recursão de cauda é considerada mais propensa a ser implementada iterativamente, ao invés de recursivamente, já que a tarefa da chamada atual é cumprida antes mesmo da próxima chamada.

Todo subprograma recursivo pode ser implementado iterativamente. No entanto, os subprogramas recursivos baseados em recursão de cauda têm um padrão que pode ser facilmente mapeado para um processo iterativo.

Um subprograma recursivo de cauda pode ser mapeado para um dos seguintes casos:

- $P \equiv \text{if}(\text{condição})\text{then}\{S; P\}$ .
- $P \equiv S; \text{if}(\text{condição})\text{then}P$ .

As duas estruturas recursivas acima podem ser respectivamente implementadas iterativamente segundo os modelos abaixo:

- $C; \text{while}(\text{condição})\text{then}\{S; C1\}$ .
- $C; \text{repita}\{S; C1\}\text{until}(\text{notcondição})$ .

once C é um comando de inicialização e C1 é um comando de atualização. A seguir é apresentado um exemplo de conversão de formato.

### Exemplo 11.5 Conversão de algoritmos

Subprograma `impvet(v,i,n)`

e: `v:vetor`

`i:inteiro` posição atual dentro do vetor

`n:inteiro` número de elementos do vetor

{subprograma recursivo para impressão do vetor - imprime elementos de i a n}

início

`j <- i`

`enquanto j <= n then`

`escreva(vet[j])`

`j <- j+1`

`fim enquanto`

fim

**Outro tipo de recursão - Recursão Indireta**

Um outro tipo de recursão que pode ser usada para implementar soluções de problemas é a recursão indireta. Dados dois subprogramas a e b, uma recursão indireta é quando a chama b que chama a.

O exemplo abaixo é de um algoritmo recursivo que determina se um número é ímpar ou par. Ele se baseia no princípio de que um número é par se seu antecessor for ímpar e é ímpar se seu antecessor é par.

**Exemplo 11.6**

Subprograma par(x):lógico

e:x:inteiro

r:verdadeiro se x é par,falso caso contrário

início

  Se x=0 então

    retorne(verdadeiro)

  senão

    Se x=1 então

      retorne(falso)

    senão

      retorne (impar(x-1))

    fim se

  fim se

fim

Subprograma ímpar(x):lógico

e:x:inteiro

r:verdadeiro se x é ímpar,falso caso contrário

início

  Se x=0 então

    retorne(falso)

  senão

    Se x=1 então

      retorne(verdadeiro)

    senão

      retorne (par(x-1))

```
        fim se
    fim se
fim
```

Subprograma `par_ou_impár(x):inteiro`

```
e:x:inteiro
r:1 se x é ímpar,2 se x é par

início
    Se par(x) então
        retorne(2)
    senão
        retorne(1)
    fim se
fim
```

Como pode ser visto pela seqüência de exemplos acima, os vários tipos de recursão e a natureza do problema a ser resolvido determinam casos em que o uso de subprogramas recursivos são adequados, inadequados, ou alternativas igualmente válidas a processos iterativos. Um pouco dessa discussão do uso apropriado de recursão é oferecido na próxima seção.

### 11.2.1 Recursão: quando usar e quando não usar

Em princípio, todo processo recursivo pode ser implementado de forma não recursiva. A recursão, do ponto de vista de programação, serve para simplificar o código de algoritmos, tornando-os mais próximos dos conceitos que visam implementar e que sejam naturalmente recursivos.

No entanto, existem alguns riscos associados ao uso da recursão. Para analisar esses riscos é necessário entender como a recursão é implementada em linguagens de programação que admitem o uso deste recurso.

Conforme mencionado anteriormente quando uma função recursiva é chamada, tudo ocorre como uma função qualquer. Cada função é acionada independentemente das demais, com novos parâmetros. Isso significa que cada nova chamada aloca área de variáveis locais todos os parâmetros e todas as variáveis locais da função.

É difícil prever quantas vezes uma função será chamada recursivamente, uma vez que isso depende normalmente do valor do parâmetro. Por exem-

plo, a função fatorial( $n$ ) é chamada  $n-1$  vezes, e a permuta( $n$ ) é chamada  $(n-1)!$  vezes. Como o espaço disponível para alocação de variáveis locais (pilha de passagem de parâmetros) é limitado, existe boa chance de que uma chamada executada muitas vezes (ou com parâmetros muito grandes) ultrapasse o tamanho desta memória e interrompa a execução do programa.

Assim, a cada vez que uma escolha entre um procedimento recursivo e um não recursivo para resolver um problema tenha que ser feita, é preciso estimar a possibilidade de causar 'stack overflow' devido ao excesso de chamadas.

Em muitos casos, como o de recursão de cauda (recursão chamada no final da função), a transformação de um processo recursivo num não recursivo não só é simples como também não compromete o entendimento do código. Este é o momento de optar por soluções não recursivas.

A outra ocasião adequada para evitar recursão é aquela em que a natureza recursiva do problema, se implementada como tal, gera uma ineficiência muito grande se comparada com a versão não recursiva da solução do problema. Um caso clássico é a sequência de Fibonacci.

A sequência de números de Fibonacci é a seguinte:

0, 1, 1, 2, 3, 5, 8, ...

Ou seja, cada número de Fibonacci é a soma dos dois anteriores e os dois primeiros são 0 e 1.

A função que calcula o  $n$ -ésimo número de Fibonacci é definida da seguinte forma:

$fib(n) = 0$  se  $n = 0$ .

$fib(n) = 1$  se  $n = 1$ .

$fib(n) = fib(n - 1) + fib(n - 2)$  se  $n > 1$ .

Implementado recursivamente, o algoritmo para o cálculo do  $n$ -ésimo número de Fibonacci assume a forma do algoritmo abaixo.

```
Subprograma fib(n):inteiro

e:n:inteiro

r:o n-ésimo número de Fibonacci

início
  se n = 0 então
    retorne (0)
  senão
    se n = 1 então
      retorne (1)
    senão
```

```

        retorne (fib(n-1)+fib(n-2))
    fim se
    fim se
fim

```

No algoritmo acima, quando  $\text{fib}(n)$ ,  $n > 1$ , é chamado, ele aciona  $\text{fib}(n-1)$  e  $\text{fib}(n-2)$ .

Suponha que a função tenha sido chamada para  $n=6$ . A sequência de chamadas neste caso fica.

```

fib(6) fib(5) fib(4)  fib(3)  fib(2) fib(1)
                                fib(0)
                                fib(1)
                                fib(2) fib(1)
                                fib(0)
                                fib(3) fib(2) fib(1)
                                fib(0)
                                fib(1)
                                fib(4) fib(3)  fib(2)  fib(1)
                                fib(0)
                                fib(1)
                                fib(2)  fib(1)
                                fib(0)

```

Note-se, na sequência acima, que a função é chamada um número grande de vezes para o mesmo valor do parâmetro. Por exemplo,  $\text{fib}(2)$  é acionada 5 vezes. Isso causa uma sobrecarga tanto de armazenamento quanto de tempo de execução desnecessária.

A versão não-recursiva, apresentada abaixo, possui também clareza adequada e é muito mais eficiente.

Subprograma  $\text{fib}(n)$ :inteiro

$e:n$ :inteiro

$r$ :o  $n$ -ésimo número de Fibonacci

variável

$\text{nfib0}$ ,  $\text{nfib1}$ ,  $\text{nfib}$ ,  $i$ : inteiro

```
início
  se n = 0 ou n=1 então
    retorne (n)
  senão
    nfib0 ← 0
    nfib1 ← 1
    i ← 1
    enquanto i < n faça
      nfib ← nfib0 + nfib1
      nfib0 ← nfib1
      nfib1 ← nfib
      i ← i + 1
    fim enquanto
  fim se
  retorne(nfib)
fim
```

Além desse tipo de recursão, conforme já mencionado anteriormente, é recomendado que se considere uma versão iterativa para o algoritmo em casos de recursões de cauda e recursões que necessitem que de parâmetros grandes sejam passados por valor. Essas recomendações são destacadas a seguir.

**Sugestão 17** *Use recursão quando:*

- *O problema é naturalmente recursivo (clareza) e a versão recursiva do algoritmo não gera ineficiência evidente se comparado com a versão iterativa do mesmo algoritmo.*
- *O algoritmo se torna compacto sem perda de clareza ou generalidade.*
- *É possível prever que o número de chamadas ou a carga sobre a pilha de passagem de parâmetros não irá causar interrupção do processo.*

**Sugestão 18** *Não use recursão quando:*

- *A solução recursiva causa ineficiência se comparada com uma versão iterativa.*
- *A recursão é de cauda.*
- *Parâmetros consideravelmente grandes têm que ser passados por valor*
- *Não é possível prever se o número de chamadas recursivas irá causar sobrecarga da pilha de passagem de parâmetros.*



### 11.3 Recursão em Pascal

Em Pascal a recursão é implementada, do ponto de vista sintático, da mesma forma que em pseudo-código, ou seja, através de procedimentos e funções que chamam a si próprios direta ou indiretamente.

Do ponto de vista de implementação as considerações sobre chamadas de subprogramas e passagem de parâmetros valem, isto é, cada chamada a um subprograma recursivo gera uma nova alocação de parâmetros e variáveis locais e uma chamada independente da anterior. Múltiplas chamadas sobrecarregam a pilha de passagem de parâmetros e variáveis locais dependendo da 'profundidade' da recursão.

A seguir são apresentados dois exemplos de códigos em Pascal dos algoritmos desenvolvidos acima, especificamente os algoritmos do Exercício 11 e do Exemplo 11.3.

**Exemplo 11.7** *Subprograma para a busca binária, versão recursiva*

```
{definição do tipo a ser utilizado}

const
  max_el = 15;

type
  vetor = array[1..max_el] of real;

Procedure buscabin(var vet:vetor;i,j:integer;c:real;var
pos:integer);
{e: vet: vetor com valores ordenados}

  {i,j delimitadores do trecho do vetor aonde procurar o valor
  }
  {c:valor a ser procurado }
{ s:pos : posição onde encontrou o elemento; -1 se o elemento não
se encontra no vetor }

var
  aux:real;
  med:integer;

begin
```

```

if i<=j then
begin
  med := (i+j) div 2;
  if (cad[med]=c) then
    pos := med
  else
    if c < cad[med] then
      buscabin(cad,i,med-1,c,pos);
    else
      buscabin(cad,med+1,j,c,pos);
  end
else
begin
  pos := -1;
end;
end;

```

**Exemplo 11.8** *Subprograma para a permutação de uma cadeia de caracteres, versão recursiva*

```

const
  max_el = 15;

type
  cadeia = string[max_el];

Procedure permute(var cad:cadeia;n:integer);

{e: cadeia: a cadeia de caracteres
 n:inteiro: o numero de elementos da sub-cadeia a ser
 permutada}

{Este subprograma imprime todas as permutações dos n primeiros
 caracteres da cadeia cad}

var
  i:integer;
  aux:char;

```

```

begin
  if n=1 then
    writeln(cad)
  else
    begin
      permute(cad,n-1);
      for i:=1 to n-1 do
        begin
          {troque o caracter da posicao n com um outro da cadeia}
          aux := cad[i];
          cad[i] := cad[n];
          cad[n] := aux;
          {fixe o caracter da posicao n e permute a sub-cadeia
            de n-1 elementos}
          permute(cad,n-1);
        end;
      end;
    end;
end;

```

**Exercício Sugerido 42** *Desenvolver programas principais para testar os subprogramas dos Exemplos 11.7 e 11.8.*

Como pode ser observado nos códigos em geral a tradução de código recursivo para Pascal é imediata.

Um aspecto digno de nota, entretanto, é a recursão indireta, como no caso do Exemplo 11.6. O problema aqui é que, para usar a função ímpar ela tem que ser definida previamente. Para usar a função par, ela também tem que ser definida previamente. Então em qualquer ordem que você defina essas funções você obterá um erro de compilação por identificador não definido.

Neste caso podemos utilizar um recurso do Pascal (o *forward*). Com esse recurso é possível definir o cabeçalho de uma função, mas ‘adiar’ a apresentação do seu código interno para um outro trecho do mesmo arquivo. Com isso o compilador ‘sabe’ a definição dos parâmetros e do retorno, que é tudo que ele precisa para compilar funções que usam aquela definida no ‘forward’. A codificação é apresentada abaixo:

```
Function impar(x:integer):boolean; forward;
```

```
Function par(x:integer):boolean;

{e:x:inteiro
  r:verdadeiro se x é par,falso caso contrário }

begin
  if x=0 then
    par := TRUE
  else
    if x=1 then
      par := FALSE
    else
      par := impar(x-1);
end;

Function impar(x:integer):boolean;

{e:x:inteiro
  r:verdadeiro se x é ímpar,falso caso contrário }

begin
  if x=0 then
    impar:=FALSE
  else
    if x=1 then
      impar:=TRUE
    else
      impar:=par(x-1);
end;
```

**Exercício Sugerido 43** *Codificar em Pascal os demais subprogramas recursivos apresentados neste capítulo.*

## 11.4 Considerações Finais

