9

Programação Imperativa

L''homme est né libre, et partout il est dans les fers."

—Jean-Jacques Rousseau

Assim como o fundamento para a programação funcional é o λ-cálculo, o fundamento para a programação imperativa é o conceito de **máquina de Turing**. A máquina de Turing é uma abstração matemática – proposta pelo pesquisador Alan Turing nos anos 30, daí o nome como ela é conhecida hoje em dia – que corresponde ao conjunto de funções computáveis. Essa caracterização das funções computáveis foi aproximada por John von Neumann a uma arquitetura de computadores que fundamenta os computadores construídos até hoje. Existe, portanto, uma ligação entre a programação imperativa e a arquitetura dos computadores "concretos", que talvez explique parcialmente o sucesso desse paradigma de programação.

Um outro motivo para o sucesso da programação imperativa é a sua fundamentação em si. A essência da programação imperativa (e também da máquina de Turing) se resume a três conceitos:

1. A descrição de estados de uma máquina abstrata por valores de um conjunto de variáveis, sendo que uma variável é um identificador de um *local* – um endereço físico de memória, por exemplo – que atua como repositório para determinado conjunto de valores.

2. Reconhecedores desses estados, que são expressões compostas por relações entre valores e/ou resultados de operações utilizando valores. Alguns desses valores podem ser substituídos por variáveis e nesse caso o valor presente na variável será o valor utilizado na expressão.

3. Comandos, que podem ser de dois tipos:

   (a) comandos de atribuição, que constroem valores efetuando operações a partir de valores preexistentes e atualizam os conteúdos de variáveis;

   (b) comandos de controle, que determinam qual o próximo comando a ser executado.

A execução de um programa imperativo se assemelha, portanto, à simulação da operação de uma máquina física. Cada estado da máquina, uma vez reconhecido, leva a uma seqüência de ações. As ações alteram o estado da máquina, suscitando novas ações e assim por diante até que seja reconhecido um "estado final", que indica a conclusão de uma tarefa.

Uma "máquina física" pode ser um equipamento industrial, toda uma fábrica, uma estrutura administrativa em operação, um modelo econômico de mercado, etc. Podemos depreender daqui o grande espectro de aplicabilidade de programas imperativos.

As linguagens no paradigma imperativo têm sido usadas nas várias áreas de aplicação. Grande parte da automação bancária, por exemplo, é ainda hoje implementada nas linguagens imperativas. Projetos científicos em universidade e órgãos de pesquisa também são, na sua maioria, implementados em linguagens imperativas. Uma das razões para termos a grande aplicação dessas linguagens é de caráter histórico:

os primeiros projetos de linguagens comercialmente utilizadas foram no paradigma imperativo. Uma outra razão é a proximidade dessas linguagens às máquinas existentes, o que faz com que sejam implementadas de forma mais eficiente que algumas linguagens em outros paradigmas.

9.1 VARIÁVEIS, VALORES E TIPOS

Todos os valores e tipos de valores apresentados no Capítulo 3 são encontrados nas linguagens de programação imperativa. Evidentemente que diferentes linguagens apresentam qualidade e versatilidade também diferentes, assim o mais comum é encontrarmos subconjuntos de todos aqueles tipos e valores possíveis em linguagens específicas.

A vinculação dos tipos às variáveis em linguagens de programação imperativa pode ser feita de diversas maneiras distintas (aqui, abreviamos vinculação estática/dinâmica de tipos para tipos estáticos/dinâmicos respectivamente):

Tipos estáticos : se uma linguagem tem tipos estáticos, então as variáveis precisam ser declaradas antes de ser utilizadas, e o tipo de cada variável é determinado quando ela é declarada. Uma variável não pode ser declarada mais de uma vez, portanto, o tipo dos valores que podem ser armazenados no local identificado pelo nome de uma variável é único, fixo e determinado ao longo de todo o programa.

Quando o programa é compilado, o local correspondente a cada variável utilizada é reservado na memória do computador, com um tamanho justo e adequado, de acordo com o tipo dos valores a serem armazenados naquele local. Todas as atribuições de valores a variáveis precisam ser de tipos compatíveis com aqueles das variáveis que recebem os valores. A verificação de compatibilidade pode ser efetuada durante a compilação do programa, liberando a execução da necessidade de novas verificações de tipos. Com isso, os programas se tornam mais eficientes e o código executável mais compacto.

Uma vantagem adicional dos tipos estáticos é requerer do programador uma disciplina de organização de suas idéias mais rigorosa durante a construção dos

programas. As variáveis de um programa imperativo caracterizam os estados possíveis da máquina abstrata que o programa representa. A rigor, esses estados são um subconjunto do conjunto de tuplas compostas pelos valores de todas as variáveis no programa. Os elementos que pertencem a esse subconjunto são determinados por restrições semânticas de integridade, as quais identificam quais valores das variáveis fazem sentido de serem agrupados como valores concomitantes que caracterizam um estado. Os tipos estáticos exigem que o programador considere cuidadosamente esses valores e combinações de valores durante o projeto e construção de um programa.

Tipos dinâmicos fortes : os tipos dinâmicos fortes permitem que valores de diferentes tipos sejam armazenados em uma mesma variável. A única exigência é que as expressões e comandos sejam coerentes com relação a tipos. Ou seja, se uma expressão ou comando contém operações com valores e relações entre esses valores, então essas operações e relações devem fazer sentido no que diz respeito a tipos.

A verificação de compatibilidade de tipos pode ser feita durante a compilação de um programa. Entretanto, durante a execução do programa são necessárias verificações adicionais, para garantir que tipos específicos de valores armazenados nas variáveis sejam compatíveis.

Alguns programadores consideram como uma vantagem a exigência menos rigorosa de disciplina para a construção de programas, o que levaria à construção mais rápida de programas. Essa vantagem é questionável, pois se os programas resultantes forem mais difíceis de analisar e corrigir, o tempo economizado na sua construção pode ser mais do que compensado pelo tempo necessário para a sua manutenção.

Um outro argumento a favor de tipos dinâmicos fortes é a simplificação dos compiladores e interpretadores. É mais fácil construir um compilador ou interpretador com tipos dinâmicos fortes do que um com tipos estáticos. Os compiladores e interpretadores resultantes são mais compactos e podem ser construídos com mais rapidez. Essa vantagem também é questionável, pois os

programas construídos utilizando linguagens com essa característica são sobrecarregados com a necessidade de verificação de tipos em tempo de execução. Necessariamente, portanto, esses programas são mais lentos.

Tipos dinâmicos fracos : esse nome é na realidade um eufemismo para indicar que uma linguagem de programação não verifica os tipos dos valores atribuídos às variáveis. A linguagem não oferece qualquer assistência ao programador para garantir a integridade do programa no que se refere aos tipos de dados. Os compiladores e interpretadores construídos dessa forma são mais compactos e eficientes, mas fica totalmente a cargo do programador garantir a correção das atribuições, operações e relações em seu programa.

9.2 EXPRESSÕES E COMANDOS

Para que a apresentação das expressões e comandos das linguagens de programação imperativa seja mais concisa, vamos classificá-las de uma maneira um pouco diferente da vista no Capítulo 6. Os comandos e expressões são classificados como:

Operações: existem dois tipos básicos de operações:

- **Operações dependentes de tipo**, que acionam uma operação correspondente ao tipo dos valores envolvidos na expressão. Por exemplo, se o tipo dos valores de `x` e `y` é `Inteiro`, temos definida a soma aritmética de números inteiros. A operação `x + y` denota a soma aritmética dos valores de `x` e `y`. Como um outro exemplo, se `L` é uma lista não vazia de valores de tipo `Inteiro`, podemos ter definida a operação `primeiro`, cujo resultado é o primeiro elemento da lista. A operação `primeiro(L)` denota portanto um número de tipo `Inteiro`.

- **Testes**, que verificam a validade de uma relação. O resultado de um teste é sempre `verdadeiro` ou `falso` – um valor booleano, portanto. Por exemplo, se o tipo dos valores de `x` e `y` é `Inteiro`, temos definida a relação binária $\geq$. O teste `x` $\geq$ `y` denota o valor `verdadeiro` se o valor de `x` for maior que ou igual ao valor de `y`, e `falso` em caso contrário. Assumimos a existência

dos conectivos booleanos `e`, `ou` e `não`, que permitem construir testes com grande flexibilidade.

As operações nunca ocorrem isoladas. As operações dependentes de tipo são subexpressões dos comandos de atribuição, e os testes são subexpressões dos comandos de controle. Essas duas classes de comandos são detalhadas a seguir.

Atribuições: por convenção, todas as linguagens de programação imperativa têm um único comando de atribuição. Denotamos aqui esse comando utilizando o operador $\Leftarrow$. Esse operador é binário e infixo. À esquerda do operador encontramos sempre uma variável, e à direita do operador encontramos uma operação dependente de tipo. A verificação de consistência de tipos é efetuada no momento apropriado, dependendo de a linguagem utilizar tipos estáticos, dinâmicos fortes ou dinâmicos fracos. Se o tipo da variável for compatível com o tipo do resultado da operação, então o comando atualiza o valor da variável, que passa a ser o resultado da operação. Por exemplo, se o tipo dos valores de `z` também é `Inteiro`, então o comando `z` $\Leftarrow$ `x + y` atualiza o valor de `z`, que passa a ser o resultado da soma de `x` e `y`.

A atribuição admite formatos especiais para *entrada* e *saída* de dados. Podemos considerar que existem duas variáveis especiais denominadas `Entrada` e `Saída`. A variável `Entrada` ocorre sempre à direita das atribuições e a variável `Saída` ocorre sempre à esquerda das atribuições. Quando um valor é atribuído à `Saída`, valores adicionais de controle são fornecidos que determinam um dispositivo de saída (impressora, tela do computador, auto-falantes, etc.) e um formato apropriado de apresentação de um valor. Nesse caso, em vez de o resultado de uma operação ser armazenado em um local especificado pelo nome da variável, ele é enviado para o dispositivo de saída com o formato especificado. De maneira similar, os valores da variável `Entrada` são obtidos de algum dispositivo de entrada (teclado, mouse, microfone, etc.). A verificação de consistência de tipos ocorre normalmente também para essas variáveis.

Controle: um programa imperativo é uma seqüência ordenada de comandos. Esses comandos podem ser de atribuição ou de controle. Um comando de atribuição atualiza

o valor de uma variável, conforme visto acima. Um comando de controle reconhece o estado da máquina representada pelo programa e permite determinar a seqüência de execução dos comandos.

Sem comandos de controle, os comandos (de atribuição) de um programa são executados na seqüência em que são apresentados no programa. Existem duas categorias de comandos de controle:

1. *seleção:* o comando de seleção é composto por três partes:

   (a) um *teste*;

   (b) um comando ou bloco de comandos que é executado se o resultado do teste for `verdadeiro`;

   (c) um comando ou bloco de comandos que é executado se o resultado do teste for `falso`.

   Detalharemos o conceito de *bloco* mais adiante nesse mesmo capítulo. Basicamente, um bloco é uma subseqüência de comandos.

   O comando de seleção permite selecionar, dentre duas alternativas, um bloco de comandos para ser executado. Denotamos esse comando como `se teste então bloco-verdadeiro senão bloco-falso`. Por exemplo, o comando `se (x > 0) então (y ⇐ 1) senão (y ⇐ 0)` permite escolher qual comando que atribui um valor a `y` deve ser utilizado, em função do valor de `x`.

   As linguagens de programação imperativa, em sua maioria, apresentam uma variedade de comandos de seleção, para comodidade do programador. Os comandos são todos sinônimos, no sentido que para qualquer trecho de programa que utiliza algum dos comandos de seleção, podemos escrever um trecho de programa utilizando um outro comando de seleção que atribui valores equivalentes às mesmas variáveis.

2. *iteração:* o comando de iteração é composto por duas partes:

   (a) um *teste*;

(b) um comando ou bloco de comandos que é executado repetidamente enquanto o resultado do teste for `verdadeiro`. O teste é reavaliado ao final da execução comando ou bloco de comandos.

Denotamos o comando de iteração como `enquanto teste bloco`. Por exemplo, se os valores de `x`, de `y` e de `z` são de tipo `Inteiro`, ao final da execução do trecho de programa abaixo o valor de `y` é x^y (assumindo que o valor fornecido para `x` é maior que 0):

```
x ⇐ Entrada
y ⇐ 1
z ⇐ x
enquanto x > 0
início
  y ⇐ y × z
  x ⇐ x - 1
fim
```

Assim como ocorre com os comandos de seleção, as linguagens de programação imperativa, em sua maioria, apresentam uma variedade de comandos sinônimos de iteração, para comodidade do programador.

9.3 MODULARIDADE

A possibilidade de impor uma estrutura modular a um programa pode ser muito útil. Essa estrutura pode permitir:

- A determinação de um **escopo restrito** para cada nome de variável.
- A associação de **nomes** a trechos do programa utilizados com freqüência, que podem então passar a ser invocados por esses nomes.
- A separação do programa em trechos menores, que podem ser compilados separadamente e até mesmo executados de forma concorrente.

Uma unidade de programa pode ser composta por:

- Uma interface, que pode conter um nome para o módulo e a especificação dos parâmetros de entrada e de saída do módulo (ou seja, variáveis com escopo restrito ao módulo cujos valores são importados para o módulo ou exportados a partir dele).
- A declaração de itens de escopo local ao módulo (variáveis, constantes e tipos). Esses itens existem somente enquanto o módulo é executado.
- Uma seqüência de comandos, incluindo comandos de controle.

Unidade de programa é um nome genérico que se aplica a diversas estruturas de programa. A forma mais simples e comum às linguagens de programação imperativa são os *blocos, funções* e *procedimentos*. Os módulos (ou pacotes) agrupam essas unidades de a forma a obtermos unidades maiores, as quais podem inclusive denotar tipos abstratos, como na linguagem Ada. A seguir, relatamos apenas as formas mais simples e comuns a todas as linguagens.

9.3.1 Blocos

Um bloco é uma seqüência de comandos (incluindo comandos de controle) delimitado por um marcador de `início` e um marcador de `fim` de bloco. Um bloco não tem um nome nem parâmetros de entrada e saída. Essencialmente, blocos são os subprogramas associados aos comandos de seleção e iteração.

A maioria das linguagens de programação imperativa modernas permite o aninhamento de blocos. Algumas linguagens, como por exemplo Ada, permitem ainda a declaração de variáveis com escopo local ao bloco.

Um bloco é executado quando o controle da execução do programa atinge o seu marcador de `início`. O término da execução do bloco ocorre quando o controle de sua execução atinge o marcador de `fim`.

9.3.2 Procedimentos e Funções

Procedimentos e funções são muito similares a blocos, exceto que eles têm nomes e parâmetros. Um nome de procedimento é inserido na seqüência de comandos de um

programa como se fosse um comando da própria linguagem de programação. Quando o controle da execução do programa atinge o nome do procedimento, o procedimento correspondente é ativado.

A forma de um procedimento é:

```
proc NOME (parâmetros)
    declarações de variáveis locais ao procedimento
início
    comandos do procedimento
fim
```

Quando um procedimento é ativado, os seus parâmetros são avaliados. Nesse momento, o controle do programa ativador é "suspenso" e transferido para o procedimento. Ou seja, o programa ativador do procedimento fica "congelado" esperando que o controle interno ao procedimento atinja o marcador de `fim` de procedimento. Quando o procedimento atinge o seu marcador de `fim`, o controle é devolvido ao programa ativador, que segue a partir do comando imediatamente após o nome do procedimento.

Procedimentos e blocos podem ser aninhados. A maioria das linguagens modernas permitem também que um procedimento ative a si mesmo, possibilitando dessa forma a construção de programas recursivos.

Uma função é um procedimento que, quando atinge seu marcador de `fim`, retorna para o programa ativador um valor. Esse valor é retornado como o valor de uma variável cujo nome coincide com o nome da função, e o controle é devolvido ao programa ativador no ponto de chamada da função (ou seja, no ponto onde ocorre o nome da função no programa ativador). Esse ponto pode inclusive fazer parte de uma expressão.

Algumas linguagens (como Pascal) somente permitem a construção de funções escalares. Ou seja, o valor retornado por uma função precisa ser de um tipo primitivo. Outras linguagens (como Ada) permitem a construção de funções cujo valor retornado pode ser de um tipo composto.

9.3.3 Parâmetros

Procedimentos e funções utilizam parâmetros para comunicar valores. Os parâmetros são chamados de parâmetros de entrada quando os seus valores são utilizados dentro do procedimento ou função, e de parâmetros de saída quando os seus valores são os resultados fornecidos pelo procedimento ou função para o programa ativador. Em geral, o valor retornado pela função é o seu único parâmetro de saída em programas bem construídos. Se são necessários mais de um parâmetro de saída, é preferível construir um procedimento do que uma função.

Os parâmetros ocorrem em dois pontos no programa: na declaração do módulo e na sua ativação. Para diferenciá-los, denominamos os parâmetros da declaração de uma função ou procedimento de seus **parâmetros formais**, e os parâmetros na ativação da função ou procedimento de **parâmetros atuais**.

Existem duas maneiras de relacionar parâmetros formais e parâmetros atuais:

1. **Relacionamento por palavra-chave**: o mesmo nome de variável deve ser utilizado para relacionar um parâmetro formal e um parâmetro atual. Antes de ativar uma função ou procedimento, a variável com nome apropriado deve receber o valor desejado.

2. **Relacionamento posicional**: a posição de cada parâmetro nas seqüências de parâmetros formais e atuais determina o seu relacionamento. Assim, o valor do primeiro parâmetro atual é transmitido para o primeiro parâmetro formal, o valor do segundo parâmetro atual é fornecido para o segundo parâmetro formal, e assim por diante.

O relacionamento posicional é muito mais comum que o relacionamento por palavra-chave.

Existem também diversos métodos para efetivamente passar parâmetros para procedimentos e funções. Os métodos mais utilizados são:

1. **Passagem por valor**: o valor de cada parâmetro atual é calculado no contexto do programa ativador, e então transmitido para o parâmetro formal correspondente. O parâmetro formal passa a atuar como uma variável com escopo local

ao módulo ativado, cujo valor inicial é aquele determinado pelo parâmetro atual.

2. **Passagem por referência**: o valor de cada parâmetro atual é uma referência para o próprio local onde são armazenados os valores de uma variável. Uma referência explícita ao valor armazenado naquele local permite a consulta e atualização daquele valor, independente do escopo do módulo. Essa é a maneira mais comum de possibilitar que um procedimento atualize valores de variáveis, de forma que esses valores sejam visíveis para o programa ativador.

   Se x é uma variável, denotamos como $\uparrow x$ o local de armazenamento de valores ocupado por x. Temos dessa forma um tipo de dados *local de Tipo*, para cada *Tipo* presente na linguagem. Por exemplo, se temos em uma linguagem de programação imperativa o tipo `Inteiro`, temos também o tipo *local de* `Inteiro` – denotado como $\uparrow$ `Inteiro`. Uma variável declarada com esse tipo tem como valores admissíveis localizações (na memória do computador, por exemplo) cujos conteúdos são valores de tipo `Inteiro`.

   Se o tipo da variável `x` é $\uparrow$ `Inteiro` e o tipo da variável `y` é `Inteiro`, a atribuição `x` $\Leftarrow\uparrow$ `y` faz sentido: o local onde são armazenados os valores atribuídos a `y` passa a ser o valor de `x`. Uma atribuição como essa só pode ser útil se tivermos acesso ao valor armazenado no local indicado pelo conteúdo de `x`. Denotamos como $*x$ o valor armazenado no local que é o conteúdo de x. Assim, a atribuição `y` $\Leftarrow$ `* x` também faz sentido: o valor de `y` passa a ser o valor que se encontra no local indicado pelo valor de `x`.

   A passagem de parâmetro por referência tem um efeito similar aos apontadores; compartilhamento de espaço de memória. Algumas linguagens, tal como C, usam apontadores explicitamente para denotar a passagem de parâmetros por referência.

9.4 UM EXEMPLO

Para tornar mais concreta a discussão desenvolvida até aqui, vamos ilustrar os conceitos vistos nesse capítulo através de um exemplo prático. O programa calcula, dados dois números naturais p e q, o valor $\frac{p!}{q! \times (p-q)!}$ se $p \geq q$, ou $\frac{q!}{p! \times (q-p)!}$ caso contrário.

Vamos assumir que nossa linguagem de programação tem tipos estáticos. Precisamos, portanto, antes de mais nada em nosso programa declarar as variáveis `p` e `q` com os tipos apropriados, para que elas passem a ser identificadores de locais na memória do computador. As linguagens de programação imperativa não costumam contar com um tipo de dados `Natural`, mas sim com um tipo de dados `Inteiro`. Nosso programa inicia com a criação das variáveis `p` e `q`, cujo tipo é `Inteiro`.

O próximo passo é atribuir valores para essas variáveis. Os valores são provenientes de `Entrada`. Se valores com tipos incompatíveis com o tipo das variáveis criadas forem fornecidos, a própria linguagem deve interromper a execução do programa. Os valores podem, entretanto, ser negativos. Nesse caso, eles seriam compatíveis com o tipo de dados `Inteiro`, mas não seriam números naturais. Precisamos então verificar, dentro do programa, se os valores fornecidos são naturais.

Caso os dois valores sejam naturais, precisamos verificar se o valor de `p` é maior que ou igual ao valor de `q`. Em nosso exemplo, criamos um procedimento que funciona da seguinte maneira: caso o valor de `p` seja maior que ou igual ao valor de `q`, o procedimento não faz nada. Caso contrário, os valores são trocados. Essa troca deve ser efetuada dentro do procedimento, porém o seu efeito deve ser refletido no programa ativador. Os parâmetros do procedimento devem, portanto, ser passados por referência.

Após a execução desse procedimento, temos certeza que o valor de `p` é maior que ou igual ao valor de `q`. O próximo passo é simplesmente calcular o valor de $\frac{p!}{q! \times (p-q)!}$. Criamos para isso uma função.

Essa função utiliza repetidamente o cálculo do fatorial. Criamos assim uma outra função para calcular o fatorial, que será utilizada pela função anterior. Um programador um pouco mais experiente deve reconhecer que existem maneiras muito mais

eficientes de construir esse programa, mas nesse exemplo preferimos essa alternativa menos eficiente para ilustrar, de forma mais explícita, os conceitos desejados.

O programa deve então ficar assim:

```
declaração de variáveis
  p, q:  Inteiro
Início
  p ⇐ Entrada
  q ⇐ Entrada
  se (p ≥ 0 e q ≥ 0) então
  Início
    OrdenaDecrescente( ↑ p, ↑ q )
    Saída ⇐ CalculaResultado( p, q )
  Fim
  senão
    Saída ⇐ ''Dados Inadequados''
Fim
```

O procedimento `OrdenaDecrescente` deve ficar assim:

```
Procedimento OrdenaDecrescente(a: ↑ Inteiro, b: ↑ Inteiro)
declaração de variáveis locais
  c: Inteiro
Início
  se (*a < *b) então
  Início
    c ⇐ *a
    *a ⇐ *b
    *b ⇐ c
  Fim
Fim
```

Assumimos nesse exemplo que o relacionamento entre os parâmetros atuais e formais é posicional. No caso do procedimento `OrdenaDecrescente`, o comando de seleção não apresenta a parte correspondente ao `senão`. Isso porque, nesse caso específico, se o teste tiver como resultado `falso`, nenhuma ação deve ser efetuada.

A função `CalculaResultado` deve ficar assim:

```
Função CalculaResultado(a: Inteiro, b: Inteiro): Inteiro
declaração de variáveis locais
  c: Inteiro
Início
  c ⇐ (Fatorial(a) / (Fatorial(b) × Fatorial(a-b)))
  Retorna c
Fim
```

Devemos observar que a própria função tem um tipo de dados especificado, que é o tipo dos valores gerados por ela. O último comando na função indica qual o valor a ser retornado.

Finalmente, a função `Fatorial` deve ficar assim:

```
Função Fatorial(a: Inteiro) : Inteiro
declaração de variáveis locais
  b: Inteiro
Início
  b ⇐ 1
  enquanto (a > 1)
  Início
    b ⇐ b × a
    a ⇐ a - 1
  Fim
  Retorna b
Fim
```

9.5 A LINGUAGEM PASCAL

A linguagem Pascal foi desenvolvida pelo Prof. Niklaus Wirth e apresentada em 1970. Essa linguagem de programação imperativa foi desenvolvida tendo em vista especificamente o ensino de programação.

Pascal utiliza tipos estáticos e relacionamento posicional entre parâmetros de funções e procedimentos. Essa linguagem de programação se mostra muito bem-sucedida tanto em seu objetivo inicial de construção como para o desenvolvimento de programas eficientes e de qualidade para os mais diversos conceitos. Diversas novas linguagens e implementações de ambientes de programação comerciais se fundamentam em Pascal, dentre eles o bem conhecido ambiente de programação Borland Delphi.

Diversas implementações com código aberto de compiladores Pascal podem ser encontradas pela Internet. Destacamos o projeto Free Pascal, que distribui um compi-

lador a partir de `http://www.freepascal.org`. Esse compilador é distribuído com licença de utilização GNU, é relativamente compacto, simples de instalar e disponível para diferentes computadores e sistemas operacionais.

Em Pascal, a seqüência de apresentação das funções e procedimentos é importante. Funções e procedimentos devem ser declarados antes de serem utilizados.

As seguintes correspondências podem ser identificadas entre os comandos vistos na seção anterior e a sintaxe apropriada em Pascal:

Comando	Pascal
início de programa	`program NOME;`
declaração de variável	`var NOME-DA-VARIÁVEL: TIPO-DA-VARIÁVEL`
início de bloco	`begin`
fim de bloco	`end`
declaração de procedimento	`procedure NOME(PARAMa:TIPOa;PARAMb:TIPOb...)`
declaração de função	`function NOME(PARAMa:TIPOa;PARAMb:TIPOb...):TIPO-DA-F`
seleção	`if TESTE then BLOCO-V else BLOCO-F`
iteração	`while TESTE do BLOCO`
atribuição (⇐)	`:=`
tipo de dados Inteiro	`Integer`
aritmética de inteiros	`+, -, *, div`
ordem natural dos inteiros	`<, >, <=, >=, <>`
entrada de dados	`readln(VARIÁVEL)`
saída de dados	`writeln(VARIÁVEL)`

DRAFT 3 de fevereiro de 2003, 5:52pm D R A F T

Por convenção, todo comando em Pascal termina com um ponto-e-vírgula (`;`), exceto o `end` que marca o final do programa principal, o qual termina com um ponto.

Quando um parâmetro formal deve ser recebido por valor, somente o nome e o tipo da variável correspondente devem ser declarados. Quando um parâmetro formal deve ser recebido por referência, o nome da variável correspondente deve ser precedido da palavra chave `var`.

Evidentemente, uma descrição completa dessa linguagem de programação requer muito mais detalhes, aqui apresentamos apenas alguns de seus elementos para que possamos complementar nosso exemplo na linguagem. O programa Pascal que implementa o exemplo da seção anterior pode ser escrito como:

```
program comb;

procedure OrdenaDecrescente(var a: Integer; var b: Integer);
  var c: Integer;
  begin
     if (a < b) then
     begin
        c := a;
        a := b;
        b := c
     end
  end;

function Fatorial(a: Integer): Integer;
  var b: Integer;
  begin
     b := 1;
     while (a > 1) do
     begin
        b := b * a;
```

```
        a := a - 1
    end;
    Fatorial := b
end;

function CalculaResultado(a: Integer; b: Integer): Integer;
  var
    c: Integer;
  begin
    c := Fatorial(a) div (Fatorial(b) * Fatorial(a-b));
    CalculaResultado := c
  end;

  var
    p,q:Integer;
  begin
    readln(p);
    readln(q);
    if ((p >= 0) and (q >= 0)) then
    begin
       OrdenaDecrescente(p,q);
       writeln(CalculaResultado(p,q))
    end
    else
       writeln('dados inadequados')
  end.
```

Vários outros exmplos foram vistos na Parte I, quando da apresentação dos conceitos fundamentais de linguagens de progrmação. Neste capítulo evitamos repetir os tipos predefinidos bem como os comandos já mencionados nos Capítulos 3 e 6.

9.6 LEITURA RECOMENDADA

Como este é um paradigma de programação vastamente utilizado, existem diversos bons livros que tratam especificamente da programação imperativa. Recomendamos em especial o capítulo do livro [64] dedicado a esse tema e o texto de I. C. Wand [63].

Vários livros já foram publicados sobre a boa prática de programação, desde alguns mais antigos como [28] a alguns mais modernos que resumem resultados sobre anos de prática de desenvolvimento de sistemas [39]. A abordagem formal de desenvolvimento, a qual prima por programas corretos, também descreve boas práticas de programação e pode ser vista tanto com o uso de uma linguagem específica (ALGOL) como em [25], quanto com um método formal, como em [37].

Existem hoje várias linguagens imperativas comercialmente usadas. A linguagem C tem sido vastamente utilizada porque se mostra uma linguagem eficiente para vários tipos de problemas. Existe uma vasta literatura sobre este linguagem disponível hoje.

9.7 EXERCÍCIOS

Construir programas imperativos para resolver os seguintes problemas:

1. Dada uma seqüência com n números inteiros, determinar quantos desses números são pares.

2. Dados dois números inteiros positivos, determinar o máximo divisor comum entre eles.

3. Dados n números inteiros positivos, determinar quantos são primos.

4. Dada uma seqüência com n inteiros positivos, imprimi-los na ordem inversa à ordem da leitura.