Introdução à Programação

Ulysses de Oliveira

Professor Adjunto do Departamento de Informática
do Centro de Ciências Exatas e da Natureza da
Universidade Federal da Paraíba

Editora Universitária
Universidade Federal da Paraíba
João Pessoa

Ficha Catalográfica
Biblioteca Central da UFPB. Catalogação na fonte

Oliveira, Ulysses de

O48i Introdução à Programação / Ulysses de Oliveira .- João Pessoa : Editora Universitária, 1999.

ISBN 85-XXX-XXXX-X

190p. : il.
Inclui apêndice

1. Programação (computadores) 2. Pascal 3. Algoritmos I. Título

C.D.U.: 681.3.06

O Autor

Ulysses de Oliveira é Professor Adjunto do Departamento de Informática do Centro de Ciências Exatas e da Natureza da Universidade Federal da Paraíba. Ele graduou-se em Engenharia Civil (1982) e em Física (1983) nesta Universidade. Em 1988, obteve o título de Mestre em Ciências no Instituto Militar de Engenharia (IME), no Rio de Janeiro, na área de Ciências da Computação. Em 1996, obteve o título de PhD em Ciências Cognitivas e da Computação na Universidade de Sussex, na Inglaterra.

Prefácio

Este texto foi elaborado para um curso introdutório de programação de um semestre, e é oriundo de notas-de-aula continuamente refinadas durante vários semestres de ensino da disciplina Introdução à Programação no curso de Ciências da Computação da Universidade Federal da Paraíba (Campus I). Com este livro esperamos preencher uma lacuna existente devido à deficiência de textos nacionais nesta área.

Este livro destina-se não apenas a alunos de Computação como também a alunos de Engenharia e áreas afins. Quando utilizado por alunos de Computação, todo o conteúdo pode ser abrangido em um único semestre. Como livro-texto para cursos de Iniciação à Computação para alunos de outras áreas, alguns tópicos mais avançados (como, por exemplo, tratamento de arquivos e estruturas dinâmicas) podem ser omitidos.

O texto não pressupõe conhecimento prévio de computação, embora seja desejável, pelo menos, que o estudante tenha cursado ou esteja cursando, paralelamente, uma disciplina de introdução aos princípios básicos de computação ou de organização de computadores. Aconselha-se ao aluno menos experiente que estude os Apêndices A e B antes de tentar construir algum programa. É também aconselhável para estes alunos que iniciem a prática de programação editando e executando os programas-exemplo mais simples apresentados no livro antes de tentarem construir seus próprios programas. Fazendo isso, o aluno familiarizar-se-á com o ambiente de programação utilizado sem ter que estar envolvido com outra tarefa mais complexa.

O Capítulo 1 apresenta uma breve introdução às linguagens de programação e inclui conceitos básicos necessários para o entendimento do restante do texto. O Capítulo 2 explora a noção de algoritmo e de outros tópicos intimamente relacionados a este conceito. Os detalhes da notação utilizada na linguagem algorítmica ali apresentada não são importantes e algumas variações e adições podem ser adotadas. Atenção especial deve ser dedicada a este capítulo, uma vez que o conceito de algoritmo é fundamental em programação (mesmo considerando o paradigma de programação orientada a objetos). No Capítulo 3, as construções da linguagem Pascal são introduzidas. Aconselha-se a alunos e a instrutores que se adote um ritmo de estudo mais lento do que aquele a ser adotado nos capítulos seguintes, visto que, nessa etapa do curso, o aluno ainda não está familiarizado com a sintaxe da linguagem Pascal. Neste estágio, também se aconselha que os passos sugeridos nas Seções 2.4 e 3.13 sejam rigorosamente seguidos nas construções dos programas iniciais. À medida em que os alunos forem se familiarizando com a linguagem Pascal, alguns destes passos podem ser abreviados ou fundidos. Os capítulos subseqüentes apresentam tópicos mais avançados da linguagem Pascal. Alguns destes tópicos podem ser omitidos, a critério do instrutor, dependendo

do nível do aluno-alvo do curso. O Apêndice A apresenta noções básicas de sistemas operacionais e é dirigido àqueles alunos sem nenhum conhecimento prévio de computação. O Apêndice B contém material básico necessário para a utilização de uma das versões mais simples do compilador Turbo Pascal. O material apresentado neste apêndice deve ser complementado ou substituído se outro compilador for utilizado. Finalmente, o Apêndice C contém uma lista de verificação para programas escritos em Pascal. Esta lista de verificação pode ser utilizada como referência na depuração de programas em Pascal.

Vários exemplos de programas e algoritmos são apresentados ao longo do texto. Alguns destes exemplos são propositadamente incompletos e espera-se que o aluno não tenha dificuldade em completá-los. A maioria destes exemplos foi testada por alunos, monitores e pelo próprio autor durante vários semestres, quando parte do texto foi utilizado como notas-de-aula de Introdução à Programação. Não houve tentativa de otimizar os programas-exemplo apresentados. Ao invés disto, procuraram-se apresentar soluções didáticas e simples de serem entendidas.

A linguagem utilizada neste texto é Pascal. Embora esta linguagem seja um tanto antiga e pouco utilizada na construção de programas comerciais, ela ainda nos parece ser a mais indicada para o ensino introdutório de programação tanto por ser simples quanto por abranger estruturas de dados e de controle encontradas nas linguagens de programação mais utilizadas na prática profissional.

Programação é uma disciplina que se aprende apenas com muita prática. Por isso, o material complementar que o aluno deve ter disponível para acompanhamento deste curso é um computador e um ambiente de programação Pascal. Algumas sugestões para um melhor aprendizado das técnicas de programação apresentadas aqui são:

- Edite e execute os exemplos apresentados no texto, tentando entender como estes funcionam na prática.

- Faça modificações nos exemplos apresentados e tente responder questões associadas a tais modificações (por exemplo, “O que acontece quando eu troco uma instrução por outra?”, “O programa continua funcionando como deveria?”, “O desempenho do programa melhora com esta mudança?”, “Esta mudança torna a solução mais elegante?”).

- Finalmente, resolva os exercícios de programação propostos no final de cada capítulo.

Gostaríamos de agradecer a alunos, a monitores e a professores do Departamento de Informática da UFPB por indicarem falhas em versões anteriores do texto e apresentarem sugestões para melhoria do mesmo. Em especial, gostaríamos de agradecer ao Professor Raimundo Nóbrega pelas inestimáveis sugestões e incentivos oferecidos durante as várias etapas de elaboração do texto. Finalmente, agradecemos à Editora Universitária da Universidade Federal da Paraíba pelo apoio dado para a publicação desta obra.

Ulysses de Oliveira

Conteúdo

Capítulo 1

INTRODUÇÃO ÀS LINGUAGENS DE PROGRAMAÇÃO

1.1 Histórico de Linguagens de Programação

Para um computador executar uma dada tarefa é necessário que se informe a ele, de uma maneira clara, **como** ele deve executar aquela tarefa. Em outras palavras, para cada tarefa a ser executada, o computador deve ser **programado**. Infelizmente, a **linguagem nativa** dos computadores convencionais (i.e., a linguagem que eles entendem a nível de máquina) é muito limitada em dois aspectos fundamentais. Primeiro, qualquer construção nesta linguagem deve ser representada por números binários formados apenas a partir de zeros (0) e uns (1) (cada um destes dígitos é chamado de **bit**). Além disso, os computadores entendem apenas um conjunto relativamente pequeno de instruções como, por exemplo, "mova este número de uma posição para outra em memória" ou "some estes dois números inteiros". Cada computador (ou, mais precisamente, cada CPU de computador) possui um conjunto específico de instruções deste tipo denominado de o **conjunto de instruções** do computador.

Programas escritos na linguagem binária (i.e., usando 0s e 1s) usando o conjunto de instruções de um computador são ditos serem programas em **linguagem de máquina**, e este tipo de programação é considerada como **programação de baixo nível**. Como você já deve ter desconfiado, programas escritos em linguagem de máquina são difíceis de escrever, ler e modificar (para nós, humanos, não para os computadores).

Um grande melhoramento sobre a linguagem de máquina foi o surgimento da **linguagem assembly** (ou **linguagem de montagem**), que usa mnemônicos para representar instruções originalmente escritas em linguagem de máquina (por exemplo, uma instrução para adicionar números identificada por 10011011 em linguagem de máquina poderia ser identificada como `ADD` em *assembly*). A linguagem *assembly* também permite que sejam atribuídos nomes a variáveis (i.e., posições na memória do computador), por exemplo, uma variável identificada por 00111010 em linguagem de máquina poderia ser identificada (a critério do programador) por (digamos) X. Como a linguagem *assembly* é estranha ao computador (que só entende 0s e 1s), programas escritos nesta linguagem requerem um programa especial, chamado **assembler** (ou

montador), que traduz instruções escritas em *assembly* para instruções em linguagem de máquina.

Apesar de ter sido um melhoramento considerável (para o programador) com relação à linguagem de máquina, programação em linguagem *assembly* ainda é difícil, e este tipo de programação ainda é considerado de baixo nível. No início da era dos computadores (i.e., de meados da década de 40 até meados da década de 50), todos os programas de computador eram de baixo nível (i.e., escritos em linguagem de máquina ou em *assembly*).

Uma facilidade ainda maior para o programador foi a introdução das chamadas **linguagens de alto nível**, que permitem que programas sejam escritos usando uma linguagem mais *próxima* da linguagem humana. Apesar desta *proximidade*, as linguagens de programação de alto nível não podem ser consideradas como um meio de comunicação entre seres humanos. Em vez disso, elas constituem um forma de comunicação entre homens e computadores. Isto é, através delas o programador pode comandar, por meio de instruções, o computador a realizar uma tarefa desejada. O grande trunfo das linguagens de alto nível é que elas permitem que se especifiquem instruções sem a preocupação sobre detalhes do computador no qual o programa será executado. A maioria dos programas de hoje são escritos em linguagens de alto nível, mas muitos programadores experientes ainda usam a linguagem *assembly* por questões de eficiência (veja mais adiante).

1.2 Compiladores e Interpretadores

Todo programa escrito numa linguagem de programação que não seja a própria linguagem de máquina de um computador requer uma **tradução** antes de ser executado neste computador. Já foi visto acima que a linguagem *assembly* (que é uma linguagem de baixo nível) requer um tipo especial de **tradutor**, chamado *assembler* (ou montador), que traduz programas escritos nesta linguagem para programas em linguagem de máquina. Como foi visto acima, a linguagem *assembly* evita que o programador escreva programas usando seqüências de números binários da linguagem de máquina através do uso de identificadores simbólicos (mnemônicos e nomes de variáveis) para estas seqüências de números. Existe portanto uma correspondência um-a-um entre instruções de um programa em linguagem *assembly* e a tradução deste programa em linguagem de máquina. Em outras palavras, a cada instrução escrita em *assembly*, existe uma única instrução em linguagem de máquina correspondente. Por isso, o *assembler* é o tipo de tradutor mais simples que existe.

Linguagens de alto nível requerem tradutores mais complicados, visto que, usualmente, uma única instrução em linguagem de alto nível pode corresponder a muitas instruções em linguagem de máquina. Existem dois tipos de tradutores para linguagens de alto nível: **compiladores** e **interpretadores**.

Um compilador é um programa que traduz programas escritos numa linguagem de alto nível (**programas fonte**) em programas (**programa objeto**) escritos numa linguagem de baixo nível (linguagem de máquina ou *assembly*) de um computador. Um compilador depende não apenas da linguagem para a qual ele foi projetado, mas também do computador no qual programas escritos nesta linguagem devem ser traduzidos. Assim, é comum falar-se de um compilador (da linguagem) X (por exemplo, Pascal) para o computador Y (por exemplo, *Macintosh*).

Usualmente, um programa escrito numa linguagem de alto nível tem várias partes que são compiladas separadamente. Para reunir as várias partes de um programa de modo a formar uma única unidade executável, utiliza-se um programa chamado de **link editor** (ou editor de ligações). Um *link* editor é um programa que recebe como entrada um conjunto de programas compilados e faz as devidas ligações entre estes programas. O produto resultante do *link* editor é um programa integrado pronto para ser executado.

O segundo tipo de tradutor é o **interpretador**. Diferentemente do compilador, um interpretador não traduz um programa inteiro em seu equivalente em linguagem de máquina. Isto é, um interpretador não produz programas objeto. Em vez disso, ele simula um computador cuja linguagem de máquina seria aquela sendo traduzida. Assim, cada instrução do programa (em linguagem de alto nível) é traduzida exatamente antes de ser executada e de acordo com o fluxo de execução do programa. Isto significa que instruções que são executadas mais de uma vez, também são traduzidas este mesmo número de vezes; instruções que não são executadas não são traduzidas. Em contraste, um compilador traduz todas as instruções de um programa exatamente uma vez.

Outra diferença importante entre compiladores e interpretadores é que, uma vez compilado, um programa torna-se independente do compilador (i.e., torna-se um programa ***standalone***), enquanto que um programa interpretado depende sempre do interpretador para ser executado. Finalmente, programas compilados executam mais rapidamente do que programas equivalentes interpretados (por que?).

Os programadores (usualmente) não precisam se preocupar em entender como um tradutor traduz seus programas para linguagem de máquina durante a elaboração de um programa, mas este conhecimento pode ser essencial na fase de depuração do programa.

1.3 Tipos de Linguagens

1.3.1 Linguagens de Baixo Nível x Linguagens de Alto Nível

Linguagens de alto nível oferecem vantagens ao programador que podem ser analisadas segundo as seguintes propriedades desejáveis de um programa:

• **Portabilidade.** Um programa escrito em linguagem de alto nível pode ser transportado para qualquer computador que tenha um tradutor apropriado. Em contraste, um programa escrito em baixo nível é específico para um tipo de computador; i.e., se o programador desejar executar este programa num outro computador, o programa terá que ser reescrito.

• **Legibilidade**. Ao contrário das linguagens de baixo nível, as linguagens de alto nível oferecem notações que se assemelham às linguagens humanas. Como conseqüência, programas escritos em linguagens de alto nível são muito mais fáceis de serem escritos e lidos do que os correspondentes programas escritos em linguagem de baixo nível. Entretanto, note que escrever um programa numa linguagem de alto nível não garante esta propriedade: a legibilidade de um programa depende também (e muito) do próprio programador. Um dos objetivos deste livro é prover orientações para a escrita de programas legíveis e fáceis de serem modificados.

• **Manutenibilidade**. A facilidade que um programa tem de ser mantido (i.e., modificado) é denominada manutenibilidade. Esta propriedade está intimamente relacionada com a legibilidade: quanto mais fácil de ser lido um programa é, mais fácil ele será de ser modificado. Portanto, programas escritos em linguagem de alto nível são mais fáceis de serem modificados e depurados do que programas correspondentes escritos em linguagem de baixo nível.

• **Eficiência**. A eficiência de um programa é medida pelo espaço ocupado em memória e pela rapidez com que é executado. Um programa eficiente ocupa pouco espaço e tem uma execução rápida. Diferentes seqüências de instrução em linguagem de máquina podem resultar em programas que são funcionalmente equivalentes (i.e., produzem os mesmos resultados). Algumas combinações de instrução são executadas mais rapidamente do que outras. Escrevendo um programa diretamente em linguagem de máquina (ou em *assembly*), o programador pode usualmente decidir que seqüência de instruções é a mais eficiente, enquanto que escrevendo numa linguagem de

alto nível, o programador tem pouco controle sobre como um compilador traduz o programa para linguagem de máquina. A realidade é que mesmo os mais sofisticados compiladores produzem código ineficiente. Algumas linguagens de alto nível (como *C* e *Modula-2*) possuem facilidades para acomodar código de baixo nível.

1.3.2 Linguagens de Caráter Específico x Linguagens de Caráter Geral

Linguagens de caráter específico (ou voltadas para aplicação) são linguagens desenvolvidas visando uma área de aplicação e dificilmente podem ser utilizadas para o desenvolvimento de programas em outra área. Por exemplo, a linguagem *COBOL* foi desenvolvida para a área comercial e dificilmente poderia ser utilizada para a construção de um programa científico. Em contraste, uma **linguagem de caráter geral** pode em princípio ser utilizada para escrever programas em qualquer área. A linguagem Pascal é uma linguagem de caráter geral.

1.3.3 Paradigmas de Linguagens de Programação

Um paradigma de linguagem indica a filosofia ou conceitos teóricos norteando o desenvolvimento da linguagem. Alguns dos paradigmas de linguagem mais comuns são apresentados a seguir. Uma discussão mais abrangente está além do escopo deste livro.

- **Paradigma funcional**: programação baseada em teoria matemática das funções. Exemplos: *Lisp* e *Scheme*.

- **Paradigma lógico**: programação baseada em lógica. Exemplo: *Prolog*.

- **Paradigma orientada por objetos**: Exemplos: *Smalltalk*, *Java* e *C++*.

- **Paradigma modular**: programação baseada em módulos com compilação independente. Exemplo: *Modula-2*.

Capítulo 2

INTRODUÇÃO À CONSTRUÇÃO DE ALGORITMOS

2.1 Definição de Algoritmo

O computador é uma máquina utilizada na resolução de problemas. O primeiro passo para a resolução de um problema por computador é a definição precisa do problema. Depois deste passo (que usualmente não necessita do computador), planeja-se a solução do problema por meio da escrita de um algoritmo (ainda sem a ajuda do computador). Um **algoritmo** é um procedimento computacional bem definido que recebe algum(ns) valor(es) como **entrada** (*input*) e produz algum(ns) valor(es) como **saída** (*output*). Portanto, um algoritmo é uma seqüência de passos computacionais que transformam uma dada entrada na saída desejada.

Uma analogia bastante comum que ajuda o entendimento do conceito de algoritmo é aquela entre algoritmo e receita culinária. Numa receita culinária, os ingredientes e utensílios utilizados (por exemplo, ovos, farinha de trigo) compõem a entrada e o produto final (por exemplo, um bolo) é a saída. A receita em si especifica uma seqüência de passos que nos informam como processar a entrada a fim de produzir a saída (i.e., o prato) desejada.

Raramente, um algoritmo é escrito para receber um conjunto específico de valores. Mais comumente, ele é escrito para manipular vários **casos de entrada**. Por exemplo, um algoritmo para ordenar um conjunto desordenado de números inteiros pode receber como entrada 3, 2, 7, 8 e produzir como saída 2, 3, 7, 8. Este mesmo algoritmo serviria para ordenar o conjunto 12, 1, 6, 5, produzindo 1, 5, 6, 12. Neste exemplo, o algoritmo de ordenação tem como entrada um conjunto desordenado qualquer de números inteiros; 3, 2, 7, 8 e 12, 1, 6, 5 são casos de entrada deste algoritmo neste exemplo. Entretanto, casos de entrada são comumente referidos apenas como **entrada**.

Um algoritmo é **correto**, quando para cada caso de entrada, o programa pára com a saída correta. Um algoritmo **incorreto** pode não parar quando um dado caso de entrada é introduzido, ou ele pode parar com uma saída que não é correta.

2.2 Construção de Algoritmos: Abordagem Dividir-e-conquistar

O modo mais comum de se construir algoritmos é através da abordagem **dividir-e-conquistar**. Utilizando esta abordagem, divide-se sucessivamente o problema dado em subproblemas cada vez menores até que estes possam ser resolvidos (i.e., *conquistados*) de uma forma trivial. As soluções para os subproblemas são então combinadas para resultar na solução para o problema original. Esta abordagem também é conhecida como abordagem de **refinamentos sucessivos**.

Vamos exemplificar esta abordagem através da resolução de um problema (não-computacional)[1]. Suponha que você esteja viajando de carro com alguns amigos quando um pneu fura. Obviamente, a solução para este problema é trocar o pneu furado pelo estepe (supondo que você tem um em boas condições). O algoritmo a ser seguido para resolver este problema pode ser descrito como a seguir:

```
Algoritmo TrocaDePneu:

1. Pegue o macaco e levante o carro.
2. Retire o pneu furado.
3. Pegue o estepe, coloque-o na roda e aperte os porcas.
4. Rebaixe o carro.
5. Guarde o pneu furado e o macaco.
```

Note que o problema inicial de troca do pneu foi substituído por cinco subproblemas. Cada um destes subproblemas é razoavelmente independente um do outro. Portanto, você poderia atribuir o subproblema 1 para um de seus amigos, o subproblema 2 para outro, o subproblema 3 para você mesmo, etc. A analogia correspondente a este último fato em programação é que você pode (e usualmente deve) atribuir cada uma destas subtarefas a partes relativamente independentes do programa (estas partes são, usualmente, denominadas de **subrotinas** ou **procedimentos**, e serão discutidas mais tarde).

Agora, suponha que o amigo a que foi atribuída a tarefa 1 não sabe como executá-la automaticamente. Então, você terá que especificá-la melhor a fim de que ele possa executá-la. Em outras palavras, você terá que **refinar** a tarefa 1 em termos de subtarefas que seu amigo saiba realizar. A analogia correspondente em programação a este refinamento de tarefas é que o programador deve refinar (i.e., dividir) cada passo de um algoritmo de modo que cada uma das divisões correspondentes possa ser representada (idealmente) por uma única instrução numa linguagem de alto nível. Voltando ao nosso

[1] Baseado em exemplo apresentado em Koffman, Elliot B., 1986. *Turbo Pascal: A Problem Solving Approach*. Addison-Wesley Publishing.

problema inicial, o Passo 1 do algoritmo acima pode ser refinado para resultar na seguinte seqüência de subpassos:

```
1.1 Remova o macaco do porta-malas.
1.2 Coloque o macaco sob o carro e próximo ao pneu furado.
1.3 Insira a manivela no macaco (ou monte-o conforme o
    modelo de macaco).
1.4 Coloque um calço sob o carro para impedi-lo de se
    mover.
1.5 Levante o carro com o macaco até que haja espaço
    suficiente para colocar o pneu de estepe.
```

Os subpassos 1.4 e 1.5 podem ser ainda mais refinados (na realidade, todos os outros, principalmente o subpasso 1.3, também poderiam ser refinados). A colocação do calço sob o carro depende do fato de o carro estar apontando para baixo ou para cima de uma ladeira, ou ainda numa superfície plana (sendo, portanto, o calço desnecessário neste último caso). Então, o Passo 1.4 poderia ser refinado para:

1.4.1 **Se** o carro está apontando para o topo de uma ladeira **então** coloque o calço atrás de um pneu em bom estado; **senão**, **se** o carro está apontando para a base de uma ladeira **então** coloque o calço na frente de um pneu em bom estado.

Note que o subpasso 1.4.1 representa duas ações condicionadas às condições iniciais do carro (i.e., às condições de entrada do problema); apenas uma dessas ações será executada. No subpasso 1.4.1, as condições são os fatos seguindo as palavras "**se**" e as ações vêem após as palavras "**então**". Ações **condicionais** (ou de tomada de decisões) são muito comuns em programação e as linguagem de alto níveis mais comuns provêem facilidades para codificação de ações condicionais.

Voltando ao nosso algoritmo, o subpasso 1.5 envolve uma ação repetitiva que precisa ser refinada: levantar o carro até que haja espaço suficiente para colocar o pneu de estepe. Isto é feito no Passo 1.5.1 abaixo:

Enquanto não há espaço suficiente para colocar o pneu de estepe, **faça** o seguinte: levante o macaco.

ou alternativamente:

Repita o levantamento do macaco **até que** haja espaço suficiente para colocar o pneu de estepe.

Qualquer das formas alternativas do subpasso 1.5.1 possui um análogo em programação. A primeira alternativa do subpasso 1.5.1 representa uma estrutura de repetição do tipo: "**enquanto** uma dada condição não é satisfeita **faça** (ou execute repetidamente) uma determinada ação". No exemplo dado, a condição de parada é haver espaço suficiente para colocar o pneu de estepe e a ação a ser repetida é levantar o macaco.

2.3 Uma Linguagem Algorítmica

Um algoritmo pode ser especificado de várias formas, como em Português ou numa linguagem de programação. O único requerimento é que a linguagem permita uma descrição precisa do procedimento computacional a ser seguido. Uma linguagem para descrição de algoritmos em linguagem natural e que usa certas convenções próximas de uma linguagem de programação é chamada de **linguagem algorítmica** e os algoritmos escritos nesta linguagem são ditos estarem em **pseudocódigo**. As convenções da linguagem algorítmica a ser utilizado neste livro serão descritas a seguir.

2.3.1 Atribuição, Variáveis e Expressões

Uma **variável** em programação representa, através de símbolos, o conteúdo (simbólico) de uma posição (célula) de memória. Assim, quando se diz que uma variável x assume um valor 5, se quer na realidade dizer que existe uma posição de memória, representada simbolicamente por `x`, que contém o valor `5`.

Informalmente, uma **expressão** é uma seqüência de símbolos que resultam num valor (numérico ou não), incluindo um valor constante (por exemplo, 10). Uma instrução de **atribuição** representa o ato de uma variável receber o valor de outra variável ou de uma expressão (que pode incluir a própria variável). Uma atribuição em pseudocódigo será indicada pelo símbolo "←", com a seta iniciando na variável ou expressão sendo atribuída (na direita) e terminando na variável que recebe a atribuição (na esquerda). Por exemplo, se quisermos expressar a idéia de que uma variável `x` recebe o valor `y + 2`, onde `y` é uma outra variável podemos escrever em pseudocódigo:

```
x ← y + 2
```

A instrução de atribuição acima lê-se como: "x recebe o valor de y mais dois".

Poderíamos também querer expressar o fato de x ter seu último valor acrescido de 2 através de:

$$x \leftarrow x + 2$$

Este último exemplo pode ser interpretado como: “substitua o valor atual de x por este valor acrescido de dois”. Note que uma atribuição não tem nada a ver com igualdade em matemática (embora algumas linguagem de programação usem o símbolo de igualdade para representar atribuição).

Existem três tipos básicos de expressões em programação:

1. **expressões aritméticas** - aquelas que utilizam operações aritméticas (por exemplo, soma) e resultam em valores numéricos;

2. **expressões relacionais** - aquelas que utilizam operadores relacionais de comparação (por exemplo, “>”) e que resultam num valor verdadeiro ou falso; e

3. **expressões lógicas** - aquelas que utilizam conectivos lógicos (por exemplo, **ou** lógico) e resultam num valor verdadeiro ou falso.

Expressões mais gerais podem conter os três tipos básicos de expressões acima como subexpressões. Expressões serão estudadas em maior profundidade mais adiante.

2.3.2 Entrada e Saída

Freqüentemente, um programa de computador necessita obter seus dados de entrada através de algum meio externo (por exemplo, teclado, disco). Em linguagem algorítmica, representa-se este fato através da instrução **leia**, seguida por variáveis que serão associadas aos dados que devem ser lidos. Assim, uma instrução como:

leia(x_1, x_2, ..., x_n)

nos informa que n dados estão sendo lidos e então associados às variáveis x_1, x_2, ..., x_n.

De modo análogo, tem-se uma instrução para representar os dados de saída de um algoritmo no meio de saída (por exemplo, tela, impressora):

escreva(x_1, x_2, ..., x_n)

Esta última expressão significa que os dados armazenados nas variáveis x_1, x_2, ..., x_n serão impressos (ou gravados) num meio de saída. Uma instrução **escreva** pode também conter parâmetros do tipo **cadeias de caracteres** (por exemplo, "Resultado do programa:") em vez de variáveis. Cadeias de caracteres são representadas aqui como uma seqüência de caracteres quaisquer confinada entre aspas (" ").

2.3.3 Estruturas de Controle

Estruturas de controle permitem que o programador controle a seqüência e a freqüência de execução de segmentos de um programa. Algumas estruturas de controle (condicional e repetição) já foram vistas informalmente no exemplo de algoritmo apresentado anteriormente.

• **Estrutura Condicional**:

se <condição> **então** <ação 1> **senão** <ação 2>

Exemplo de uso de estrutura condicional:

```
leia(x);
se (x < 0) então
    escreva("O número é negativo")
senão
    escreva("O número não é negativo")
```

A instrução condicional descrita acima funciona da seguinte forma. Quando a condição é satisfeita (i.e., quando é verdadeira), a ação correspondente ao **então** (ação 1) é executada; caso contrário, a ação correspondente ao **senão** (ação 2) é executada. Por exemplo, se o valor lido para x fosse -5, a ação a ser executada seria: `escreva("O número é negativo")`. Quando não existe nenhuma ação a ser executada no caso em que a condição não é satisfeita, pode-se omitir a parte **senão** da estrutura.

• **Estrutura de Repetição 1:**

repita <ação> **até que** <condição>

Exemplo:

```
leia(x);
repita
      x ← x + 1
até que (x > 10);
```

• **Estrutura de Repetição 2:**

enquanto <condição> **faça** <ação>

Exemplo:

```
leia(x);
enquanto (x <10) faça
      x ← x + 1;
```

As estruturas de repetição 1 e 2 já foram vistas informalmente antes. Na prática, estas estruturas são utilizadas quando não se conhece a priori quantas vezes a ação correspondente terá que ser executada. Por exemplo, o número de vezes que a ação (`x ← x + 1`) do exemplo anterior será executada dependerá do valor inicial de `x` lido. Por exemplo, se o valor inicial de `x` for igual a `1`, a ação será executada 10 vezes (confira isso); se o valor inicial de `x` for `12`, a ação não será executada nenhuma vez. (O que aconteceria no caso do exemplo da instrução **repita** se o valor inicial de `x` fosse igual a `12`?)

• **Estrutura de Repetição 3:**

para <valor inicial>; <incremento>; **até** <valor final> **faça** <ação>

Exemplo:

```
x ← 1;
para i ← 1, incremento 1, até 10 faça
      x ← x + 1;
```

Ao contrário das estruturas de repetição 1 e 2, a estrutura de repetição 3 deve ser utilizada quando se sabe de antemão quantas vezes a ação correspondente deverá ser executada. Para utilizá-la, é necessária a introdução de uma variável de controle (`i`, no exemplo acima), juntamente com seus valores inicial e final, e de quanto esta variável será acrescida (i.e., **incrementada**) ou decrescida (i.e., **decrementada**) no final da execução de cada ação. Os valores inicial, final e do incremento podem ser negativos ou positivos, mas devem sempre ser inteiros. Quando o valor do incremento é 1, este pode ser omitido da estrutura.

Quando se usa qualquer estrutura de repetição, deve-se garantir que a condição de parada da mesma será satisfeita em algum ponto de execução do algoritmo. Por exemplo, se o incremento da variável de controle (`i`) no último exemplo fosse negativo, a condição de parada (`i = 10`) nunca seria atingido (convença-se de que entende isto). Ter-se-ía, então, neste caso, um laço infinito e o algoritmo jamais pararia.

2.3.4 Estrutura do Algoritmo

A estrutura das várias partes que compõem o algoritmo será indicada por meio de **endentação**. Um **bloco** (i.e., uma seqüência de instruções) com um nível de endentação significa que o bloco pertence à (ou está subordinado à) instrução que imediatamente *envolve* o bloco endentado. Por exemplo, no conjunto de instruções a seguir,

```
x ← 1;
enquanto (x <10) faça
      x ← x + 1;
      escreva(x);
escreva("Bye, bye");
```

as instruções **`x ← x + 1`** e **`escreva(x)`** estão num mesmo nível de endentação e portanto fazem parte de um mesmo bloco de instruções (pertencente à instrução "**`enquanto (x <10) faça`**"). A instrução **`escreva`**`("Bye, bye")`, que não está endentada, não pertence a este mesmo bloco.

A maioria das linguagens de alto nível ignoram completamente espaços adicionais em branco, como aqueles usados para fazer endentação. Portanto, outros mecanismos são usados em programação para indicar os limites de um bloco. Em Pascal, por exemplo, um bloco é delimitado pelas palavras **begin** e **end**. Na linguagem algorítmica usada aqui, que será lida apenas por seres humanos, entretanto, a endentação é suficiente para fazer esta delimitação. Em resumo, lembre-se que endentar programas e algoritmos é apenas uma forma de melhorar a legibilidade dos mesmos. Endentação não tem o menor significado para o computador.

2.4 Resolvendo Problemas Usando o Computador

Programação é a arte de se instruir o computador a resolver problemas. É bastante tentador (mesmo para programadores mais experientes) correr para o computador e começar a escrever um programa tão logo se tenha uma idéia daquilo que seria a solução para o problema em mãos. Também, é comum gastar-se mais tempo depurando (i.e., retirando erros de) um programa do que escrevendo o mesmo. Parece humanamente impossível ter-se um programa (não-trivial) que não contenha erros (*bugs*), mas um programa escrito *às pressas* é muito mais susceptível a erros do que um programa que implementa a solução de um problema profundamente estudado previamente. Portanto, o conselho aqui é simples: resista à tentação de correr para o computador e escrever o programa tão logo você tenha uma vaga idéia de como escrever o programa. Em vez disso, tente seguir o seguinte procedimento:

1. Leia e reflita cuidadosamente sobre o problema. Tente responder às seguintes questões:

1.1 Que dados iniciais do problema estão disponíveis (entrada)? Escreva a resposta a esta questão precedida da palavra ***entrada****.*

1.2 Qual é o resultado esperado (saída)? Escreva a resposta precedendo-a por ***saída****.*

1.3 Que tipo de processamento (algoritmo) é necessário para obter o resultado esperado a partir dos dados de entrada? Escreva um algoritmo que faça isso. Não se preocupe, por enquanto, com que o algoritmo seja bem detalhado.

As respostas às questões 1.1 e 1.2 são necessárias para se identificar os dados de entrada e saída do programa. Certifique-se de estas respostas estão bem claras antes de prosseguir.

2. Refine (i.e., subdivida) cada passo do algoritmo esboçado em 1.3 que não tenha solução imediata (i.e., trivial).

3. Verifique se o algoritmo realmente funciona conforme o esperado. Em outras palavras, verifique se o algoritmo faz o que ele deveria fazer.

3.1 Teste o algoritmo com alguns casos de entrada que sejam ***qualitativamente*** *diferentes e verifique se ele produz a saída desejada para cada um destes casos. Se o algoritmo produz respostas indesejável, volte ao Passo 1.*

Suponha, por exemplo, que seu algoritmo se propõe a resolver equações de segundo grau. Se você usar casos de entrada (i.e., equações de segundo grau) que resultem sempre em $\Delta > 0$ (lembram-se disso de Matemática do primeiro grau?), estes casos de entrada (não importa a quantidade) não são qualitativamente diferentes, e portanto, não são suficientes para testar seu algoritmo completamente (por exemplo, talvez o algoritmo não funcione quando os dados resultam em $\Delta = 0$). Para testarmos efetivamente este algoritmo, deveríamos também testar casos de entrada que resultem em $\Delta = 0$ e $\Delta < 0$. Resumindo, é a qualidade (e não a quantidade) dos casos de entrada que conta para testar um algoritmo.

Note ainda que no caso em que você tenha que voltar ao Passo 1, como recomendado acima, você não precisa desfazer tudo que já fez até aqui. Talvez você tenha apenas esquecido de levar em consideração um dado de entrada (Passo 1.2), ou um dos passos de seu algoritmo (Passo 1.3) ou seu refinamento (Passo 2) seja inadequado.

4. Escreva o programa no computador, utilizando as convenções (sintaxe) da linguagem de programação escolhida. Isto resulta no programa-fonte.

4.1 Traduza cada instrução no algoritmo numa instrução correspondente na linguagem de programação escolhida.

Esta última etapa resulta num programa-fonte que deverá ser compilado antes de ser executado. Um último conselho: nunca menospreze o Passo 3 (teste do algoritmo) do procedimento acima: um teste bem executado pode lhe economizar muita dor-de-cabeça na depuração de um programa.

2.5 Operadores e Expressões Aritméticas

Operadores aritméticos são utilizados em algoritmos (e em programas) na construção de expressões aritméticas. Os operadores aritméticos usados em programação correspondem às operações usuais em matemática (por exemplo, soma, subtração, multiplicação, etc.). Entretanto, nem sempre eles utilizam a mesma notação vista em matemática. Por exemplo, em matemática, o operador de multiplicação pode ser representado por um ponto (por exemplo, `a.b`) ou pela simples justaposição de operandos (por exemplo, `ab`), enquanto que em programação, este operador é usualmente representado por "*" (asterisco).

Os operadores aritméticos mais comuns em programação são os seguintes:

Operador	Significado
`-`	menos unário (i.e., inversão de sinal)
`+`	soma
`-`	subtração
`*`	multiplicação
`/`	divisão (por exemplo, `5/2 = 2.5`)
`div`	divisão inteira (por exemplo, `5` **`div`** `2 = 2`)
`mod`	resto da divisão inteira (por exemplo, `5` **`mod`** `2 = 1`)

Existem ainda **funções predefinidas** que podem ser utilizadas para compor expressões aritméticas. Por exemplo, a função `SQRT` resulta na raiz quadrada de um número que a mesma recebe como entrada (por exemplo, `SQRT(4) = 2`). Linguagens de programação usualmente provêem o usuário com uma vasta coleção de tais funções que incluem, por exemplo, funções trigonométricas, exponenciais, etc. Estas funções serão vistas à medida em que se fizerem necessárias.

Expressões aritméticas de complexidades arbitrárias podem ser construídas utilizando-se operadores e funções aritméticos. Por causa disso, deve-se estabelecer uma convenção sobre a ordem em que as operações, representadas por operadores e funções, são executadas. Por exemplo, considere a simples expressão aritmética "`2 + 3*4`". Sabemos que esta expressão resulta em `14` (e não em `20`) porque nos foi ensinado desde os primeiros anos de estudo que deve-se efetuar primeiro a multiplicação antes de efetuar a adição. Em outras palavras, o operador "*" tem precedência sobre o operador "+". Em geral, tem-se o seguinte quadro de precedência entre os operadores usuais:

Operador	Precedência
- (unário)	**Alta** (aplicado primeiro)
*, /, **div**, **mod**	↓
+, - (binário)	**Baixa** (aplicado por último)

Na tabela de precedência acima, os operadores em uma mesma linha têm a mesma precedência. Portanto, quando tais operadores são encontrados juntos em uma mesma expressão aritmética, o operador mais à esquerda é aplicado primeiro (isto é chamado de regra de **associatividade à esquerda**). Note também que, apesar de compartilharem o mesmo símbolo, o menos unário (i.e., o sinal de menos) tem maior precedência do que o menos binário (i.e., o símbolo de subtração). Finalmente, observe que, como em aritmética usual, o uso de parênteses altera a precedência dos operadores. Por exemplo, a expressão "`(2 + 3)*4`" resulta em `20` (e não em `14` como antes) porque os parênteses fazem com que, agora, a operação de soma seja aplicada antes da multiplicação. O uso de parênteses é indicado quando se quer ter certeza que uma operação será efetuada antes de outra, mesmo que estes sejam desnecessários (voltaremos a este ponto novamente logo mais).

2.6 Operadores Relacionais, Expressões e Variáveis Booleanas

Operadores (aritméticos) **relacionais** são similares aos operadores aritméticos vistos antes no sentido de que eles operam sobre números (ou, mais geralmente, sobre expressões aritméticas que resultam em números). No entanto, estes operadores **não resultam** em números como os operadores aritméticos. Em vez disso, estes operadores fazem comparações entre números (ou expressões ou variáveis que resultem em números) e resultam num valor que pode ser **verdadeiro** ou **falso**, dependendo da situação. Por exemplo, o valor retornado pelo operador "`>`" na expressão "`2 > 3`" é falso; enquanto que o valor retornado em "`3 > 2`" é verdadeiro.

Os operadores relacionais mais comuns correspondem exatamente aos operadores de igualdade (=) e desigualdade (≠, >, <) encontrados em Matemática. Note ainda que uma expressão relacional corresponde a uma questão cuja resposta é *sim* ou *não*. Quando a resposta à esta pergunta é *sim*, o resultado da expressão é *verdadeiro*; quando a resposta à esta pergunta é *não*, o resultado da expressão é *falso*. Por exemplo, "`2 > 3`" corresponde à questão: "dois é maior do que três?"; a resposta a esta questão é obviamente "*não*" e, portanto, a expressão "`2 > 3`" resulta em falso. A lista dos

operadores mais usais em programação, bem como seus significados, é apresentada a seguir[2].

A = B - verdadeiro quando A é **igual a** B
A ≠ B - verdadeiro quando A é **diferente de** B
A > B - verdadeiro quando A é **maior do que** B
A ≥ B - verdadeiro quando A é **maior do que ou igual a** B
A < B - verdadeiro quando A é **menor do que** B
A ≤ B - verdadeiro quando A é **menor do que ou igual a** B

Conforme foi mencionado anteriormente, os operadores relacionais podem ser utilizados para comparar expressões aritméticas. Em outras palavras, pode-se ter expressões mais gerais contendo operadores aritméticos e relacionais. Por exemplo, "`2 + 3 > 4 + 6`" é uma expressão contendo operadores aritméticos (+) e relacionais (>) e tem o significado: "dois mais três é maior do que quatro mais seis?". Note que, para esta expressão ter este significado, está implícito que as operações de soma devem ser efetuadas antes da operação relacional "maior do que". Isto é, o operador de soma tem maior **precedência** do que o operador maior do que. Em geral, os operadores relacionais têm prioridade menor do que qualquer operador aritmético. Isto significa que numa dada expressão contendo operadores aritméticos e relacionais, os operadores aritméticos são aplicados em primeiro lugar.

Expressões contendo operadores relacionais constituem um exemplo de **expressões lógicas**. Expressões lógicas (também conhecidas como **expressões booleanas** em homenagem ao matemático inglês **George Boole** - pronuncie "buol") são simplesmente expressões que resultam em um valor verdadeiro ou falso. **Variáveis lógicas** (ou **variáveis booleanas**) são variáveis que podem assumir apenas os valores verdadeiro ou falso. A uma variável booleana pode-se atribuir diretamente um valor verdadeiro ou falso, ou pode-se ainda atribuir o valor retornado por uma expressão booleana. Por exemplo, as atribuições abaixo são perfeitamente válidas para uma variável booleana.

```
booleana  bol1, bol2;

bol1 ← falso;
bol2 ← 2 > 3;
```

[2] Os operandos A e B podem ser números, variáveis ou expressões aritméticas.

Variáveis e expressões booleanas podem também ser combinadas entre si por meio de **operadores lógicos** (não confunda com operadores relacionais!). Existem três operadores lógicos usados em programação: a **negação** (representada por **não** ou **not**), a **conjunção** (representada por **e** ou **and**), e a **disjunção** (representada por **ou** ou **or**). Os possíveis resultados das aplicações destes operadores podem ser assimilados por meio de **tabelas-verdade**. As tabelas-verdade para os operadores **não**, **e**, e **ou** são apresentados a seguir. Note que, nestas tabelas, `operando1` e `operando2` podem corresponder a qualquer variável ou expressão booleana.

Operando1	**não Operando1**
verdadeiro	falso
falso	verdadeiro

Operando1	**Operando2**	**Operando1 e Operando2**
verdadeiro	verdadeiro	verdadeiro
verdadeiro	falso	falso
falso	verdadeiro	falso
falso	falso	falso

Operando1	**Operando2**	**Operando1 ou Operando2**
verdadeiro	verdadeiro	verdadeiro
verdadeiro	falso	verdadeiro
falso	verdadeiro	verdadeiro
falso	falso	falso

As seguintes observações podem ser feitas a partir das tabelas acima. Primeiro, a negação de um operando é verdadeira quando o operando é falso e vice-versa. Segundo, ao invés de decorar toda a tabela-verdade do operador **e**, é necessário apenas lembrar que `operando1` **`e`** `operando2` é verdadeiro apenas quando cada um dos operandos é verdadeiro; em qualquer outra situação a aplicação do **e** resulta em falso. Finalmente, ao invés de decorar toda a tabela do **ou**, é necessário apenas lembrar que sua aplicação resulta em falso apenas quando os dois operandos são falsos.

Exemplos:

```
booleana bol1, bol2, bol3, bol4, bol5, bol6;

bol1 ← 2 = 5; {bol1 assume o valor falso}
bol2 ← falso; {óbvio}
bol3 ← 10 ≤ 2*5; {bol3 assume o valor verdadeiro}
bol4 ← não bol1; {bol4 assume o valor verdadeiro porque o
                  valor de bol1 é falso}
bol5 ← bol2 e bol3; {bol5 assume o valor falso porque o valor
                     de bol2 é falso (apesar de o valor de
                     bol3 ser verdadeiro)}
bol6 ← bol2 ou bol3; {bol6 assume o valor verdadeiro porque
                      o valor de bol3 é verdadeiro}
```

Operandos e operadores lógicos podem ser agrupados para formar expressões mais complexas. Para isso, é necessário que se defina uma precedência de operações lógicas similar àquela precedência de operadores aritméticos com a qual estamos acostumados. Esta precedência é que **não** tem a maior precedência, **e** tem a seguinte maior precedência, e **ou** tem a menor precedência. Isto implica em que, por exemplo, `não A ou B e C` representa `(não A) ou (B e C)` [ao invés de, por exemplo, `não ((A ou B) e C))`].

Para concluir, note que todos os operadores (aritméticos, relacionais e lógicos) vistos até aqui podem ser colocados juntos para formarem expressões de complexidades arbitrárias. Por exemplo, `(2 + 4 > 7) e não (x = 10)` representa uma expressão perfeitamente legal. Até aqui, falou-se em precedência de operações considerando-se apenas as possíveis operações dentro de cada um desses grupos de operadores. Ora, como todos estes operadores podem aparecer juntos numa mesma expressão, é necessário que se examinem as precedências relativas a todas estas operações. Esta precedência geral de operadores é apresentada na tabela a seguir.

Operador	Precedência
- (unário), **não**	**Alta** (aplicado primeiro)
*, /, **div**, **mod**, **e**	↓
+, - (binário), **ou**	↓
operadores relacionais (=, ≠, ≥, etc.)	**Baixa** (aplicado por último)

Como visto anteriormente, pode-se modificar a prioridade de operações por meio do uso de parênteses. Algumas vezes o uso de parênteses é obrigatório a fim de que uma expressão faça algum sentido. Por exemplo, a expressão:

```
x > y e y > z
```

seria interpretada como `x > (y e y) > z`, e portanto, completamente sem sentido (convença-se disso). Assim, a forma correta de escrever esta expressão seria:

```
(x > y) e (y > z)
```

Outras vezes, o uso de parêntese é obrigatório para dar à expressão o significado que você pretende que ela tenha. Por exemplo, se você pretende que a expressão:

```
bol1 ou bol2 e bol3 ou bol4
```

tenha o significado:

```
(bol1 ou bol2) e (bol3 ou bol4 )
```

você tem que escrevê-la exatamente como acima (i.e., com os parênteses); caso contrário, a primeira expressão seria interpretada como:

```
bol1 ou (bol2 e bol3) ou bol4
```

visto que o **e** tem maior precedência do que o **ou**. Um conselho: tome muito cuidado com expressões lógicas pois elas representam uma fonte de erro muito comum em programação (mesmo para programadores experientes!).

O uso de parênteses também é recomendado nos seguintes casos:

- Quando se tem dúvida com relação a precedência entre dois ou mais operadores.
- Para tornar uma expressão complexa mais legível.

Expressões envolvendo operadores relacionais e lógicos são particularmente vulneráveis e susceptíveis a erros em programação. O uso judicioso de parênteses para melhorar a legibilidade destas expressões é uma arma preventiva contra erros. Como um conselho final, lembre-se que o uso excessivo de parênteses não prejudica o entendimento de uma expressão, mas a falta de parênteses pode resultar numa interpretação que não é a pretendida. Em outras palavras, é melhor ser redundante e ter um programa funcionando do que tentar ser sucinto e ter um programa com um comportamento errático.

2.7 Comentários e Legibilidade de Algoritmos

Legibilidade é uma propriedade altamente desejável de um algoritmo. Um algoritmo legível é mais fácil de entender, testar e refinar do que um algoritmo ilegível.

Os seguintes procedimentos são recomendados na construção de algoritmos legíveis:

> • Incorpore **comentários** (em Português) no algoritmo. Estes podem ser incorporados entre chaves e devem descrever pelo menos as variáveis utilizadas. O objetivo de se escrever comentários num algoritmo (i.e., **comentar o algoritmo**) é o de torná-lo mais legível. O melhor momento de se comentar um algoritmo é durante sua concepção (e não um mês depois, por exemplo), i.e., quando seus detalhes ainda estão *frescos*. Comentários devem acrescentar alguma coisa além daquilo que pode ser facilmente apreendido. Por exemplo, na instrução:
>
> ```
> x ← 2        {x recebe o valor 2}
> ```
>
> o comentário é completamente redundante e irrelevante.

Comentários no início do algoritmo também são extremamente úteis. Estes comentários devem conter pelo menos a seguinte informação: (1) uma descrição daquilo que o algoritmo faz, (2) como deve utilizado, (3) os significados das variáveis mais importantes, (4) descrição de algum método especial utilizado para implementar o algoritmo ou parte dele, (5) descrição dos tipos de dados utilizados, (6) o nome do autor do algoritmo e (7) a data em que foi escrito. Por exemplo, o seguinte poderia ser a descrição de um algoritmo para ordenar uma lista de nomes representados como cadeias de caracteres (não se preocupe em entender os detalhes desta descrição; note apenas como cada um dos itens acima poderia ser descrito). Declarações entre colchetes e em itálico não são parte do comentário em si: estes representam explicações sobre cada parte do comentário.

```
{Este algoritmo ordena uma lista de nomes e deve ser
utilizado quando uma lista alfabética de nomes for
necessária [o quê o algoritmo faz (1)]. O algoritmo
recebe como entrada um ponteiro para uma lista
desordenada e retorna como saída um ponteiro para esta
mesma lista ordenada [como o algoritmo deve ser
utilizado (2)]. Cada nome é representado como uma cadeia
de caracteres e a lista de nomes é uma estrutura
dinâmica encadeada [descrição dos tipos de dados
utilizados (5)]. A variável "nome" representa cada nome
na lista [descrição de uma das variáveis utilizadas;
seguir-se-íam descrições de outras variáveis se fosse o
caso (3)]. O algoritmo utiliza o método quicksort de
ordenação [método utilizado para implementar o algoritmo
(4)].  Autor: João da Silva. Data: 21/12/95. }
```

Na prática, este algoritmo poderia ser modificado pelo mesmo ou por outro programador. Esta informação deve também constar do comentário inicial do algoritmo.

• Utilize **nomes de variáveis** que sejam **significativos**. Em outras palavras, o nome de uma variável deve lembrar o tipo de informação armazenada na variável. Por exemplo, num algoritmo para calcular as médias de vários alunos em várias disciplinas, os nomes de variáveis: `nome`, `matrícula`, `disciplina` e `média` serão certamente muito mais significativos do que simplesmente `x`, `y`, `w` e `z`.

• **Grife todas as palavras-chave** (por exemplo, leia, se, então) utilizadas no algoritmo[3].

• Use **endentação coerentemente**. Não existe nenhuma fórmula determinada para se fazer endentação (i.e., alinhar comandos), mas, uma vez que você tenha escolhido uma forma de fazer endentação, utilize-a sempre de maneira consistente em seus algoritmos. Por exemplo, algumas pessoas preferem alinhar comandos relacionados a se ... então ... senão como:

```
se <condição> então
     <instruções>
senão
     <instruções>
```

enquanto outros podem preferir:

[3] Palavras-chave são escritas muitas vezes em **negrito** em livros e outros impressos, mas este recurso não está disponível, ou pelo menos não é muito prático, em algoritmos manuscritos.

```
se <condição>
     então
          <instruções>
     senão
          <instruções>
```

Nenhuma das duas formas de endentação acima é *mais correta* ou *mais legível* do que a outra. Isto é, usar uma ou a outra é uma questão de gosto pessoal, mas, uma vez que uma forma é escolhida, ela deve ser consistentemente mantida na escrita de um algoritmo.

• Além de usar endentação, o uso judicioso de **espaços em branco** também pode melhorar a legibilidade dos algoritmos (da mesma forma que a divisão em parágrafos melhora a legibilidade de textos comuns). Em especial, use espaços em branco para separar declarações e o corpo do algoritmo. Separe também grupos de instruções que têm funções diferentes. Como em endentação, o uso de espaços em branco para melhorar a legibilidade de um algoritmo não possui regras fixas: simplesmente use o bom senso!

• Use **parênteses** em expressões para esclarecer dúvidas e torná-las mais legíveis (ver *Seções 2.5* e *2.6*). Não se preocupe em ser redundante. Lembre-se que para cada parêntese aberto deve haver um parêntese de fecho. Isto significa que o número de parênteses de abertura [“(”] deve ser igual ao número de parênteses de fechamento [“)”]. Uma forma de se checar as correspondências entre parênteses numa expressão contendo muitos parênteses é traçando-se linhas conectando cada par de parênteses, como no exemplo abaixo (obviamente, estas linhas não devem fazer parte do algoritmo!). Esta forma de se checar expressões parentéticas é muito mais eficiente do que simplesmente contar o número de parênteses abertos e comparar para ver se este casa com o número de parênteses fechados. O uso de espaços entre parênteses adjacentes (como o primeiro e o segundo pares de parênteses no exemplo abaixo) também ajuda na legibilidade de uma expressão parentética.

```
( (3 + 4) / (5*2) ) * (4 - 7*(3 + 8) )
```

• Finalmente, **nunca escreva mais de uma instrução por linha**. Isto é, mesmo que você tenha um conjunto de instruções suficientemente pequeno para caber numa linha, não o faça porque isso prejudicará a legibilidade do algoritmo. Também, quando uma instrução for muito extensa para caber numa única linha, use endentação para indicar a continuação de uma instrução na próxima linha.

2.8 Exercícios de Revisão

1. Considere o seguinte algoritmo, onde i1, i2, i3, i4 e i5 representam instruções:

```
booleana: b1, b2, b3;

      se b1 então i1
            senão
               se b2 então
                        se b3 então
                              i2;
                           senão
                              i3;
                              i4;
      i5;
```

a) Que instruções serão executadas quando `b1 = V`, `b2 = V` e `b3 = F`?
b) Que instruções serão executadas quando `b1 = F`, `b2 = V` e `b3 = F`?
c) Que instruções serão executadas quando `b1 = F`, `b2 = V` e `b3 = V`?
d) Que valores devem assumir `b1`, `b2` e `b3` para que apenas a instrução `i5` seja executada?

2. Qual será o valor da variável `resultado` após a execução do algoritmo a seguir?

```
booleana: b1, b2, b3;
      real: x, y;
      inteiro: resultado;

      b1 ← falso;
      b2 ←  verdadeiro;
      b3 ←  falso;

      x ←  1.5;
      y ←  3.5;
      x ←  x + 1;

      se b3 ou ( (x + y > 5) ou (não b1 e b2) )
             então   resultado ←  0;
             senão   resultado ←  1;
```

3. Escreva a tabela-verdade correspondente à expressão booleana: A ou B e C, onde A, B e C são variáveis lógicas.

4. Se A = 150, B = 21, C = 6, L1 = falso e L2 = verdadeiro, qual será o valor produzido por cada uma das seguintes expressões booleanas?
a) não L1 e L2
b) não L1 ou L2
c) não (L1 e L2)
d) L1 ou não L2
e) (A > B) e L1
f) (L1 ou L2) e (A < B + C)
g) Por que a expressão: L1 ou A < B não faz sentido?

5. Duas instruções são **funcionalmente equivalentes** se elas produzem o mesmo efeito em quaisquer circunstâncias. Verifique se as instruções 1 e 2 abaixo são funcionalmente equivalentes:

Instrução 1:
```
L ←  X = Y
```

Instrução 2:
```
se X = Y
      então L ←  verdadeiro;
      senão L ←  falso;
```

6. Uma ***seqüência de Fibonacci*** é constituída por uma seqüência de números naturais, cujos dois primeiros termos são iguais a 1, e tal que cada número (exceto os dois primeiros) na seqüência é igual a soma de seus dois mais próximos antecedentes. Isto é, a seqüência de Fibonacci é constituída da seguinte forma:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Escreva um algoritmo que gera a seqüência de Fibonacci até o n-ésimo termo. (Uma solução para este problema é apresentada no final, mas tente resolvê-lo antes de examinar a solução.)

7. Escreva um algoritmo que recebe o raio de um círculo como entrada e calcula sua área. Dado: $A = \pi r^2$.

8. Escreva um algoritmo que transforme polegadas em centímetros. Dado: 1pol = 2.54cm.

9. Escreva um algoritmo que troca os valores de duas variáveis. Em outras palavras, se as variáveis dadas são x e y, no final x terá o valor inicial de y, e y terá o valor inicial de x.

10. Escreva um algoritmo que resolve equações do segundo grau. O algoritmo deve ler os valores dos coeficientes a, b e c que caracterizam a equação, e então calcular e imprimir as raízes. (Lembre-se que algumas equações do segundo grau não possuem raízes reais!)

11. Escreva um algoritmo que lê três dados nas variáveis x, y e z, e calcule e imprima a soma e o produto destas variáveis.

12. Escreva um algoritmo que calcule e imprima uma tabela apresentando as primeiras 15 potências de 2 (i.e., 2^n).

13. Escreva um algoritmo que recebe dois números como entrada e imprime o menor deles. Se os números forem iguais, não haverá diferença em qual deles será impresso.

14. Escreva um algoritmo ligeiramente diferente do anterior que imprima uma mensagem (por exemplo, “Os números são iguais”) quando os números forem iguais.

15. Escreva um algoritmo que calcule a soma de um conjunto de valores. O algoritmo deve terminar quando um valor zero for lido.

16. Escreva um algoritmo que calcule e imprima o fatorial de um dado número inteiro *não-negativo*. O fatorial de um número inteiro $n \geq 0$ é dado por:

$$\begin{cases} 1, & \text{se } n = 0 \\ n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1, & \text{se } n > 0 \end{cases}$$

Solução para o Exercício 6 sobre a seqüência de Fibonacci

Entrada: número de termos da seqüência.
Saída: os termos da seqüência.

```
Algoritmo Fibonacci:

1. Leia o número de termos da seqüência
2. Gere e imprima os termos da seqüência
  2.1 Imprima os dois primeiros termos da seqüência
  2.2 Gere e imprima os termos da seqüência a partir do terceiro
      termo até o n-ésimo.
      2.2.1 Enquanto o n-ésimo termo não for gerado e impresso faça o
            seguinte:
            2.2.1.2 Calcule o termo atual como sendo a soma dos dois
                    termos antecedentes
            2.2.1.2 Imprima o termo atual
            2.2.1.3 Atualize os termos antecedentes:
                    2.2.1.3.1 O primeiro antecedente passa a ser o
                              segundo.
                    2.2.1.3.1 O segundo antecedente passa a ser o
                              atual.
```

```
Algoritmo Fibonacci: {Gera e imprime os primeiros n elementos da
                      seqüência de Fibonacci}

      inteiro:  antecedente1, antecedente2, atual, numeroDeTermos, i;

      leia(numeroDeTermos);

                     {Imprima os dois primeiros termos da série}
      antecedente1 ←  1;
      antecedente2 ←  1;
      escreva(antecedente1, antecedente2);

      {Gere e imprima os termos da seqüência a partir do terceiro
            termo até o n-ésimo}

      para i ←  3 até numeroDeTermos
         faça
            atual ←  antecedente1 + antecedente2;
            imprima(atual);

                           {Atualize os termos antecedentes}

            antecedente1 ←  antecedente2;
            antecedente2 ←  atual;
```

Capítulo 3

INTRODUÇÃO À LINGUAGEM PASCAL

3.1 A Linguagem Pascal

A linguagem Pascal foi desenvolvida em 1971 por Nicklaus Wirth e é uma das linguagens mais utilizadas no ensino de programação. Sua popularidade deve-se à simplicidade de sua sintaxe. Pascal também provê facilidades (por exemplo, estruturas de controle como aquelas vistas no capítulo anterior) para a escrita de **programas estruturados**; i.e., programas que são fáceis de ler, entender e modificar. A linguagem Pascal possui muitas características que serão vistas à partir deste capítulo e à medida em que a introdução de cada uma delas se fizer necessária.

3.2 Forma Geral de um Programa em Pascal

Um programa em Pascal tem a seguinte forma geral (nesta ordem):

<Cabeçalho do Programa>
<Seção de Declarações>
<Corpo do Programa>

Estas partes serão descritas nas subseções seguintes.

3.2.1 Cabeçalho do Programa

O cabeçalho de um programa Pascal deve conter a palavra reservada **program** seguida do nome do programa, que pode ser qualquer um que o programador queira dar (desde

que certas regras de formação sejam obedecidas, como se verá mais tarde). Finalmente, o nome do programa deve ser seguido por **(input, output)**[4] e ";".

A expressão (*input*, *output*) que segue o nome do programa no cabeçalho não é requerido na implementação particular de Turbo Pascal, mas é obrigatório em Pascal padrão. Portanto um cabeçalho de programa como:

```
program MeuPrograma;
```

é válido em Turbo Pascal, mas não em Pascal padrão.

3.2.2 Seção de Declarações

A seção de declarações de um programa não contém instruções propriamente ditas. Em vez disso, nesta seção, o programador declara (i.e., define ou especifica) os objetos a serem utilizados no programa. Estes objetos podem ser **constantes**, **variáveis**, **tipos de dados** ou **procedimentos** definidos pelo programador. Declarações de constantes e variáveis serão vistas a seguir, enquanto que declarações de tipos de dados e procedimentos definidos pelo programador serão vistas mais adiante.

3.2.3 Corpo do Programa

O corpo do programa contém as instruções que implementam, na linguagem Pascal, as instruções escritas previamente na linguagem algorítmica e que efetivamente devem resolver o problema proposto. O corpo do programa deve começar com a palavra reservada **begin** e terminar com a palavra **end** seguida de ponto. Note que **begin** e **end** são apenas delimitadores; eles não são instruções ou declarações.

O corpo de um programa em Pascal é normalmente constituído por uma seqüência de instruções. Utiliza-se ponto-e-vírgula como separador de instruções (ou declarações) contíguas, e a ausência deste, quando o mesmo é necessário, resulta num erro de compilação. Não é necessário colocar-se ponto-e-vírgula separando a última instrução de uma seqüência de instruções e o **end**, porque este último não é instrução, mas sim um delimitador. Entretanto, o uso de ponto-e-vírgula neste caso não acarreta em erro.

---

[4] Na realidade, nem sempre é (**input**, **output**); mas por enquanto este conhecimento é suficiente. Voltaremos a este ponto no Capítulo 6.

3.3 Identificadores

Cada objeto (por exemplo, variáveis, constantes, tipos de dados, etc.) utilizado num programa escrito em linguagem de alto nível deve ter um nome que o identifica. Este nome utilizado para identificar um objeto é chamado precisamente de **identificador**. Cada linguagem de programação especifica regras (por exemplo, comprimento máximo, caracteres permitidos, etc.) para a composição de identificadores. O comprimento de um identificador em Pascal depende da implementação da linguagem utilizada, mas usualmente este requerimento permite ao programador *sensato* utilizar identificadores tão longo quanto queira (identificadores muito longos, por exemplo, com 50 caracteres, não podem ser considerado sensatos!). Identificadores em Pascal (e na maioria das linguagens de programação) devem começar com uma letra e não devem conter caracteres especiais (por exemplo, +, £, !, etc.), com exceção do caracter "_" (sublinha). Letras maiúsculas e minúsculas não fazem diferença em Pascal (isto nem sempre é verdade em outras linguagens de programação). Isto significa, por exemplo, que os identificadores `MeuPrograma`, `MEUPROGRAMA` e `MEUprograma` representam o mesmo identificador[5].

Palavras reservadas de uma linguagem de programação são identificadores padrão que possuem significado especial naquela linguagem. Por exemplo, "program", "begin" e "end" são palavras reservadas em Pascal. Não é permitido o uso de palavras reservadas como identificadores.

Exemplos de uso de identificadores legais:

```
x, x1, Minha_Variavel_Numero_1
```

Exemplos de uso ilegal de identificadores:

`1x` (começa com número), `Minha Variavel` (contém espaço em branco - um caracter especial), `var` ("**var**" é uma palavra reservada em Pascal).

[5] Algumas implementações não aceitam identificadores contendo letras acentuadas (por exemplo, um identificador como `Valor_Máximo` pode ou não ser considerado legal numa dada implementação da linguagem Pascal). Portanto, é melhor evitar o uso de acentuação em identificadores.

3.4 Declarações de Constantes

Constantes são objetos de dados cujos valores nunca mudam num programa. Declarações de constantes vinculam identificadores com valores constantes. O uso de identificadores de constantes (em vez dos próprios valores) tem dois objetivos principais em programação: (1) um identificador é usualmente muito mais legível do que um valor (por exemplo, o identificador "`PI`" é muito mais legível do que `3.14`); (2) se uma constante é usada em várias partes de um programa e o programa precisa ser modificado de modo que o valor desta constante deve ser atualizado (por exemplo, passando de `3.14` para `3.1416`), é necessário que o valor desta constante seja atualizado apenas uma vez, i.e., na declaração (definição) que associa o identificador com o valor.

Declarações de constantes seguem a seguinte forma em Pascal (note que ";" faz parte da declaração)[6]:

const <identificador> = <valor>;

Numa seção de declaração de constantes, a palavra **const** deve aparecer apenas uma vez; se houver mais de uma constante sendo definida, estas e seus valores devem ser separados por ponto-e-vírgula. Exemplo de uma seção de declaração de constantes:

```
const
      TERMINAL = '$';
      PI = 3.141592;
      CARGA = 1.6E-19;
      FIM = false;
      LIMITE = 1000;
```

É considerado bom estilo de programação utilizar diferentes convenções para nomes de identificadores que representam tipos de objetos diferentes (por exemplo, variáveis e constantes). Embora a linguagem Pascal não faça distinção entre maiúsculas e minúsculas em nomes de identificadores, tem-se utilizado a convenção de que identificadores de constantes são escritos com todos os caracteres em maiúsculas. Se um identificador de constante representa mais de uma palavra (para o programador), ele deve ser escrito com sublinha para melhorar a legibilidade. Por exemplo, `MINHA_CONSTANTE` é mais legível do que `MINHACONSTANTE`.

[6] Nomes escritos entre "<" e ">" representam objetos arbitrários aceitáveis daquele tipo sendo descrito. Por exemplo, <*valor*> representa qualquer valor ***aceitável*** em Pascal.

3.5 Tipos de Dados Elementares

Pascal oferece vários tipos de dados predefinidos que o programador pode utilizar. Aqui serão discutidos quatro tipos de dados elementares providos por Pascal: **integer**, **real**, **char** e **boolean**.

3.5.1 O Tipo Integer

O tipo **integer** representa números inteiros num intervalo que depende da implementação de Pascal utilizada. Por exemplo, numa dada implementação, este intervalo poderia ser [-2^{15}, ..., -1, 0, 1, ..., 2^{15} -1]. Neste exemplo particular, isto significaria que inteiros abaixo (i.e., menores do que -32768) ou acima (i.e., maiores do que 32767) deste intervalo não poderiam ser utilizados. As seguintes operações aritméticas são permitidas sobre números do tipo **integer**:

adição	+
subtração	-
multiplicação	*
divisão	**div**
resto de divisão	**mod**
inversão de sinal	-

3.5.2 O Tipo Real

O tipo de dados **real** representa (de uma forma limitada) um intervalo de números reais, ou, mais precisamente, números com uma **parte inteira** e outra **fracionária** separados por um ponto. Estes números são também denominados de **números de ponto flutuante**. Portanto, números reais em Pascal são constituídos por uma seqüência de dígitos contendo um ponto decimal entre eles[7]. Em Pascal, números reais também podem ser escritos em notação científica. Neste caso, o número deve terminar com "`E`" seguido de um número inteiro que representa um expoente. Por exemplo, `1.6E-19`

[7] Turbo Pascal permite que um número real comece ou termine com ponto, mas Pascal padrão não permite isto. Por exemplo, "`.15`" e "`20.`" seriam válidos em Turbo Pascal, mas não em Pascal padrão.

representa o número 1.6 x 10^{-19}. Os valores admitidos para números reais são dependentes de implementação.

As seguintes operações aritméticas são permitidas sobre números do tipo real:

adição	+
subtração	-
multiplicação	*
divisão	/
inversão de sinal	-

3.5.3 O Tipo Char

O tipo **char** representa caracteres em Pascal, e o conjunto de caracteres permitidos depende de implementação (usualmente, o conjunto **ASCII** é utilizado). Este conjunto de caracteres é constituído por caracteres imprimíveis (letras, dígitos e caracteres especiais) e caracteres não-imprimíveis (`ESC`, `RETURN`, etc.). Objetos do tipo **char** devem ser representados entre apóstrofos num programa.

Objetos do tipo **char** não possuem operações definidas sobre eles, i.e., operações que resultem em novos objetos do tipo char. No entanto, operações relacionais (=, >, <, etc.) sobre estes objetos são permitidas. Neste caso, os operadores "<" e ">" representam ordem de precedência, conforme mostrado no quadro a seguir:

```
'0' < '1' < '2' < ... < '9' <

'A' < 'B' < ... < 'X' < 'Y' < 'Z' <

 'a' < 'b' < ... < 'x' < 'y' < 'z'
```

Observe no quadro acima que os dígitos precedem as letras maiúsculas, que por sua vez precedem as letras minúsculas (por exemplo, `'0' < 'A' < 'a'`).

3.5.4 O Tipo Boolean

O tipo **boolean** corresponde ao tipo lógico visto no capítulo anterior. Isto é, uma variável do tipo **boolean** é uma variável que pode assumir apenas o valor **true** ou o valor **false**, que são palavras reservadas que significam respectivamente verdadeiro e falso. As operações permitidas para este tipo de dado também já foram discutidas antes e são as seguintes:

conjunção	**and**
disjunção	**or**
negação	**not**

Essas operações correspondem respectivamente aos operadores lógicos **e**, **ou,** e **não** vistos anteriormente.

3.6 Declarações de Variáveis

Declarações de variáveis seguem a seguinte forma em Pascal:

var <identificador> **:** <tipo de dado>;

Onde <*tipo de dado*> pode ser qualquer tipo de dado predefinido na própria linguagem Pascal ou um tipo de dado definido pelo programador. Quando se têm várias variáveis de um mesmo tipo pode-se declará-las de uma só vez separando-as por vírgulas (v. exemplo abaixo).

Exemplo de uma seção de declaração de variáveis:

```
var
   minhaVariavelInteira : integer;
   raiz1, raiz2, determinante : real;
   condicao : boolean;
```

A convenção a ser adotada aqui para a escrita de identificadores de variáveis é a de que variáveis que não representam mais de uma palavra (para o programador) são escritas em letras minúsculas; se elas representam mais de uma palavra (para o programador), a primeira palavra é escrita em minúsculas e as palavras seguintes começam por

maiúsculas. Evita-se assim a necessidade do uso de sublinha ("_") para melhorar a legibilidade de variáveis. Por exemplo, `minhaVariavelInteira` é mais prático de ser escrito do que `minha_variavel_inteira`.

3.7 Atribuição

Uma instrução de atribuição em Pascal possui a seguinte forma:

<variável> := <expressão>

Onde, o símbolo ":=" corresponde a "←" na linguagem algorítmica vista no *Capítulo 2*, e <*variável*> deve ser uma variável de um tipo compatível com o tipo de dado resultante da <*expressão*>. Por exemplo, uma atribuição como:

```
minhaVariavelInteira := 3.5 + 1;
```

irá causar um erro se `minhaVariavelInteira` for declarada como uma variável inteira porque o valor resultante na expressão será real, que não é compatível com o tipo inteiro. Observe, entretanto, que a recíproca é válida; i.e., atribuir um valor inteiro a uma variável real é legal em Pascal.

3.8 Expressões Aritméticas e Funções Predefinidas

Expressões aritméticas são constituídas de variáveis e constantes numéricas, operadores e funções aritméticos. As regras para a escrita de expressões aritméticas (inclusive as regras de precedência) são as mesmas vistas anteriormente para algoritmos.

Exemplos de expressões aritméticas válidas em Pascal:

```
b*b - 4*a*c
1/(a*a + b*b)
a * (-(b + c))
```

Os operadores +, -, * e / podem ser usados tanto com números inteiros quanto com reais, e o tipo do resultado de uma expressão mista contendo números inteiros e reais

sempre resulta num tipo real. Note também que o operador "/" sempre resulta num número real, mesmo quando a divisão entre dois inteiros é exata (por exemplo, `4/2` resulta no real `2.0` e não no inteiro `2`). Lembre-se que um tipo real não pode ser atribuído a uma variável inteira, embora a recíproca seja válida.

Pascal provê ainda várias **funções aritméticas predefinidas** como as seguintes:

Função	Descrição
abs(x)	O valor absoluto do argumento, que pode ser inteiro ou real. O resultado será do mesmo tipo do argumento.
sqr(x)	O quadrado do argumento, que pode ser inteiro ou real. O resultado será do mesmo tipo do argumento.
sqrt(x)	A raiz quadrada positiva do argumento. O argumento pode ser inteiro ou real, mas deve ser positivo. O resultado será sempre real.
round(x)	O valor inteiro mais próximo do argumento que deve ser real.
trunc(x)	A parte inteira do argumento que deve ser real.
sin(x)	O seno do argumento que representa um arco em radianos.
cos(x)	O co-seno do argumento que representa um arco em radianos.

Expressões mais complexas do que as aritméticas também envolvem operadores relacionais e lógicos. As regras de formação para estas expressões são as mesmas vistas no capítulo anterior sobre algoritmos.

3.9 Outras Funções Predefinidas

Os tipos de dados **integer**, **boolean** e **char** são considerados **tipos ordinais**. Isto significa que cada elemento, exceto o primeiro e o último, de cada um destes tipos possui um **predecessor** e um **sucessor**. Por exemplo, o sucessor de `5` (um tipo **integer**) é 6 e seu predecessor é 4. A função predefinida **ord** resulta na ordem de um tipo ordinal em sua seqüência de valores. A ordem de um objeto do tipo **integer** é o próprio objeto (por exemplo, `ord(-19) = -19`; `ord(0) = 0`, etc.). Em outros tipos ordinais, o número ordinal do primeiro valor na seqüência é 0, o segundo é 1, e assim por diante. Por exemplo, os valores do tipo **boolean** são ordenados como {**false**, **true**}; portanto, `ord(false) = 0` e `ord(true) = 1`. As funções predefinidas **pred** e **suc** determinam, respectivamente, o predecessor e o sucessor de um tipo ordinal, se estes existirem. Por exemplo, `pred(true) = false`, mas `pred(false)` é indefinido (i.e., pode resultar em erro quando da execução do programa).

Embora as funções **ord**, **pred** e **suc** possam ser utilizadas com qualquer tipo de dado ordinal, elas são utilizadas com mais freqüência com o tipo **char**. O número ordinal de um caracter é baseado no código do conjunto de caracteres utilizado e, portanto, é dependente de implementação. Um conjunto de caracteres bastante utilizado é o ASCII (pronuncie "ásqui"), cujo nome é derivado de *American Standard Code for Information Interchange*. Neste conjunto de caracteres, tem-se, por exemplo, `ord('B') = 66`, `ord('C') = 67`, `ord('7') = 55` (não confunda o dígito "7" neste exemplo, que é um caracter, com o número inteiro "7").

A função predefinida **chr** tem um resultado inverso que a função **ord** para caracteres. Em outras palavras, a função **chr** retorna um caracter cujo número ordinal corresponde ao seu argumento. Por exemplo, `chr(66) = 'B'`, `chr(67) = 'C'`, `chr(55) = '7'` (dígito; não número).

Existem outras funções predefinidas em Pascal que não foram discutidas aqui. Estas funções serão introduzidas à medida em que se fizerem necessárias. Além disso, algumas implementações de Pascal provêem outras funções predefinidas que não são especificadas na definição de Pascal padrão. O programador deve ter ciência de todas as funções predefinidas em Pascal padrão de modo que ele possa beneficiar-se delas, i.e., sem ter que programar funções que já existem prontas. Entretanto, o programador deve evitar utilizar funções que são específicas de uma implementação particular de Pascal se ele deseja que seus programas sejam portáveis para outros computadores diferentes.

3.10 Entrada e Saída

Serão discutidas aqui algumas funções predefinidas que servem para entradas e saída de dados para programas em Pascal.

3.10.1 Instruções de Entrada

Existem duas funções básicas para entrada de dados em Pascal: **read** e **readln**. A função **readln** tem a seguinte forma geral:

readln(<lista de entrada>)

onde <*lista de entrada*> é uma lista de variáveis separadas por vírgulas às quais os valores lidos serão atribuídos. Quando uma instrução de entrada (**read** ou **readln**) é

executada, a execução do programa é suspensa até que os dados requeridos sejam introduzidos pelo usuário do programa. Deve haver um dado para cada variável na lista de entrada, e a ordem dos dados corresponde à ordem das variáveis na lista de entrada. Os dados devem ser introduzidos numa mesma linha e deve haver um espaço depois de cada *dado numérico* que não seja o último. A tecla `<ENTER>` deve ser pressionada ao final de cada entrada de dados. Por exemplo, a instrução:

```
readln(c1, c2, i, r)
```

onde `c1`, `c2` são variáveis declaradas como sendo do tipo **char**, i é do tipo **integer** e r é do tipo **real**, poderia ser utilizada para ler uma entrada de dados to tipo:

```
ab10 3.14<ENTER>
```

Neste caso, a `c1` seria atribuído o valor 'a', `c2` receberia 'b', `i` receberia `10` e `r` receberia `3.14`. Note que, neste exemplo, existe um espaço em branco entre os valores `10` e `3.14`.

A instrução **read** tem basicamente o mesmo formato da instrução **readln**:

read(<lista de entrada>)

No entanto, **read** and **readln** funcionam de forma diferente. Sempre que uma instrução de entrada (**read** ou **readln**) é executada, um programa em Pascal continuará a ler quaisquer dados (se houver algum) que não tenham sido processados na linha de dados prévia antes de passar para a próxima linha de dados. O efeito que diferencia **readln** de **read** é que **readln** não *deixa* nenhum dado para ser lido pela próxima instrução de leitura. Em outras palavras, quando executada, uma instrução **readln** *salta* os dados que não foram lidos na instrução corrente e *pula* para a próxima linha de dados. Considere o seguinte exemplo, onde `c1`, `c2` e `c3` são variáveis do tipo **char**:

```
read(c1, c2);
read(c3);
```

Se o usuário do programa contendo estas instruções digitar a seqüência de caracteres:

```
abc<ENTER>
```

a `c1`, `c2` e `c3` serão atribuídos os caracteres a, b e c, respectivamente. No entanto, se aquelas instruções fossem substituídas por:

```
readln(c1, c2);
readln(c3);
```

a `c1` e `c2` serão atribuídos os caracteres a e b como antes, mas o caracter c não será lido (a não ser que o usuário digite-o novamente na próxima linha de dados).

Esta distinção entre **readln** e **read** é aquela prescrita pelo Pascal padrão. Em Turbo Pascal, as coisas são diferentes. Isto é, se não for informado explicitamente que deve proceder como Pascal padrão nessas situações, o Turbo Pascal sempre lê uma nova linha de dados quando uma instrução **read** ou **readln** é executada. O modo de se informar Turbo Pascal a proceder como descrito acima é por meio de uma **diretiva de compilador**, que, no caso, seria `{$B-}`. Uma diretiva de compilador deve ser escrita antes de começar o programa propriamente dito (i.e., antes do cabeçalho do programa) e não faz parte do próprio programa. Em outras palavras, uma diretiva de compilador é uma forma de se informar ao compilador como ele deve interpretar (i.e., traduzir) o programa. Outras diretivas serão vistas mais adiante quando elas se fizerem necessárias.

Existe ainda uma última instrução de entrada que é **readln** sem lista de entrada. O efeito desta instrução é o de forçar com que a próxima instrução de entrada sempre leia dados numa nova linha de dados. Evidentemente, uma instrução deste tipo só será útil quando houver algum dado *deixado para trás* na linha de dados prévia (i.e., provavelmente, após um **read**).

3.10.2 Instruções de Saída

Existem três instruções básicas de saída em Pascal. As instruções de saída **write** e **writeln** têm basicamente o mesmo formato:

```
writeln(<lista de saída>)
write(<lista de saída>)
```

onde *<lista de saída>* é uma lista de variáveis, constantes ou expressões separadas por vírgulas. Cadeias de caracteres também são permitidas e estas devem ser colocadas entre apóstrofos ('). O efeito destas instruções é o de imprimir no meio de saída corrente (usualmente a tela do computador) o valor de variáveis e constantes, o resultado de expressões e as cadeias de caracteres na lista de saída. A diferença entre **write** e **writeln** é que a última faz com que o cursor avance para o início da próxima linha no final da saída de dados, enquanto que a primeira não produz este efeito. Por exemplo, o trecho de programa:

```
variavelInteira1 := 15;
variavelInteira2 = 2;
writeln('Resultados: ', variavelInteira1, ' ', variavelInteira2);
```

produziria uma impressão como:

```
Resultados: 15    2
```

e faria o cursor passar para linha seguinte. Note que, neste exemplo, o efeito produzido pela cadeia de caracteres constituída de espaços em branco (' ') na lista de saída é o de separar os dois valores das variáveis; caso contrário, os valores seriam impressos "colados" como `152`.

Números inteiros podem ser formatados para impressão através de **descrições de formato**. Uma tal formatação informa ao computador qual é o *tamanho* (i.e., o número de posições a serem ocupadas) do número a ser impresso. Uma descrição de formato deve ter a seguinte forma seguindo uma variável do tipo **integer**:

```
:<tamanho>
```

Por exemplo, a instrução:

```
writeln('Valor inteiro: ', minhaVarInteira :4);
```

informa ao computador que `minhaVarInteira` deve ser impressa utilizando quatro posições no meio de saída; se o valor da variável não preencher todos estes espaços reservados, brancos serão adicionados à esquerda do número impresso. Note também que o sinal (se existir) de um número conta para o cálculo do número de posições ocupadas pelo número.

Pode-se ainda utilizar descrições de formato de números reais em listas de saída. Tal descrição de formato especifica o tamanho (i.e., o número total de dígitos mais um referente ao ponto decimal mais um referente ao sinal se for o caso) e o número de casas decimais do número real a ser impresso. Uma descrição de formato deve ter a seguinte forma seguindo uma variável do tipo **real**:

```
:<tamanho>:<número de casas decimais>
```

Por exemplo, um programa que imprimisse a altura de uma pessoa em metros (armazenada na variável `altura`) com duas casas decimais poderia ter a seguinte

instrução de saída (não esqueça que o ponto também conta no cálculo do número total de posições):

```
writeln('Altura: ', altura :4:2);
```

Neste exemplo, um número com mais de duas casas decimais seria arredondado para duas casas decimais antes de ser impresso (por exemplo, `1.73677` seria arredondado para `1.74`). Note ainda que um número real com um número de posições *menor* do que o especificado é impresso precedido por espaços em branco. Por outro lado, um número com um número de posições *maior* do que o especificado poderá não ser impresso[8]. Lembre-se que se uma variável pode assumir um valor negativo, a posição ocupada pelo sinal também deve ser levada em consideração na especificação do tamanho do campo. Uma regra prática para a especificação de formatos de números reais: se você não tem certeza do *tamanho* do número que vai ser impresso, é melhor superdimensionar do que subdimensionar a especificação.

Existe ainda uma outra instrução **writeln** utilizada sem argumentos (i.e., sem lista de saída) cujo único efeito é o de fazer com que o cursor avance para a primeira coluna da próxima linha (em outras palavras, **writeln** sem argumentos serve para *pular linhas* na saída).

3.10.3 O Uso de Avisos

O uso de **avisos** (*prompts*) precedendo entradas de dados (i.e., instruções **read** ou **readln**) é altamente recomendável em programação. Estes avisos informam ao usuário do programa, por meio de instruções **write** ou **writeln**, que tipo de entrada de dados o programa espera que o usuário introduza. Por exemplo, no trecho de programa a seguir, a instrução **write** *avisa* ao usuário que o programa está esperando que ele introduza sua altura *em metros* (isto é importante porque o usuário poderia pensar *em centímetros*!) antes de executar a instrução **readln**.

```
var altura : real;

write(‘Introduza sua altura em metros e digite enter: ');
readln(altura);
```

[8] Este último fato depende de implementação; o Turbo Pascal expande automaticamente a especificação para que o número seja impresso.

3.11 Comentários

Comentários em Pascal podem ser colocados entre "{" e "}" ou entre "(*" e "*)"[9]. Comentários podem conter qualquer seqüência de caracteres e são completamente ignorados pelo compilador. Eles podem ser colocados em qualquer parte do programa, e regras para escrita de comentários para tornar o programa legível já foram discutidas no *Capítulo 2*.

3.12 Estruturas de Controle

Pascal possui estruturas de controle análogas àquelas vistas na linguagem algorítmica apresentada no *Capítulo 2*. As estruturas de controle de Pascal serão vistas a seguir.

3.12.1 Estruturas Condicionais e Operadores Relacionais

Instruções condicionais permitem que sejam tomadas decisões que dependem de dados de entrada do programa. Um exemplo no qual uma tomada de decisão seria necessária seria um programa de cálculo de imposto de renda devido. Sabe-se que a alíquota a ser deduzida depende da renda bruta do contribuinte. Por exemplo, se a renda bruta for maior do que R$ 100.000,00 a alíquota é de 25%; se a renda bruta for maior do que R$ 200.000,00 a alíquota é de 40%; e assim por diante (estes dados são hipotéticos!). Portanto, um programa para cálculo de imposto teria que ter instruções condicionais que implementassem esta tomada de decisão baseada na renda do contribuinte.

Uma instrução condicional em Pascal tem a seguinte forma:

if <condição> **then** <seqüência de instruções 1> **else** <seqüência de instruções 2>

Tal instrução condicional em Pascal tem exatamente o mesmo significado que uma instrução **se ... então ... senão** correspondente em linguagem algorítmica. Uma condição é uma expressão booleana que resulta em verdadeiro ou falso. Quando esta expressão resulta em verdadeiro, a seqüência de instruções correspondentes ao **then** é executada; quando esta expressão resulta em falso, a seqüência de instruções correspondentes ao **else** é executada. Note que, quando qualquer seqüência de instruções contém mais de

[9] Não existe nenhuma diferença entre estes delimitadores de comentários, mas não se pode abrir um comentário com "{" e fechá-lo com "*)" ou vice-versa.

uma instrução, ela deve começar por **begin** e terminar com **end** (na realidade, isto é o caso para qualquer seqüência de instruções em Pascal). Um erro bastante comum entre os novatos em programação ocorre quando há apenas uma instrução no **then** e utiliza-se um ponto-e-vírgula para separar esta instrução e o **else** (ou quando há várias instruções delimitadas entre **begin** e **end** e coloca-se um ponto-e-vírgula entre o **end** e o **else**). No entanto, se existe uma seqüência de instruções seguindo o **else** e existe outra instrução seguindo o **if**, deve-se colocar um ponto-e-vírgula após o **end** da seqüência de instruções do **else** para separar o **if** da próxima instrução. Note ainda que o **else** é opcional; em outras palavras, se não há nada a ser feito quando a condição é falsa, o **else** pode ser deixado de fora.

Exemplo 1:

```
if (x > 0)
     then x := x + 1;
     else x := 0;
```

Isto resultará num erro de compilação porque não deve haver ponto-e-vírgula separando a instrução x := x + 1 e o **else**.

Exemplo 2:

```
if (z = 0)
   then z := 1
   else
      begin
         z := z + 10;
         writeln(z)
      end;
readln(x);
```

Note neste último exemplo que o ponto-e-vírgula é necessário para separar as instruções **if** e **readln**(x).

Exemplo 3:

```
if (y >= 0)
     then y := y + 1;
```

Esta última instrução **if** está correta quando não há nada a ser feito quando y < 0.

O modo mais comum de se escreverem expressões condicionais é através de **operadores relacionais** que já foram discutidos no capítulo precedente sobre construção de algoritmos. Pascal permite o uso de todos os operadores relacionais vistos naquele capítulo, mas alguns utilizam notações diferentes daquelas vistas anteriormente. Os operadores relacionais em Pascal são apresentados na tabela a seguir:

Símbolo	Significado
<	menor do que
<=	menor do que ou igual a
>	maior do que
>=	maior do que ou igual a
=	igual a
<>	diferente de

Instruções condicionais **if** podem ser aninhadas de modo a poderem representar várias alternativas, em vez de apenas duas como foi visto até aqui. Uma instrução **if** aninhada ocorre quando a instrução seguindo um **then** ou **else** também é um **if**. Por exemplo, a instrução **if** abaixo:

```
if (x > 0) then
  if (y > 1) then
     x := x + y
  else
     x := x - y
else
    x := 10;
```

é uma instrução **if** aninhada porque a instrução seguindo o primeiro **then** (i.e., a instrução a ser executada quando x > 0) é também uma instrução **if**.

Cuidados especiais devem ser tomados quando da escrita de expressões **if** mais complexas. Considere o seguinte exemplo onde os graus de estudantes são obtidos de acordo com a nota obtida (por exemplo, uma nota maior do que `9.0` eqüivale ao grau `A`, uma nota maior do `8.0` e menor do que `9.0` eqüivale ao grau `B`, etc.). Uma maneira correta de se escrever a instrução de decisão **if** seria:

```
if nota >= 9.0 then
    write('A')
else if nota >= 8.0 then
    write('B')
else if nota >= 7.0 then
    write('C')
else if nota >= 6.0 then
    write('D')
else if nota >= 5.0 then
    write('E')
else
    write('F')
```

No entanto, a instrução **if** a seguir:

```
if nota >= 5.0 then
    write('E')
else if nota >= 6.0 then
    write('D')
else if nota >= 7.0 then
    write('C')
else if nota >= 8.0 then
    write('B')
else if nota >= 9.0 then
    write('A')
else
    write('F')
```

produziria um efeito indesejável, pois todos os alunos com nota superior a `5.0` seriam incorretamente categorizados como "`E`".

3.12.2 Laços de Contagem e a Instrução FOR

Um **laço de repetição** é uma estrutura contendo uma ou mais instruções que se repetem um determinado número de vezes ou até que uma dada condição seja satisfeita. A capacidade de representar laços de repetição numa linguagem de programação é extremamente importante e Pascal provê três tipos de laços de repetição que serão examinados nesta e nas próximas duas seções.

Laços de repetição com contagem, ou, abreviadamente, **laços de contagem**, são utilizados quando se pode especificar a priori o número exato de repetições requeridas. A instrução **for** em Pascal pode ser utilizada para representar laços de contagem. A instrução **for** tem a seguinte forma geral:

for <variável de controle> := <valor inicial> **to** <valor final> **do**
<corpo do laço>

onde, a **variável de controle** do laço é uma variável de um tipo ordinal (**integer**, **char** ou **boolean**, embora a primeira seja utilizada na vasta maioria das vezes); *<valor inicial>* e *<valor final>* são os valores entre os quais (e inclusive) a variável de controle pode variar durante a execução do laço. Estes valores podem ser constantes, variáveis ou expressões que resultam num valor do mesmo tipo que a variável de controle. O corpo do laço é constituído de uma ou mais instruções, sendo que neste último caso estas instruções devem ser delimitadas por **begin** e **end**.

Exemplo de uso da instrução **for**:

```
for i := 1 to 10 do
  begin
     write('A raiz de ', i, 'eh: ');
     x := sqrt(i);
     writeln(x)
  end
```

O resultado da execução do trecho de programa acima seria o de calcular e imprimir as raízes dos dez primeiros inteiros positivos.

Como funciona a instrução **for**? Primeiro, o valor inicial da variável de controle é calculado (obviamente, se este não for uma constante) e atribuído a esta variável; então, o valor final também é calculado (se não for uma constante) e comparado com o valor inicial da variável de controle. Se o valor da variável de controle for menor do que ou igual ao valor final, o corpo do laço é executado e, ao final da execução, a variável de controle é incrementada de um (i.e., soma-se 1 ao valor da variável de controle). Então, testa-se novamente este novo valor da variável de controle com o valor final; se este ainda for menor do que ou igual ao valor final, executa-se novamente o corpo do laço e incrementa-se novamente a variável de controle, e assim por diante. Este procedimento de execução do corpo do laço, e incremento e teste da variável de controle é repetido até que esta assuma um valor que excede o valor final. Quando isto ocorre, o laço é terminado e o controle passa para a próxima instrução seguindo o laço (se houver alguma). Convença-se de que entendeu este funcionamento da instrução **for**, pois este conhecimento é muito valioso na depuração de programas contendo este tipo de instrução.

Outras observações são importantes para se evitarem efeitos indesejáveis ("*bugs*") no uso de instruções **for**. Em primeiro lugar, note que o valor final é calculado apenas uma vez (se necessário). Isto significa que se este valor for calculado a partir de uma expressão contendo uma variável que muda de valor dentro do corpo do laço, esta mudança de valor não vai afetar o valor final. Por exemplo, no trecho de programa:

```
j := 2;
for i := 1 to 2*j do
   j := j + 2;
```

o valor final representado por `2*j` será calculado apenas uma vez quando da entrada no laço (resultando em `4` neste exemplo) e apesar do fato de este valor depender de `j`, ele não será atualizado quando `j` mudar de valor.

Se o valor final for menor do que o valor inicial, o corpo do laço não será executado nenhuma vez. Se você deseja que um laço varie de um dado valor até um valor menor (ao invés de maior), você deve ter a variável de controle decrementada (i.e., subtraída de 1 no final de cada passagem), ao invés de incrementada. Isto pode ser especificado numa instrução **for** através da substituição de **to** por **downto**. Em outras palavras, uma instrução do tipo:

for <variável de controle> := <valor inicial> **downto** <valor final> **do**
    <corpo do laço>

significa que a variável de controle será continuamente decrementada ao final de cada execução do corpo do laço.

Pascal padrão proíbe que a variável de controle seja modificada dentro do corpo do laço. O Turbo Pascal permite isto, mas, mesmo assim, isto deve ser evitado para evitar efeitos indesejáveis. Note ainda que a variável de controle retém o último valor assumido na execução do laço[10].

3.12.3 A Instrução WHILE

Nos laços de contagem, o número de repetições pode ser especificado antes do início da execução dos laços. Entretanto, isto nem sempre é possível. Em muitas situações, apesar

[10] Isto é o caso em Turbo Pascal e em C, mas não em Pascal padrão que especifica que este valor deve ser indefinido.

de não se poder determinar a priori o número de vezes que um laço deve ser executado, pode-se especificar uma condição tal que o laço será executado até que esta condição seja satisfeita. **Laços condicionais** servem para este propósito. Existem dois tipos de laços condicionais em Pascal: a instrução **while** e a instrução **repeat**.

A instrução **while** em Pascal tem a seguinte forma geral:

while <condição> **do**
    <corpo do laço>

onde <*condição*> é uma expressão booleana e <*corpo do laço*> tem o mesmo significado definido para a instrução **for**. Esta instrução funciona da seguinte maneira: a condição é avaliada e se for verdadeira, o corpo do laço é executado; caso contrário, o laço é encerrado. Após a execução do corpo do laço, volta-se a testar a condição; se esta for ainda verdadeira, o laço é executado novamente, e assim por diante. Usualmente, a execução do corpo do laço modifica a condição de modo a garantir o final da repetição.

Exemplo de uso da instrução **while**:

```
var meuCaracter : char;

read(meuCaracter);
while (meuCaracter <> '.')  do
    read(meuCaracter);
```

Este trecho de programa pode ser utilizado para ler uma seqüência de caracteres até que um ponto ('.') é digitado. Note que o primeiro caracter é lido *antes* do laço; caso contrário, a variável `meuCaracter` seria indefinida.

É importante observar que o final do laço não ocorre no instante em que a condição deixa de ser satisfeita, mas sim no momento em que ela é *avaliada* e não satisfeita. Por exemplo, suponha que existissem outras instruções seguindo o **read** no corpo do laço no último exemplo. Logo após a leitura do ponto, que invalidaria a condição, esta ainda não teria sido avaliada e, portanto, as instruções seguindo o **read** no corpo do laço seriam executadas normalmente. Apenas quando a condição fosse novamente testada

(i.e., ao final do corpo do laço), e se verificasse que ela era falsa, a execução do corpo do laço seria encerrada.

3.12.4 A Instrução REPEAT

A instrução de repetição **repeat** representa um laço de repetição condicional muito parecido com a instrução **while**. A instrução **repeat** tem a seguinte forma em Pascal:

```
repeat
    <corpo do laço>
until <condição>
```

Nesta instrução, o corpo do laço é executado até que a condição de parada seja verdadeira. É importante notar que, diferentemente da instrução **while**, o corpo do laço numa instrução **repeat** é executado pelo menos uma vez, visto que a condição de parada só é testada depois da repetição. Portanto, a instrução **repeat** deve ser utilizada quando o programador tem certeza de que o corpo do laço deve ser executado pelo menos uma vez.

Exemplo de uso da instrução **repeat** (compare cuidadosamente com o exemplo equivalente visto acima para a instrução **while**):

```
var meuCaracter : char;

repeat
    read(meuCaracter)
until (meuCaracter = '.')
```

Como uma ilustração final do uso de laços de repetição, compare como a instrução **for** abaixo seria representada de forma equivalente utilizando-se instruções **while** e **repeat**. (Neste exemplo, `vc` é uma variável de controle inteira; `INICIAL` e `FINAL` representam constantes inteiras; <*seqüência de instruções*> representa uma seqüência de instruções qualquer a ser repetida.)

Instrução **for**:

```
for vc := INICIAL to FINAL do
    begin
      <seqüência de instruções>
    end
```

Instrução **while**:

```
vc := INICIAL;
while vc <= FINAL do
    begin
       <seqüência de instruções>;
       vc := vc + 1
    end
```

Instrução **repeat**:

```
vc := INICIAL;
if vc <= FINAL then
   repeat
       <seqüência de instruções>;
       vc := vc + 1
   until vc > FINAL
```

Claramente, a instrução **for** é mais conveniente para implementar o efeito desejado no exemplo particular acima.

3.12.5 Instrução CASE

A instrução **case** é uma **instrução de decisão** que permite selecionar uma dentre várias alternativas. O formato de uma instrução **case** é:

```
case <seletor> of
     <valor 1> : <instrução 1>;
     <valor 2> : <instrução 2>;
                    .
                    :
     <valor n> : <instrução n>;
else
    <instrução e>
end
```

onde <*seletor*> é uma expressão (que ***não*** pode ser do tipo **real**); <*valor 1*>, ..., <*valor n*> são possíveis valores de <*seletor*>, e <*instrução 1*>, ..., <*instrução n*> são instruções (ou seqüências de instruções) que serão executadas quando <*seletor*> resulta em <*valor 1*>, ..., <*valor n*>, respectivamente; <*instrução e*> seguindo o **else** (que é opcional em Pascal) será executada se <*seletor*> não resultar em nenhum dos valores listados. Note que cada valor listado deve ser diferente um do outro. Note ainda que, se <*seletor*> resultar num valor que não é listado e não houver um **else**, nenhuma instrução dentro do **case** será executada. Finalmente, o tipo de dado de <*seletor*> deve coincidir o tipo de dado de cada um dos valores listados; apenas tipos de dados ordinais (**integer**, **boolean** ou **char**) são permitidos.

Como exemplo de uso da instrução case, suponha que seu programa apresenta três opções, A, B e C, para o usuário escolher uma delas. Para cada uma destas opções, haveria um procedimento distinto a ser seguido. Se o usuário, digitar uma letra diferente de A, B ou C, você quer imprimir uma mensagem de alerta para o usuário informando-o que apenas as opções A, B e C são permitidas. O trecho de programa a seguir implementa isto:

```
read(meuCaracter);
case meuCaracter of
       'A' :
          begin
           (* Execute aqui as ações correspondentes à opção A *)
          end;
       'B' :
          begin
           (* Execute aqui as ações correspondentes à opção B *)
          end;
       'C' :
          begin
           (* Execute aqui as ações correspondentes à opção C *)
          end;
else
   writeln('A opcao digitada nao eh permitida. Digite A, B ou C.')
end
```

A instrução **case** é especialmente útil quando a seleção de casos é baseada no valor de uma única variável ou expressão. Note, entretanto, que a instrução **case** é mais restrita do que a instrução **if** vista acima no sentido de que aquilo que pode ser escrito com um **case** pode ser escrito usando **if**, mas não o contrário. Por exemplo, o exemplo acima poderia ser implementado utilizando-se instruções **if** aninhadas (embora de forma menos elegante), mas se a variável lida fosse do tipo **real**, não seria possível implementar o exemplo utilizando **case**. Sempre que possível, é preferível usar uma

instrução **case** do que uma instrução **if**, pois a primeira será certamente mais legível do que a segunda.

3.12.6 Instrução GOTO

A instrução **goto** é uma instrução que permite que o fluxo do programa seja **desviado incondicionalmente** para um outro trecho de programa que começa por uma instrução identificada por uma declaração **label**. O uso de instruções **goto** e declarações **label** levam a programas que são difíceis de entender e depurar, e, por isso, este uso deve ser evitado. Aliás, uma das motivações para o desenvolvimento das estruturas de controle vistas acima foi exatamente evitar o uso de **gotos**, e assim, tornar os programas mais legíveis e fáceis de ser mantidos.

3.13 Editando, Compilando e Executando um Programa

Suponha, neste ponto, que você tem um algoritmo pronto (idealmente, construído de acordo com as recomendações apresentadas na *Seção 2.4*). Os passos que você tem a seguir para construir um programa codificando seu algoritmo em Pascal são os seguintes:

1. Traduza o algoritmo da linguagem algorítmica para a linguagem Pascal. Se você escreveu o algoritmo corretamente utilizando a linguagem algorítmica vista no *Capítulo 2*, não terá provavelmente dificuldades para fazer esta tradução.

2. Edite o programa utilizando um editor de texto simples (i.e., sem formatação especial). Um editor dirigido por sintaxe (de Pascal), como aquele encontrado no ambiente Turbo Pascal, é o ideal para esta tarefa. Ao terminar a edição do programa, você obterá um **programa-fonte** (ou **arquivo-fonte**) que deverá ter sido salvo (armazenado) em disco.

3. Compile o programa utilizando o compilador Pascal para transformá-lo em programa executável[11]. Freqüentemente, quando se tenta compilar um

[11] Muitos programas consistem de várias unidades (arquivos-fonte) que são compiladas separadamente produzindo um arquivo-objeto para cada unidade. Nenhum destes arquivos contendo código-objeto é executável. Um arquivo executável, nesta situação, é obtido apenas quando estes códigos-objetos são colocados juntos por meio de um editor de ligações.

programa pela primeira vez, isto não é possível devido a **erros de sintaxe** (i.e., falta de conformidade do programa com as regras de formação válidas da linguagem). Não se sinta frustrado com isso. A freqüência com que os erros de sintaxe ocorrem diminui à medida em que você se torna mais experiente, mas mesmo programadores experientes comentem erros de sintaxe por engano. Quando receber uma mensagem de erro emitida pelo compilador, tente primeiro entender a mensagem (infelizmente, nem sempre uma mensagem de erro de compilação é clara), e, então, volte ao editor de texto e corrija o erro apontado pelo compilador. Tente compilar o programa-fonte novamente e repita o processo de correção de erros até que o programa esteja livre de erros sintáticos.

4. Teste o programa e verifique se o mesmo faz realmente aquilo que deveria fazer. Execute o programa com casos que sejam *qualitativamente diferentes* e veja se obtém os resultado esperados (v. *Seção 2.4*). Se o programa não funciona conforme seria esperado, volte ao *Passo 2* acima e edite o programa novamente. Pode também acontecer que uma revisão no projeto do algoritmo seja necessária; neste caso, você deve voltar ao *Passo 1* das recomendações para projeto de algoritmos vistas na *Seção 2.4*.

Para ilustrar o processo acima, suponha que o problema a ser resolvido com o computador é aquele da seqüência de Fibonacci, cuja solução algorítmica aparece no final do *Capítulo 2*. O algoritmo proposto para gerar a seqüência de Fibonacci é novamente apresentado a seguir:

Usualmente, um comando no ambiente de programação, tal como **Build** ou **Make** no Turbo Pascal, é responsável por juntar as diversas partes constituintes de um programa e transformá-las num único arquivo executável.

```
Algoritmo Fibonacci: {Gera e imprime os primeiros n elementos da
                      seqüência de Fibonacci }

    inteiro:  antecedente1, antecedente2, atual, numeroDeTermos, i;

    leia(numeroDeTermos);

                     {Imprima os dois primeiros termos da série}
    antecedente1 ←  1;
    antecedente2 ←  1;
    escreva(antecedente1, antecedente2);

    {Gere e imprima os termos da seqüência a partir do terceiro
         termo até o n-ésimo  }

    para i ←  3 até numeroDeTermos
       faça
          atual ←  antecedente1 + antecedente2;
          imprima(atual);

                           {Atualize os termos antecedentes}

          antecedente1 ←  antecedente2;
          antecedente2 ←  atual;
```

algoritmo acima pode ser traduzido para um programa em Pascal como visto a seguir:

```
Program Fibonacci;
     {Gera e imprime os primeiros n elementos da série de Fibonacci}

var
    antecedente1, antecedente2, atual, numeroDeTermos, i : integer;

begin
    write('Introduza o numero de termos da sequencia de Fibonacci: ');
    readln(numeroDeTermos);

                         {Imprima os dois primeiros termos da série}
    antecedente1 := 1;
    antecedente2 := 1;
    write(antecedente1, ' ', antecedente2)

{Gere/imprima os termos da série a partir do terceiro termo até o n-ésimo}

    for  i := 3   to   numeroDeTermos  do  begin
     atual := antecedente1 + antecedente2;
     write(' ', atual);
                                {Atualize os termos antecedentes}
      antecedente1 := antecedente2;
      antecedente2 := atual
    end   { for }
end.
```

Edite o programa acima no editor do Turbo Pascal exatamente como ele é apresentado. Então, salve o programa-fonte com o nome "`FIB.PAS`" e tente compilá-lo (utilize a opção **Compile** do Turbo Pascal). Se você realmente digitou o programa como apresentado acima, quando você tentar compilar o programa, o cursor irá parar no início da instrução **for** do programa e você obterá a seguinte mensagem de erro do compilador:

```
Error 85: ";" expected.
```

Esta mensagem significa que o compilador encontrou um erro de sintaxe em seu programa que o impediu de prosseguir. O erro, conforme indicado na mensagem, foi a ausência de ";" na instrução exatamente anterior ao **for** e o reparo deste erro é bem trivial: simplesmente acrescente o ponto-e-vírgula ausente no final daquela instrução. Faça isso e compile novamente o programa. Agora, o programa é compilado com sucesso (note que o Turbo Pascal apresenta um quadro informando o sucesso da compilação) e está pronto para ser executado. Execute o programa escolhendo a opção **Run**. O programa começará então a ser executado, com a primeira instrução **write** do programa causando a apresentação do seguinte na tela:

```
Borland Pascal Version 7.0 Copyright (c) 1983, 92 Borland International
Introduza o numero de termos da sequencia de Fibonacci:
```

Então, o programa pára na instrução **readln** esperando que você digite um número inteiro representando o número de termos desejados na seqüência de Fibonacci. Digite `10` em resposta a esta solicitação e você obterá como resposta final do programa a seqüência[12]:

```
1  1  2  3  5  8  13  21  34  55
```

Neste ponto, você ainda não tem um programa executável; i.e., um programa que possa ser executado independentemente do ambiente Turbo Pascal. Para obter um programa executável, escolha a opção **Make** (ou **Build**) no menu **Compile**. Faça isso e o Turbo Pascal irá gerar um arquivo executável denominado "`FIB.EXE`" (supondo que você salvou o arquivo-fonte com o nome "`FIB.PAS`"). Este arquivo poderá ser executado simplesmente digitando-se `FIB` (i.e., o primeiro nome do arquivo) no *prompt* do DOS.

[12] Na realidade, você só será capaz de ver este resultado se você utilizar a opção **Output** ou **User Screen** do menu **Debug** (v. *Apêndice B*).

3.14 Procedimentos

A abordagem dividir-e-conquistar de resolução de problemas baseia-se na idéia de dividir o problema dado em subproblemas e resolver cada subproblema de uma forma independente dos outros. Para problemas não-triviais, é desejável que se implemente a solução para cada subproblema como um módulo separado, de modo que se possa concentrar em cada um destes módulos independentemente. Em Pascal, isto pode ser realizado por meio de **procedimentos**.

Uma **declaração de procedimento** em Pascal é similar à de um programa e contém três partes: (1) **cabeçalho** (2) **seção de declarações**, e (3) **corpo do procedimento**. Declarações de procedimentos em um programa Pascal devem ser feitas seguindo a seção de declarações de variáveis do programa (i.e., antes do corpo do programa).

O cabeçalho de um procedimento começa com a palavra reservada **procedure**, seguida pelo nome do procedimento (um identificador), e finalmente, quando for o caso, uma **lista de parâmetros formais** (este último será discutido abaixo).

A seção de declarações de um procedimento contém quaisquer declarações de variáveis, constantes ou tipos de dados necessários para implementar o procedimento. Estas declarações têm formatos exatamente iguais àqueles vistos antes em seções de declarações para programas. No entanto, os objetos declarados dentro de um procedimento têm validade apenas no próprio procedimento. Em outras palavras, os identificadores declarados em um procedimento podem ser referenciados *apenas* dentro do próprio procedimento; por isso, eles são ditos serem **locais** ao procedimento.

Finalmente, o corpo do procedimento contém as instruções necessárias e deve começar com a palavra **begin** e terminar com **end;** (o ponto-e-vírgula é obrigatório neste caso).

Para que um procedimento seja utilizado (i.e., executado) num programa, é necessário que ele seja **chamado** pelo programa. Um procedimento que nunca é chamado por um programa (ou por outro procedimento) é completamente inútil. Uma **chamada de procedimento** em Pascal consiste simplesmente do nome do procedimento seguido (se for o caso) por uma **lista de parâmetros reais** (estes serão discutidos logo mais) que são compatíveis com a lista de parâmetros formais do procedimento. Chamadas de procedimentos devem ser colocadas no corpo do programa ou no corpo de um procedimento que utiliza tais procedimentos[13].

[13] Um procedimento pode ainda chamar a si próprio; tal procedimento é chamado de **recursivo**. Procedimentos recursivos serão discutidos no *Capítulo 7*.

Um aspecto interessante do uso de procedimentos na implementação de um programa é que eles nos permitem adiar a implementação de um problema complicado. Outro importante aspecto do uso de procedimentos é que, uma vez escrito, um procedimento pode ser utilizado (i.e., **chamado**) em vários pontos de um programa como se fosse uma nova instrução da própria linguagem Pascal. Finalmente, uma vez implementado e testado, um procedimento pode ser utilizado por vários programas que o necessitem.

Como exemplo de procedimento, considere o procedimento abaixo que simplesmente imprime (*quando chamado*) cinco espaços verticais no meio de saída:

```
procedure Imprime5EspaçosVerticais;
var  i : integer;

begin
   for  i := 1  to  5  do
     writeln;
end;
```

Este procedimento imprimiria cinco linhas verticais no meio de saída sempre que fosse chamado pelo programa principal (ou por outro procedimento) através da execução da chamada:

```
Imprime5EspaçosVerticais;
```

3.15 Parâmetros de Procedimentos

Parâmetros de procedimento permitem que haja comunicação de dados entre dois procedimentos ou entre um procedimento e o programa principal. Uma lista de parâmetros formais identifica os **parâmetros formais** que serão usados dentro do procedimento no lugar dos nomes de variáveis que serão realmente utilizadas quando o procedimento for chamado. Pode-se ainda pensar em parâmetros formais como guardadores de lugares que serão ocupados quando o procedimento for chamado. Uma lista de parâmetros formais é semelhante a uma seção de declaração de variáveis colocada entre parênteses:

(<variável1> : <tipo1>; <variável2> : <tipo2>; ...; <variávelN> : <tipoN>)

Como numa seção de declaração de variáveis, variáveis de um mesmo tipo podem ser agrupadas e separadas por vírgulas antes do nome do tipo, como, por exemplo:

```
procedure MeuProcedimento(x, y : real; n : integer);
```

Um parâmetro formal pode ainda ser precedido pela palavra reservada **var**, cujo significado neste contexto será discutido a seguir.

Cada chamada de procedimento declarado com uma lista de parâmetros formais contém uma lista de **parâmetros reais**, que consiste de variáveis, constantes ou expressões que serão efetivamente utilizadas pelo procedimento. Cada parâmetro formal declarado no cabeçalho deve ser emparelhado com um parâmetro real na chamada do procedimento de modo que o primeiro parâmetro formal é emparelhado com o primeiro parâmetro real, o segundo parâmetro formal é emparelhado com o segundo parâmetro real, e assim por diante. Suponha, por exemplo, que um programa contém a seguinte chamada do procedimento do último exemplo:

```
MeuProcedimento(minhaVariávelReal1, minhaVariávelReal2, 5);
```

as variáveis `minhaVariávelReal1` e `minhaVariávelReal2` (supondo que estas tenham sido declaradas anteriormente como reais no programa) são os parâmetros reais que serão emparelhados com os parâmetros formais `x` e `y` (respectivamente) declarados no exemplo anterior, enquanto que a constante `5` é o parâmetro real a ser emparelhado com o parâmetro formal n. É importante notar que nem sempre a utilização de parâmetros reais constantes é permitida. Por exemplo, se um parâmetro formal é utilizado no lado esquerdo de uma atribuição, ele não pode ser substituído por um parâmetro formal constante pois a uma constante não se pode atribuir nenhum valor.

Existem dois tipos de parâmetros formais em Pascal que influenciam a forma como o emparelhamento de parâmetros (mais conhecido como **passagem de parâmetros**) visto acima pode ser realizado: (1) **parâmetros variáveis** e (2) **parâmetros de valor**. Sintaticamente, a diferença entre parâmetros variáveis e parâmetros de valor é que os primeiros são precedidos pela palavra reservada **var**. A diferença mais importante, no entanto, é que, numa chamada de procedimento, quando parâmetros variáveis são utilizados, as próprias variáveis passadas como parâmetros reais são manipuladas, enquanto que quando parâmetros de valor são utilizados o procedimento faz cópias destas variáveis e manipula estas cópias ao invés de as variáveis em si. Uma conseqüência importante disso é que se um dado parâmetro formal variável for modificado dentro do procedimento, esta mudança será refletida no parâmetro real correspondente. Isto não ocorre no caso de parâmetros de valor.

Quando um procedimento é chamado, Pascal aloca uma ou mais células de memória para cada parâmetro formal de valor. Então, cada uma destas posições de memória são preenchidas com os valores dos parâmetros reais correspondentes (i.e., Pascal faz uma cópia de cada parâmetro real passado por valor). Depois disso, não há mais nenhuma conexão entre parâmetros reais e parâmetros formais de valor. Isto é, quaisquer manipulações destes parâmetros são feitas utilizando-se as cópias feitas quando da chamada do procedimento. Conseqüentemente, quaisquer mudanças nestes parâmetros afetam apenas as cópias dos parâmetros reais, não estes parâmetros em si.

Pascal não permite que uma constante seja passada como parâmetro real de um parâmetro formal variável, mesmo que o parâmetro não seja modificado pelo procedimento. Um parâmetro formal de valor pode ser emparelhado com um parâmetro real constante, variável ou constituído por uma expressão. Utilizam-se parâmetros de valor quando se tem certeza de que os parâmetros reais correspondentes não serão modificados pelo procedimento.

Quando um procedimento é chamado dentro de um programa, o compilador Pascal verifica a lista de parâmetros formais (na declaração do procedimento) para testar a consistência com a lista de parâmetros reais (na chamada do procedimento). Então, as seguintes regras de consistência devem ser satisfeitas:

- **Regra 1**: O número de parâmetros reais na chamada do procedimento deve ser igual ao número de parâmetros formais na lista de parâmetros formais (na declaração do procedimento).

- **Regra 2**: O tipo de cada parâmetro real deve ser igual ao tipo de seu parâmetro formal correspondente.

- **Regra 3**: Um parâmetro real correspondente a um parâmetro formal variável deve ser uma variável. Um parâmetro real correspondente a um parâmetro formal de valor pode ser uma variável, constante ou expressão.

Exemplo: Suponha que um programa Pascal contenha as seguintes declarações de variáveis, constante e procedimento (apenas o cabeçalho do procedimento é apresentado):

```
const    MAX_INT = 32767;

var   x, y : real;
      n : integer;
      caracter : char;

procedure MeuProcedimento( A, B : integer;
                           var C, D : real;
                           var E : char );
```

As seguintes chamadas feitas no corpo do programa principal seriam legais:

```
MeuProcedimento(n + 2, 10, x, y, caracter);
MeuProcedimento(MAX_INT, n, x, y, caracter);
MeuProcedimento(MAX_INT - n, n*10, x, y, caracter);
```

Enquanto que as chamadas a seguir seriam consideradas ilegais (a razão de cada ilegalidade está escrita entre colchetes):

- `MeuProcedimento(n, 2, x, y);` [*O número de parâmetros reais (4) não é igual ao número de parâmetros formais (5).*]

- `MeuProcedimento(2, 10, x, n, caracter);` [*n é inteiro e não corresponde ao tipo do quarto parâmetro formal que é real.*]

- `MeuProcedimento(2*n, 10, x, y, 'A');` [*`'A'` é uma constante (do tipo **char**) e não pode corresponder a um parâmetro variável.*]

- `MeuProcedimento(2*n, 10, x + y, y, caracter);` [*A expressão `x + y` não pode corresponder a um parâmetro variável.*]

- `MeuProcedimento(A, B, C, D, E);` [*Nem `A`, nem `B`, nem `C`, nem `D`, nem `E` foram definidos como variáveis no corpo do programa. Este erro é muito comum entre programadores iniciantes.*]

Um melhoramento no procedimento do exemplo da seção precedente que imprimia cinco espaços verticais seria um procedimento que imprimisse um número qualquer de espaços verticais. O número de espaços desejados deveria então ser um parâmetro do procedimento como visto a seguir:

```
procedure ImprimeEspaçosVerticais(numeroDeEspacos : integer);
var  i : integer;

begin
   for  i := 1  to  numeroDeEspacos  do
      writeln;
end;
```

Parâmetros podem ainda ser classificados como (1) de **entrada**, (2) de **saída**, e (3) de **entrada/saída**:

• Um **parâmetro de entrada** é aquele que serve como dado de entrada para um procedimento, mas não deve retornar modificado. Parâmetros de entrada são *usualmente* representados por parâmetros de valor.

• Um **parâmetro de saída** é aquele que retorna modificado como conseqüência da execução do procedimento, mas que não serve como valor de entrada para o procedimento. Parâmetros de saída *devem* ser parâmetros variáveis.

• Um **parâmetro de entrada/saída** é aquele que não apenas serve como entrada como também serve como saída de um procedimento. Parâmetros de entrada/saída *devem* ser parâmetros variáveis.

Note que a classificação acima serve apenas para orientação do programador. Isto é, não existe nenhuma forma de se informar a Pascal que um parâmetro é de entrada, saída ou entrada/saída. É importante que o programador forneça esta informação em forma de comentários.

3.16 Funções

Em seções precedentes, foram vistas algumas funções predefinidas em Pascal, tais como **abs** e **sqrt**. Pascal também provê facilidades para o programador definir suas próprias funções. Na realidade, uma função pode ser vista como um tipo especial de procedimento que retorna exatamente um valor. Normalmente, os parâmetros formais de uma função são todos parâmetros de valor que não podem ser modificados durante a execução da função. Uma declaração de função é similar à uma declaração de procedimento e tem a seguinte forma geral:

function <nome da função>(<lista de parâmetros formais>) : <tipo do resultado>;

<seção de declarações>

begin
    <corpo da função>
end;

Apesar da similaridade, existem algumas diferenças fundamentais entre declaração de procedimento e declaração de função. Primeiro, numa declaração de função, a palavra **function** deve ser utilizada no lugar de **procedure**. Segundo, deve-se especificar o tipo de resultado a ser retornado pela função. Este tipo pode ser qualquer um dos tipos de dados já vistos e alguns tipos que serão vistos posteriormente, mas deve-se adiantar que nem todo tipo de dado pode ser utilizado aqui. Os outros componentes têm o mesmo significado que na definição de procedimentos vista anteriormente, mas existe uma outra diferença fundamental que não é aparente na definição acima: deve-se sempre atribuir um valor ao nome da função no corpo da mesma. O último valor atribuído ao nome da função é o valor retornado pela função depois que termina a execução do corpo da função.

Uma chamada de função dentro do corpo de um programa ou procedimento é essencialmente diferente de uma chamada de procedimento. Uma função definida pelo programador é chamada exatamente da mesma forma que uma função predefinida em Pascal; i.e., por referência à função numa atribuição ou expressão. As regras de compatibilidade vistas acima para procedimentos continuam válidas para funções (embora funções normalmente não utilizem parâmetros variáveis).

Exemplo de uma declaração de função:

```
function MinhaFuncao(n : integer) : integer;
var i : integer;

begin
    .
    :
    MinhaFuncao := i + 4*n;
end;
```

Esta função poderia ser chamada dentro do corpo de um programa ou procedimento (inclusive outra função) como (por exemplo):

```
minhaVariavel := MinhaFuncao(4);
```

ou

```
minhaVariavel := 10 + 3*MinhaFuncao(7);
```

Neste caso (conforme já foi visto em situações semelhantes), a variável `minhaVariavel` não precisa ser do mesmo tipo retornado por `MinhaFuncao`, mas sim de um tipo que tenha compatibilidade de atribuição (por exemplo, `minhaVariavel` poderia ser **real** no exemplo acima).

Como um exemplo mais concreto de função considere a função seguinte que calcula o fatorial (`n!`) de um número `n` (lembre-se que o fatorial de um número inteiro `n` ≥ `0` é definido como: `n! = n(n-1)(n-2)...1`, se `n > 0`, e `0! = 1`).

```
function Fatorial(n : integer) : integer;

var i, F : integer;

begin
    F := 1;
    if  (n > 1)  then
       for  i := 2  to  n do
          F := F * i;
    Fatorial := F
end;
```

A chamada `x := Fatorial(3)` da função acima resultaria na atribuição do valor `6` à variável `x` (verifique isso!).

3.17 Escopo de Identificadores

Normalmente, um procedimento é declarado na seção de declarações de um programa. No entanto, um procedimento pode ainda ser **aninhado** dentro de outro procedimento (i.e., contido dentro de outro procedimento). Cada procedimento em um ninho de procedimentos tem sua própria seção de declarações e um corpo; o programa principal

também possui uma seção de declarações e um corpo. Uma seção de declarações mais um corpo de programa ou procedimento é denominado de **bloco**. Um **identificador local** é aquele que só pode ser referenciado dentro do bloco no qual o mesmo é definido. Normalmente, um procedimento referencia apenas identificadores locais, mas é possível referenciar identificadores que não são declarados localmente.

O **escopo de um identificador** é o bloco no qual ele é declarado. Um identificador em Pascal pode ser referenciado apenas dentro de seu escopo (isto é denominado **regra de escopo**). A implicação desta regra de escopo é que, no caso de um procedimento aninhado noutro procedimento ou no programa principal, qualquer identificador declarado no bloco externo ao procedimento tem validade (i.e., pode ser referenciado) dentro do procedimento. O exemplo a seguir ilustra este ponto. Suponha que `P1`, `P2` e `P3` são procedimentos:

```
program MeuPrograma;
var  x, y, z : real;

procedure  P1;
var  x1, y1 : integer;

     procedure P2;
     var  x2, y2 : real;

     begin
         (* corpo do procedimento P2 *)
     end;

begin
    (* corpo do procedimento P1 *)
end;

begin
    (* corpo do programa *)
end;
```

Neste exemplo, as variáveis `x`, `y` e `z` declaradas no programa principal podem ser referenciadas no programa principal (obviamente) e nos procedimentos `P1` e `P2`, pois estes procedimentos estão dentro do escopo destas variáveis (i.e., `P1` e `P2` estão dentro do bloco no qual `x`, `y` e `z` foram definidas). As variáveis `x1` e `y1` podem ser

referenciadas nos procedimentos `P1` e `P2`, mas não no programa principal, pois este último não está nos escopos de `x1` e `y1`. Finalmente, as variáveis `x2` e `y2` podem apenas ser referenciadas no procedimento `P2`.

Note que, de acordo com a regra de escopo de Pascal, qualquer identificador (tal como `x`, `y` ou `z` no último exemplo) declarado no programa principal pode ser referenciado em qualquer parte do programa. Por esta razão, variáveis declaradas no programa principal são denominadas de **variáveis globais**. Embora variáveis globais possam ser referenciadas em quaisquer procedimentos em um programa, a referência a variáveis globais dentro de um procedimento não é considerada boa prática de programação. Portanto, é preferível que um procedimento utilize apenas variáveis declaradas localmente e aquelas na lista de parâmetros formais.

É importante notar que um mesmo identificador pode ser declarado em mais de um lugar. Por exemplo, uma variável x pode ser declarada no programa principal (sendo, portanto, uma variável global) e em um procedimento, como ilustrado a seguir:

```
program MeuPrograma;
var x, y, z : real;

procedure P1;
var x : integer;

begin
    .
    :
    x := x + 1
end;

begin

    (* corpo do programa *)

end;
```

A pergunta que surge na situação ilustrada acima é: A que `x` a expressão "`x := x + 1`" faz referência? Ou, em outras palavras, como Pascal decide esta colisão de identificadores? Afinal de contas, o escopo de `x` do programa principal inclui o corpo do procedimento `P1` e, evidentemente, o escopo de `x` declarado no procedimento `P1` também inclui o corpo de `P1`. A resposta a esta questão é que, quando encontra uma referência a um identificador, o compilador Pascal verifica se este identificador é declarado no menor bloco que o contém; se o compilador encontra tal declaração, a mesma será considerada. Caso contrário, se tal declaração não for encontrada neste

bloco mais interno, Pascal procura uma declaração deste identificador no bloco externo imediatamente abrangendo o bloco no qual o identificador foi encontrado. Se a declaração procurada for encontrada, ela será considerada; caso contrário, parte-se para o bloco externo seguinte, e assim por diante até que se chega ao programa principal[14]. Voltando ao exemplo anterior, a variável `x` utilizada na expressão "`x := x + 1`" no procedimento `P1` refere-se à variável `x` declarada neste procedimento (na declaração **`var`** `x : ` **`integer`**) e não àquela declarada no programa principal (na declaração **`var`** `x, y, z : ` **`real`**).

Uma vez que nomes de procedimentos são também identificadores, a regra de escopo em Pascal especifica aonde um procedimento pode ser chamado. Considere o seguinte exemplo:

```
program  MeuPrograma;
var  x, y, z : real;

procedure  P1;
var  x1, y1 : integer;

     procedure  P2;

     var  x2, y2 : real;

     begin
         (* corpo do procedimento P2 *)
     end;

begin
    (* corpo do procedimento P1 *)
end;

procedure P3;
var z : real;

begin
    (* corpo do procedimento P3 *)
end;

begin
    (* corpo do programa *)
end;
```

[14] Naturalmente, se não é encontrada nenhuma declaração do identificador quando a busca chega ao programa principal, haverá um erro de sintaxe do tipo "Identificador desconhecido".

Neste exemplo, o procedimento `P1` pode ser chamado em qualquer lugar, visto que ele é declarado no programa principal e portanto é um identificador global. O procedimento `P2` pode ser chamado apenas pelo procedimento `P1`. O procedimento `P3` pode ser chamado em qualquer ponto do programa exceto nos procedimentos `P1` e `P2`. Na realidade, este último fato não é decorrente da regra de escopo de Pascal, mas deve-se ao fato de, em Pascal, um identificador não poder ser referenciado antes de ser declarado. Se quiséssemos que o procedimento `P3` fosse referenciado por `P1` e `P2`, teríamos que declará-lo antes do procedimento `P1` no programa principal.

3.18 Tipos de Dados Definidos pelo Programador

Além de prover tipos de dados predefinidos, como os vistos anteriormente, Pascal também permite que o programador defina seus próprios tipos de dados. Isto é feito através de **declarações de tipos**. Uma declaração de tipo em Pascal deve preceder a seção de declaração de variáveis num programa (ou procedimento) e tem a seguinte forma:

type <nome do tipo> = <definição de tipo>;

onde <*nome do tipo*> é um identificador que será associado com o tipo de dado sendo definido e <*definição de tipo*> é uma definição de tipo que depende do tipo sendo definido. Nesta seção três tipos de dados definidos pelo programador - chamados **enumerações**, **intervalos** e **conjuntos** - serão vistos.

3.18.1 Enumerações

Um tipo enumeração em Pascal provê uma maneira de se especificar uma lista ordenada de constantes. Cada valor constante é definido como um identificador e a lista é delimitada por parênteses. Em outras palavras, uma declaração de tipo enumeração tem a seguinte forma:

type <nome do tipo> = (<lista de identificadores>);

Exemplos de declarações de enumerações[15]:

```
type
  tCores = (AZUL, VERMELHO, AMARELO, VERDE);
  tMeses = (JAN, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SEP, OUT, NOV,
            DEZ);
```

Uma variável declarada como sendo de um tipo enumeração pode apenas assumir os valores declarados na lista de constantes do tipo. Assim, por exemplo, se a variável `corFavorita` fosse definida como:

```
var corFavorita : tCores;
```

ela poderia assumir apenas os valores que constam na lista de definição do tipo cores. Qualquer tentativa de atribuição de outro valor diferente daqueles resultaria num erro.

Observe que enumerações são tipos ordinais, visto que cada valor tem um sucessor e um antecessor (exceto, é claro, o primeiro, que não tem antecessor, e o último, que não tem sucessor). Por exemplo, na definição do tipo cores no exemplo acima, **`succ`**`(AZUL) = VERMELHO`, **`pred`**`(VERMELHO) = AZUL`, etc. A ordenação de um tipo enumeração começa com 0, o próximo elemento tem ordem 1, etc. (por exemplo, **`ord`**`(AZUL) = 0`, **`ord`**`(VERMELHO) = 1`, etc.).

Uma observação importante sobre enumerações é que um dado identificador pode aparecer em apenas uma lista de definição de um tipo enumeração; a presença de um mesmo identificador em duas listas de definições de enumerações resulta em erro. Por exemplo, as definições de enumerações abaixo seriam ilegais porque os identificadores `JAN`, `FEV`, `DEZ` aparecem em mais de uma lista de identificadores.

```
type
   tMeses = (JAN, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SEP, OUT, NOV,
             DEZ);
   tVerao = (DEZ, JAN, FEV);
```

As únicas operações que podem ser efetuadas com objetos do tipo enumeração (i.e., variáveis e constantes deste tipo) são atribuição e comparação com o uso de operadores relacionais. Não se pode utilizar procedimentos de entrada e saída (por exemplo, **read** ou **write**) com variáveis de um tipo enumeração.

---

[15] Sugere-se, como nos exemplos a seguir, que se comece o identificador de um tipo definido pelo usuário pela letra “t”.

3.18.2 Intervalos

Um intervalo é outro tipo de dados que pode ser declarado pelo programador. Um intervalo consiste de um conjunto de valores associados com um tipo ordinal (**integer**, **boolean**, **char** ou enumeração). Intervalos são utilizados com duas finalidades principais: (1) tornar o programa mais legível; e (2) permitir que o Pascal verifique quando uma variável assume um valor que não é conveniente para o programa (i.e., para limitar os valores que uma variável pode assumir).

Uma declaração de tipo intervalo tem a seguinte forma:

type <nome do tipo> = <valor mínimo> .. <valor máximo>;

Nesta declaração, <*valor mínimo*> e <*valor máximo*> são constantes de um mesmo tipo ordinal (o tipo-base do intervalo) e **ord**(<*valor mínimo*>) deve ser menor do que **ord**(<*valor máximo*>). Uma variável desse tipo pode assumir qualquer valor entre <*valor mínimo*> e <*valor máximo*> (inclusive), mas a tentativa de atribuição de um valor fora deste intervalo acarretará em erro em tempo de execução do programa.

Exemplos de declarações de enumerações:

```
type
   tMaiusculas = 'A' .. 'Z';        (* O tipo base é char *)
   tDiaDaSemana = (DOMINGO, SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA,
                   SABADO)
   tDiaUtil = SEGUNDA .. SEXTA      (* O tipo base é tDiaDaSemana *)
   tDiaDoMes = 1 .. 31              (* O tipo base é integer *)
```

Uma variável de um tipo intervalo herda todas as operações permitidas para o tipo-base do intervalo. Em outras palavras, todas as operações permitidas para o tipo-base de uma intervalo são também permitidas para a próprio tipo intervalo. Por exemplo, pode-se executar qualquer operação válida para inteiros sobre uma variável do tipo `tDiaDoMes` definido no exemplo acima, pois o tipo-base deste tipo é **integer**.

Se um programador estivesse, por exemplo, construindo um programa que manipulasse dias do mês e quisesse certificar-se de que uma variável representando dias do mês não assumisse valores inadmissíveis (por exemplo, `50`), ele poderia declarar esta variável como sendo do tipo `tDiaDoMes`. Desta forma, se essa variável eventualmente assumisse um valor estranho durante a execução do programa, o erro seria facilmente

detectado, pois o programa pararia a execução e acusaria o erro. O mesmo não aconteceria se esta variável fosse declarada como sendo do tipo inteiro[16].

3.18.3 Conjuntos

Um conjunto em Pascal tem a seguinte notação:

[<lista de elementos>]

Onde <*lista de elementos*> é uma lista de constantes de um mesmo tipo ordinal separadas por vírgulas. Um grupo de elementos consecutivos pode ser representado utilizando-se a notação de intervalos (<*valor mínimo*> .. <*valor máximo*>) vista acima. Por exemplo,

```
[0, 4..9, 15]
```

representa um conjunto, cujo tipo-base é integer, contendo os inteiros `0`, `4`, `5`, `6`, `7`, `8`, `9` e `15`. Um conjunto vazio (i.e., um conjunto sem elementos) é representado por `[]`.

Uma declaração de um tipo conjunto em Pascal tem o seguinte formato:

type <nome do tipo> = **set of** <tipo-base>;

onde <*tipo-base*> deve ser um tipo ordinal. Uma variável do tipo conjunto deve ser um conjunto cujos elementos são do mesmo tipo-base do tipo conjunto da variável. Em Turbo Pascal, o número máximo de elementos em um conjunto é 256.

---

[16] Turbo Pascal não verifica automaticamente se uma variável do tipo intervalo assume valores fora do intervalo. Em outras palavras, o comportamento de verificação descrito acima não é padrão em Turbo Pascal. Para que haja verificação de intervalos é necessário que o Turbo Pascal seja instruído para assim proceder explicitamente através de uma diretiva de compilador. Se você quiser que o Turbo Pascal verifique se suas variáveis de tipo intervalo estão realmente dentro do intervalo, você deve utilizar a diretiva `{$R+}` no início do programa. Caso contrário, declarações de tipo intervalo terão pouca utilidade.

Operações sobre conjuntos

Uma **atribuição de conjunto** consiste de uma variável de um tipo conjunto na esquerda, um conjunto escrito na notação vista acima (ou uma outra variável/expressão de tipo conjunto) e um operador convencional de atribuição (":=") entre os mesmos. Por exemplo:

```
type tDigito = set of  1..9

var  par, impar :  tDigito

begin
    .
    :
    par := [2, 4, 6, 8];
    impar := [1, 3, 5, 7, 9];
    .
    :
end
```

As operações de união, interseção e diferença entre conjuntos são definidas da seguinte forma:

• **União** de dois conjuntos A e B: o conjunto dos elementos que estão contidos em A ou em B, ou em ambos. A união é representada pelo operador "+". Por exemplo:

```
meuConjunto := [1, 4, 7] + [2, 4, 8]
```

resultaria na atribuição do conjunto `[1, 2, 4, 7, 8]` à variável `meuConjunto`.

• **Interseção** de dois conjuntos A e B: o conjunto dos elementos que estão contidos simultaneamente em A e em B. A interseção é representada pelo operador "*". Por exemplo:

```
meuConjunto := [1, 4, 7] * [2, 4, 8]
```

resultaria na atribuição do conjunto `[4]` à variável `meuConjunto`.

• **Diferença** de dois conjuntos A e B: o conjunto dos elementos que estão contidos em A mas não em B. A diferença é representada pelo operador "-". Por exemplo:

```
meuConjunto := [1, 4, 7] - [2, 4, 8]
```

resultaria na atribuição do conjunto `[1, 7]` à variável `meuConjunto`.

O **operador de pertinência** "**in**" tem como argumentos um objeto (constante, variável ou expressão) à esquerda e um conjunto à direita, e resulta em verdadeiro (**true**) se o dado objeto pertence ao dado conjunto; caso contrário, resulta em falso (**false**). O objeto cuja pertinência está sendo testada deve ser do mesmo tipo-base do conjunto. Por exemplo, `4 in [1, 4, 7]` resulta em verdadeiro, enquanto que `2 in [1, 4, 7]` resulta em falso.

Conjuntos podem ainda ser comparados para verificar-se igualdade, desigualdade ou continência através dos operadores relacionais =, <>, <=, >=, que têm, no corrente contexto, significados diferentes daqueles vistos anteriormente.

• **Igualdade**: Dois conjuntos são iguais quando eles contêm os mesmos elementos (não importa a ordem dos elementos). Pode-se testar a igualdade de dois conjuntos através do operador relacional "=", que resulta em verdadeiro quando os dois conjuntos sendo testados são iguais e falso se eles não são iguais. Por exemplo, a expressão `[1, 2] = [2, 1]` resultaria em verdadeiro; enquanto que a expressão `[0] = []` resultaria em falso.

• **Desigualdade**: Dois conjuntos são diferentes quando eles não são iguais. Pode-se testar a desigualdade de dois conjuntos através do operador relacional "<>", que resulta em verdadeiro quando os dois conjuntos sendo testados são diferentes e falso se eles são iguais. Por exemplo, a expressão `[0] <> []` resultaria em verdadeiro; enquanto que a expressão `[1, 2] <> [2, 1]` resultaria em falso.

• **Subconjunto**: Um conjunto A é subconjunto de B quando todos os elementos de A são também elementos de B. Pode-se verificar se um conjunto é subconjunto de outro com o uso do operador "<=", tal que `A <= B` resulta em verdadeiro quando A é subconjunto de B e resulta em falso em caso contrário. Deve-se notar ainda que o conjunto vazio ("`[]`") é subconjunto de qualquer conjunto. Por exemplo, `[1,3] <= [0,1,3,6]` e

`[] <= [0, 1]` resultam em verdadeiro, mas `[1, 3] <= [0, 1]` resulta em falso.

• **Superconjunto**: Um conjunto A é superconjunto de B quando A contém todos os elementos de B (em outras palavras, quando B é subconjunto de A). Pode-se verificar se um conjunto é superconjunto de outro com o uso do operador “>=”, tal que `A >= B` resulta em verdadeiro quando A é superconjunto de B e resulta em falso em caso contrário. Por exemplo, `[0, 1, 3, 6] >= [1, 3]` resulta em verdadeiro, mas `[1, 3] >= [0, 1]` resulta em falso.

3.19 Testando e Depurando um Programa

Normalmente, um programa recém-criado apresenta defeitos. Existem três tipos de programas defeituosos:

• **Programas contendo erros sintáticos.** Programas contendo erros sintáticos não podem ser compilados, pois não estão de acordo com as regras para escrita de programas em Pascal. Este tipo de erro é indicado pelo compilador Pascal em *tempo de compilação*.

• **Programas que apresentam erros em tempo de execução.** Um programa contendo um erro deste tipo é compilado sem problemas, mas pode apresentar problemas que interrompem a execução do mesmo.

• **Programas que apresentam erros lógicos.** Um programa contendo um erro deste tipo é compilado e executado sem problemas (i.e., sem interrupções), mas seu funcionamento é errático (i.e., ele não faz o que deveria fazer ou faz aquilo que não deveria fazer).

A enumeração acima está em ordem usual de dificuldade que o programador tem para identificar e corrigir os erros (i.e., erros sintáticos são relativamente fáceis de ser corrigidos, enquanto que erros lógicos às vezes levam anos para ser corrigidos). Os erros dos segundo e terceiro tipos na enumeração acima são comumente conhecidos em

programação como ***bugs***. Estes erros serão melhor descritos e exemplificados nas subseções seguintes.

3.19.1 Erros Sintáticos

Erros sintáticos são causados: (1) **por distração** por parte do programador, ou (2) **por falta de entendimento** adequado das construções sintaticamente válidas em Pascal. Erros sintáticos são, na maioria das vezes, fáceis de serem encontrados e corrigidos, pois, normalmente, quando existe um erro sintático, o compilador indica a instrução e o tipo de erro cometido. Infelizmente, mesmo bons compiladores nem sempre são capazes de fazer essa indicação de erro com precisão. Por exemplo, um compilador pode indicar um erro algumas instruções adiante da instrução onde realmente o erro está localizado, ou pode indicar a existência de um erro diferente do erro real. Qualquer compilador, entretanto, deve ser capaz de indicar a *existência do erro* quando este existe de fato. Erros sintáticos são muito freqüentes em programas implementados por programadores inexperientes com a linguagem, mas a freqüência de ocorrência deste tipo de erro deve diminuir à medida em que o programador torna-se mais experientes. Um exemplo de erro sintático foi apresentado no programa da *Seção 3.13* (ausência de um ponto-e-vírgula no final de uma instrução).

3.19.2 Erros de Execução

Erros de execução (***run-time errors***) são erros em programas que podem ser detectados pelo computador apenas quando o programa é executado. Por exemplo, considere o seguinte programa que se propõe a calcular a média aritmética de um conjunto de valores reais introduzidos pelo usuário. O programa assume que a final da entrada de dados é indicado pelo valor `0`.

```
program  ExemploDeErroDeExecucao;

var
   numero, media  :  real;
   n  :  integer;

begin
    media := 0.0;
    n := 0;
    write('Introduza um numero (zero para terminar): ');
    read(numero);

    while  (numero <> 0)  do  begin
       media := media + numero; (* Acumula a soma dos numeros introduzidos*)
       n := n + 1;  (* Acumula o numero de valores introduzidos *)
       write('Introduza um numero (zero para terminar): ');
       read(numero)
    end; (* while *);

    media := media/n;  (* Calcula a media dos numeros introduzidos *)
    write('A media eh: ', media)
end.
```

O problema com o programa acima vem à tona quando o usuário introduz `0` (zero) como dado para a primeira instrução **read**. Quando isto ocorre, o corpo do laço **while** não é executado e o controle passa para a instrução `media := media/n`, que calcula a média. Acontece que neste caso, `n` vale `0` e, neste ponto, se estaria tentando executar uma divisão aonde o dividendo é `0`, o que é ilegal. Portanto, quando tenta executar esta instrução, o computador emite a mensagem de erro:

```
Error 200: Division by zero.
```

Esta mensagem de erro indica que o programa tentou executar uma divisão ilegal. Existem várias formas de se consertar o erro no programa acima. Uma maneira de fazer isto é substituir as duas últimas instruções pela instrução **if** a seguir[17]:

```
if  n <> 0  then
   begin
      media := media/n; (* Calcula a media dos numeros introduzidos *)
      write('A media eh: ', media)
   end
else
   write('A media eh: 0')
```

[17] É sempre recomendável utilizar uma instrução **if** deste tipo para testar se o divisor de uma divisão é diferente de `0`.

Note que um erro deste tipo não pode ser apontado pelo compilador, uma vez que o programa está sintaticamente correto (i.e., ele não viola nenhuma regra de construção de programas em Pascal). É interessante observar ainda que alguns erros de execução aparecem já durante a primeira execução do programa, enquanto outros, como o do exemplo acima, podem demorar um pouco para aparecerem.

3.19.3 Erros Lógicos

Considere o seguinte programa que se propõe (novamente) a calcular a média aritmética de um conjunto de valores reais introduzidos pelo usuário. O programa assume que o final da entrada de dados é indicado pelo valor `0` e é uma variação daquele apresentado na subseção anterior.

```
program  ExemploDeErroDeLogico;

var
   numero, media  :  real;
   n  :  integer;

begin
    media := 0.0;
    n := 0;

    repeat
         write('Introduza um numero (zero para terminar): ');
         read(numero);
         n := n + 1;  (* Acumula o numero de valores introduzidos *)
         media := media + numero;  (* Acumula a soma dos numeros *)
    until numero = 0;

    if (numero <> 0) then
         media := media/n; (* Calcula a media dos numeros introduzidos *)
    writeln('A media eh: ', media :7:2)
end.
```

O programa acima não contém nem erros sintáticos nem erros de execução (verifique isso!). O problema com o programa acima é simplesmente que ele não faz o que deveria fazer; isto é, esperava-se que o programa acima fosse capaz de calcular a média aritmética dos números introduzidos, mas, na realidade, ele não faz este cálculo corretamente. Por isso, o programa acima é dito conter um **erro de lógica** (outro tipo de ***bug***). Para verificar isto, execute o programa acima e utilize os valores `4.0`, `6.0`, `5.0` e

`0.0` como entrada. Obviamente, o valor esperado para a média destes valores seria `5.0`; entretanto, o programa produz como saída:

```
A media eh: 3.75
```

O programa acima está errado porque a própria lógica (i.e., o *raciocínio*) utilizada para construir o algoritmo que resolve o problema está errada. Portanto, consertar o programa acima envolve rescrever o algoritmo e, então, reeditar o programa. Uma forma de correção resultaria no seguinte corpo de programa[18]:

```
begin
    media := 0.0;
    n := 0;

    repeat
         write('Introduza um numero (zero para terminar): ');
         read(numero);
         if (numero <> 0) then
            n := n + 1; (* Acumula o numero de valores introduzidos *)
         media := media + numero;  (* Acumula a soma dos numeros *)
    until numero = 0;

    if  n <> 0  then
        begin
          media := media/n; (* Calcula a media dos numeros introduzidos *)
            write('A media eh: ', media :7:2)
        end
      else
         write('A media eh: 0')
end.
```

3.19.4 Testando Manualmente um Programa

Bugs são muito freqüentes em programação e, virtualmente, qualquer programa de dimensão razoável possui *bugs*. No momento, não existe nenhum método prático para garantir que um dado programa não contenha *bugs*[19]. Entretanto, a presença destes erros pode ser consideravelmente reduzida através da execução de testes rigorosos com o programa. Uma discussão completa das técnicas de **verificação de software** está além do escopo deste livro. Mas, para os programas requeridos num curso introdutório de programação, simples **testes manuais** (***desk checking***) são suficientes. Um teste manual de um programa consiste em dois passos:

---

[18] Note que a mudança introduzida no programa foi a inclusão de uma instrução **if** que evita que o número de valores introduzidos, `n`, seja acrescido quando o número introduzido for `0`.

[19] Evidentemente, isto não se aplica a programas triviais como os apresentados aqui.

1. **Leitura** (ou **inspeção**) **do programa** na tentativa de responder a uma **lista de verificação** contendo questões referentes a erros comuns de programação (por exemplo, *Cada variável possui um valor atribuído antes de ser referenciada no programa?*). Uma lista de verificação (incompleta) é apresentada no *Apêndice C*. É importante que você acrescente seus próprios itens a esta lista e enfatize questões referentes a erros que você comete com mais freqüência.

2. **Execução manual do programa** com caso de entrada *qualitativamente diferentes*. Estes casos de entrada devem envolver não apenas *entradas válidas*, mas também *entrada inválidas*. Por exemplo, um programa que calcula o fatorial de um número inteiro deve ser testado não apenas com números não-negativos (válidos), mas também com números negativos (inválidos).

Idealmente, o teste de um programa deve ser realizado por uma pessoa que não seja o próprio programador. A razão para isso é de natureza psicológica: a construção de um programa é uma atividade *construtiva*, enquanto que os testes são de natureza *destrutiva*. Isto é, o objetivo de um teste é o de encontrar erros de modo a invalidar o programa. Em cursos de programação uma atividade saudável neste sentido é a troca de programas entre os alunos, de modo que um aluno testa o programa do outro (sem *falso coleguismo!*).

3.20 Exercícios de Revisão

1. Em que consiste um programa em Pascal?

2. Diga por que cada um dos identificadores abaixo são inválidos em Pascal.
   (a) minha-variavel
   (b) 1a
   (c) variavel(a)

3. Por que e quando é mais aconselhável o uso de uma constante declarada num programa ao invés do uso do valor da constante em si (por exemplo, o uso da constante "`PI`" é geralmente melhor do que "`3.14`")?

4. Para que servem as declarações de variáveis em Pascal?

5. O que é um tipo de dados ordinal e quais são os tipos de dados ordinais em Pascal?

6. Qual é a diferença entre os símbolos “=” e “:=” em Pascal?

7. O que é uma diretiva de compilador Pascal?

8. Para que servem comentários em um programa Pascal?

9. O que é uma estrutura de controle? Quais são as estruturas de controle de Pascal?

10. Quais são as diferenças principais entre procedimento e função em Pascal?

11. Compare parâmetro formal com parâmetro real em Pascal.

12. Descreva as regras de passagem de parâmetros em Pascal.

13. O que são (a) parâmetro de valor, e (b) parâmetro variável.

14. Descreva:
    (a) Parâmetro de entrada;
    (b) Parâmetro de saída; e
    (c) Parâmetro de entrada/saída;

15. Descreva os seguintes conceitos em Pascal:
    (a) Bloco;
    (b) Escopo de um identificador;
    (c) Regra de escopo

3.21 Exercícios de Programação

EP3.1) Escreva um procedimento em Pascal que troca os valores de dois números inteiros ou reais. É possível utilizar este mesmo procedimento para trocar os valores de duas variáveis do tipo **char**? Explique.

EP3.2) Escreva um procedimento em Pascal que receba como entrada os coeficientes reais de uma equação do 2º grau e retorne as raízes desta equação. Em seguida, escreva um programa em Pascal que leia os valores destes coeficientes no meio de entrada, chame este procedimento, e imprima os resultados.

EP3.3) Uma série de Fibonacci é constituída por uma seqüência de números naturais, cujos dois primeiros termos são iguais a 1, e tal que cada número (exceto os dois primeiros) na seqüência é igual a soma de seus dois mais próximos antecedentes. Escreva uma função que retorne o n-ésimo termo da série de Fibonacci.

EP3.4) A implicação lógica ("=>") tem a seguinte tabela-verdade:

Operando1	**Operando2**	**Operando1 ⇒ Operando2**
verdadeiro	verdadeiro	verdadeiro
verdadeiro	falso	falso
falso	verdadeiro	verdadeiro
falso	falso	verdadeiro

Escreva uma função booleana (i.e., uma função cujo tipo de retorno seja **boolean**) que receba dois operandos lógicos (booleanos), `op1` e `op2`, como entrada e retorne o valor `op1 => op2`.

Capítulo 4

ARRANJOS E CADEIAS DE CARACTERES

4.1 Introdução

Cada variável vista até aqui tem sido associada a um única célula de memória (ou a mais de uma célula de memória mas que pode apenas ser acessada como uma unidade). Essas variáveis são chamadas de **variáveis simples** e os tipos de dados dessas variáveis são **tipos simples**. A partir deste capítulo, serão estudados vários **tipos de dados estruturados** (ou **estruturas de dados**), que são agrupamentos de itens de dados relacionados entre si, tal que cada um desses itens pode ser acessado individualmente. Tipos de dados estruturados em Pascal são definidos pelo programador utilizando declarações de tipos semelhantes àquelas vistas no capítulo anterior.

O primeiro tipo de dados estruturado a ser visto será o **arranjo**. Um arranjo é uma estrutura de dados utilizada para armazenar uma coleção de dados que são todos do mesmo tipo (por causa disto, o arranjo é considerado uma estrutura de dados **homogênea**). Em outras palavras, uma variável do tipo arranjo refere-se a toda uma coleção de dados, mas existe também uma forma de se acessar cada um dos itens dessa coleção.

4.2 Arranjos Unidimensionais

Arranjos são análogos às matrizes em Matemática. Em particular, os **arranjos unidimensionais** são análogos às matrizes de uma única linha (ou vetores). Os arranjos unidimensionais serão vistos primeiro e, depois, será feita uma generalização para arranjos de mais de uma dimensão.

Um arranjo é declarado através de uma **declaração de tipo arranjo**, que tem o seguinte formato:

type <nome do tipo arranjo> **= array** [<tipo do subscrito>] **of** <tipo do elemento>

onde, <*nome do tipo arranjo*> é um identificador que dá nome ao tipo arranjo sendo declarado; <*tipo do subscrito*> é o tipo do índice (ou subscrito) que será utilizado para acessar cada elemento do arranjo; <*tipo do elemento*> é o tipo de cada elemento do arranjo. O tipo do subscrito pode ser **boolean**, **char**, enumeração ou intervalo, embora o tipo mais comum seja um intervalo da forma `min .. max`, onde `min` e `max` são inteiros e `min < max`. Existe um elemento do arranjo para cada valor possível no <*tipo do subscrito*> (por exemplo, se o tipo do subscrito fosse definido como `1 .. 10`, haveria dez elementos num arranjo desse tipo). Observe que não existe restrição quanto ao <*tipo do elemento*>; isto é, pode-se construir arranjos com elementos de quaisquer tipos.

Considere o seguinte exemplo:

```
type
   tArranjoReal = array [1 .. 6] of real;

var
   meuArranjo : tArranjoReal;
```

Quando encontra a declaração da variável `meuArranjo`, Pascal aloca seis posições de memória (o número de subscritos na declaração de tipo de `tArranjoReal`), cada uma capaz de conter um elemento do tipo **real** (que é o tipo de elemento na declaração de tipo de `tArranjoReal`). A variável `meuArranjo` refere-se a toda a coleção de seis elementos reais. O acesso a cada um desses elementos é feita através de **índices** (ou **subscritos**) como será visto a seguir.

Um subscrito de arranjo é utilizado para distinguir elementos de um mesmo arranjo e, assim, possibilitar o acesso individual de elementos. Uma **referência** a um elemento de um arranjo é feita utilizando-se a seguinte notação:

<nome da variável do tipo arranjo>[<subscrito>]

onde <*subscrito*> deve ser uma constante, variável ou expressão, cujo tipo deve ser compatível com o tipo de subscrito na declaração do tipo arranjo (caso contrário, haverá um **erro em tempo de compilação**) e cujo valor no instante da referência deve ser um dos valores admissíveis para o tipo de subscrito na declaração do tipo arranjo (caso contrário, haverá um **erro em tempo de execução do programa**). Por exemplo, considerando as declarações do exemplo anterior no trecho de programa a seguir:

```
meuArranjo[3] := 5.2;            (* 1 *)
meuArranjo[2*1 + 3] := 10;       (* 2 *)
meuArranjo[10] := 3.14;          (* 3 *)
meuArranjo[2.5 + 1.5] := 3.14;   (* 4 *)
```

as referências `(* 1 *)` e `(* 2 *)` ao arranjo `meuArranjo` são perfeitamente legais, mas as referências `(* 3 *)` e `(* 4 *)` não o são. A referência `meuArranjo[10]` é ilegal porque 10 está fora do intervalo do subscrito, enquanto que `meuArranjo[2.5 + 1.5]` é ilegal porque a expressão entre colchetes é real, e portanto, diferente do tipo do subscrito[20].

Referências a elementos de arranjos não são limitadas apenas a atribuições (como no exemplo anterior). Ou seja, pode-se utilizar uma referência a um elemento de arranjo em qualquer local de um programa aonde pode-se colocar uma variável comum (por exemplo, numa expressão ou numa lista de parâmetros reais numa chamada de procedimento).

Freqüentemente, deseja-se manipular todos os elementos de um arranjo em seqüência (por exemplo, pode-se querer atribuir um valor inicial a cada elemento de um arranjo). Em arranjos com subscritos inteiros (os mais comuns), isso pode ser implementado através de instruções **for**. Numa tal instrução **for**, a variável de controle corresponde ao índice (subscrito) do arranjo. Por exemplo, suponha que se desejasse atribuir o valor `0.0` a cada elemento do arranjo `meuArranjo` do exemplo acima. Isso poderia ser feito através da instrução **for** abaixo (supondo que a variável i tenha sido declarado como sendo do tipo **integer**):

```
for i := 1 to 6 do
   meuArranjo[i] := 0.0;
```

Esse último exemplo pode ser facilmente entendido: quando a variável i é igual a `1`, ao primeiro elemento do arranjo (referenciado por `meuArranjo[1]`) é atribuído o valor `0.0`; quando i é igual a `2`, ao segundo elemento do arranjo (referenciado por `meuArranjo[2]`) é também atribuído o valor `0.0`, e assim por diante até que a variável i assume o valor `6` (lembre-se que quando i assume o valor `7` o laço **for** é encerrado).

[20] Turbo Pascal não checa automaticamente se um dado índice está dentro do intervalo de valores permitidos para o subscrito. Para que essa verificação seja efetuada, deve-se instruir explicitamente o compilador através da diretiva `{$R+}`.

O exemplo seguinte é mais completo e consiste de um programa utilizado para calcular a média de uma seqüência de números reais introduzidos pelo usuário:

```
program CalculoDeMedia;
{Calcula  e apresenta a média de uma seqüência de números reais}

const MAX_NUMERO_ITENS = 10;

type   tArranjoReal = array [1 .. MAX_NUMERO_ITENS] of real;

var
   meuArranjo : tArranjoReal;
   media, soma : real;
   i : integer;

begin
writeln('Introduza ', MAX_NUMERO_ITENS:2,' números(um em cada linha):');
  for i := 1 to MAX_NUMERO_ITENS do {Lê os dados e ...}
      readln(meuArranjo[i]);      { ... os atribui aos elementos do arranjo}

  media := 0.0;     {Inicializa a variável que vai conter a média com zero}
  for i := 1 to MAX_NUMERO_ITENS do {Acumula soma dos elementos em media}
      media := media + meuArranjo[i];

       {Neste ponto, media contém a soma de todos os elementos do arranjo}

  media := media/MAX_NUMERO_ITENS; {A média é realmente calculada aqui!}

  writeln;
  writeln('A média é: ', media:8:2); {Apresenta resultado para o usuário}
end.
```

Note que no exemplo acima foi utilizada uma constante simbólica (`MAX_NUMERO_ITENS`) na declaração do tipo `tArranjoReal` e em outros pontos do programa. Conforme já foi discutido antes, essa constante tem dois objetivos principais: (1) *melhorar a legibilidade* do programa, e (2) *facilitar a modificação* do mesmo. Note que Pascal não permite o uso de variáveis na definição de subscritos de arranjos (i.e., não se poderia, por exemplo, utilizar uma variável no lugar da constante `MAX_NUMERO_ITENS` na declaração de `tArranjoReal`). Em outras palavras, a dimensão (i.e., o número de elementos) de um arranjo deve ser completamente conhecida em tempo de compilação do programa; essa dimensão permanece imutável durante toda a execução do programa. Algumas linguagens de programação (por exemplo, Algol) permitem o uso de arranjos (chamados **arranjos dinâmicos**) cujas dimensões variam no decorrer do programa, mas Pascal não permite isso.

Quase todas as operações permitidas sobre arranjos referem-se a elementos individuais de arranjos, e não a arranjos como um todo. Em Pascal, a única exceção é a operação de **atribuição de arranjo**, na qual todos os elementos de um arranjo são atribuídos aos elementos de outro arranjo numa única instrução. Neste caso, os arranjos devem ser do

mesmo tipo. Por exemplo, suponha que a variável `meuSegundoArranjo` seja uma variável do tipo `tArranjoReal` (v. exemplo no início desta seção). Então, a seguinte atribuição de arranjo:

```
meuSegundoArranjo := meuArranjo;
```

atribuiria o valor do primeiro elemento de `meuArranjo` ao primeiro elemento de `meuSegundoArranjo`, o valor do segundo elemento de `meuArranjo` ao segundo elemento de `meuSegundoArranjo`, e assim por diante.

Às vezes é necessária a manipulação de arranjos inteiros por procedimentos ou funções. Para uma passagem de parâmetros do tipo arranjo ser bem sucedida, os parâmetros reais e formais correspondentes devem ser exatamente do mesmo tipo de arranjo. As outras regras vistas anteriormente para passagem de parâmetros em Pascal continuam válidas. Por exemplo, o trecho de programa do exemplo anterior responsável pelo cálculo da média poderia ser substituído por uma chamada ao procedimento `CalculaMedia` abaixo que efetua esse cálculo:

```
procedure CalculaMedia(vetor : tArranjoReal; numeroDeItens: integer;
                       var M : real);

var i : integer;

begin
  M := 0.0;  {Inicializa a variável que vai conter a média com zero}
  for i := 1 to numeroDeItens do {Acumula soma dos elementos em M}
     M := M + vetor[i];

  M := M/numeroDeItens     {A média é realmente calculada aqui!}
end;
```

Dado o procedimento acima, o corpo do programa de média poderia ser reescrito como a seguir:

```
begin
writeln('Introduza ',MAX_NUMERO_ITENS:2,' números(um em cada linha): ');
  for i := 1 to MAX_NUMERO_ITENS do {Lê os dados e...}
     readln(meuArranjo[i]); {...os atribui aos elementos do arranjo}
  CalculaMedia(meuArranjo, MAX_NUMERO_ITENS, media);
  writeln;
  writeln('A média é: ', media:8:2)  {Apresenta resultado para o usuário}
end.
```

Exercícios:

1. O procedimento `CalculaMedia` acima poderia ser escrito de forma mais elegante como uma função que retorna a média do arranjo de entrada. Modifique esse procedimento e transforme-o numa função. É possível utilizar o nome `Media` para essa função?

2. Escreva um procedimento (denominado `LeArranjo`) cuja chamada substitui o trecho do programa responsável pela leitura do arranjo. (Note que o arranjo na lista de parâmetros formais neste caso deve ser um parâmetro do tipo variável.)

Com relação ao que acontece durante passagem de parâmetros, Pascal trata parâmetros consistindo de arranjos da mesma forma que ele trata outros tipos de parâmetros vistos anteriormente. Isto significa que um arranjo que é passado como um parâmetro de valor será completamente copiado antes de ser utilizado pelo procedimento. Quando arranjos muito grandes (i.e., arranjos que ocupam muito espaço em memória) são utilizados, pode ser que esta operação (*invisível* para o programador) provoque um alto custo tanto de tempo de processamento quanto de memória. Por isso, nesses casos, programadores experientes utilizam parâmetros variáveis mesmo quando os parâmetros são apenas parâmetros de entrada. Quando um parâmetro deve ser apenas utilizado como entrada (i.e., ele não deve ser modificado pelo procedimento) e utilizam-se parâmetros variáveis por questões de economia e desempenho, cuidados adicionais devem ser tomados para evitar que estes parâmetros não sejam modificados dentro do procedimento, pois qualquer mudança será refletida no parâmetro real fora do procedimento. Essa discussão é válida para qualquer estrutura de dados complexa utilizada como parâmetro, e não apenas para arranjos.

4.3 Arranjos Multidimensionais

Nos exemplos de arranjos vistos até aqui os elementos eram todos de tipos de dados simples. Mas, conforme visto na seção precedente, não existe nenhuma restrição sobre os tipos de dados dos elementos de um arranjo. Isto significa que, em particular, os elementos de um arranjo podem ser também arranjos, cujos elementos podem por sua vez ser arranjos, e assim por diante, de modo que se pode, em princípio, construir estruturas arbitrariamente complexas consistindo de arranjos de arranjos. Um arranjo cujos elementos são arranjos é denominado **arranjo multidimensional**. O caso mais simples de arranjo multidimensional é o arranjo de duas dimensões (ou **bidimensional**). Considere o seguinte exemplo:

```
type
   tArranjo1 = array [1..3]  of   real;
   tArranjo2 = array [1..5]  of   tArranjo1;

var
   meuArranjo2 : tArranjo2;
```

No exemplo acima, `tArranjo2` é um arranjo bidimensional no qual cada elemento é um arranjo do tipo `tArranjo1` que, por sua vez consiste de três elementos reais. Como `tArranjo2` contém cinco elementos do tipo `tArranjo1`, `tArranjo2` contém na realidade 15 (i.e., 5 x 3) elementos reais. Um arranjo do tipo `tArranjo2` pode ser interpretado como sendo análogo a uma matriz (ou tabela) com cinco linhas e três colunas.

Em geral, uma declaração de um tipo arranjo multidimensional em Pascal tem a seguinte forma:

type array [subscrito1] **of array** [subscrito2] ... **of array** [subscritoN] **of** <tipo do elemento>

Existe, no entanto, uma notação mais conveniente para a declaração de arranjos multidimensionais em Pascal. Ou seja, a declaração acima é equivalente a:

type array [subscrito1, subscrito2, ..., subscritoN] **of** <tipo do elemento>

Nas declarações acima, `subscrito1` é o tipo do subscrito (índice) da dimensão 1 do arranjo, `subscrito2` é o tipo do subscrito da dimensão 2, e assim por diante. Evidentemente, as mesmas restrições impostas para índices de arranjos unidimensionais (v. seção anterior) continuam válidas aqui[21]. Usando as duas formas sintáticas de declaração acima, o tipo `tArranjo2` do exemplo anterior poderia ser declarado como:

```
type
    tArranjo2 = array [1..5] of array [1..3]   of   real;
```

ou de modo equivalente:

[21] Note que subscritos de dimensões diferentes podem ser de tipos diferentes.

```
type
    tArranjo2 = array [1..5, 1..3]   of   real;
```

Embora, a priori, não haja limitação no número de dimensões de um arranjo em Pascal, arranjos de mais de três dimensões são raramente usados em situações práticas. Note ainda que referências a elementos de arranjos multidimensionais devem incluir ***todos*** os índices (i.e., todas as dimensões) do arranjo. Assim, uma referência a um elemento de um arranjo bidimensional deve incluir dois índices entre colchetes e separados por vírgula (cada índice deve seguir as condições impostas acima para referência a arranjos unidimensionais). Por exemplo, se a variável `meuArranjo2` fosse declarada como sendo do tipo `tArranjo2` acima, as seguintes referências (atribuições) seriam legais:

```
meuArranjo2[2, 1] :=  3.14;
meuArranjo2[2 + 3*1, 2] :=  0;
```

enquanto que a referência:

```
meuArranjo2[1, 3 + 2*1] :=  1.5;
```

seria ilegal porque a expressão representando o segundo índice `(3 + 2*1)` resulta num valor (`5`) que está fora do intervalo de definição de tipo do segundo índice (`1..3`)[22].

Laços de repetição **for** encadeados podem ser utilizados para acessar todos os elementos de um arranjo multidimensional numa dada ordem. Por exemplo, os seguintes laços for podem ser utilizados para inicializar o arranjo `meuArranjo2` com o valor `0.0` (suponha que i e j foram declarados como **integer** no início do programa):

```
for i := 1 to 5 do
   for j := 1 to 3 do
     meuArranjo2[i, j] := 0.0;
```

A ordem de referência aos elementos especificada pelas instruções **for** acima é a seguinte:

`meuArranjo2[1, 1]` → `meuArranjo2[1, 2]` → `meuArranjo2[1, 3]` → `meuArranjo2[2, 1]` → ... → `meuArranjo2[5, 1]` → `meuArranjo2[5, 2]` → `meuArranjo2[5, 3]`

[22] Deve-se notar ainda que Pascal também aceita a notação de índices entre colchetes e justapostos. Por exemplo, `meuArranjo2[2][1]` significa exatamente a mesma coisa que `meuArranjo2[2,1]`, mas a segunda notação parece ser mais simples e intuitiva, e portanto, é a mais usada.

Esta ordem de referência aos elementos de um arranjo multidimensional, na qual o último índice do arranjo (neste caso particular, o índice representado por `j`) varia enquanto os outros (neste caso, o índice representado por `i`) são mantidos constantes, é denominada **ordenação por coluna**. Outra forma de ordenação seqüencial para arranjos bidimensionais é a **ordenação por linha** como mostra o exemplo abaixo:

```
for j := 1 to 3 do
   for i := 1 to 5 do
     meuArranjo2[i, j] := 0.0;
```

Exercício: Qual é a ordem de referência aos elementos do arranjo `meuArranjo2` especificada pelas instruções **for** deste último exemplo?

4.4 Determinando o Número de Elementos de um Arranjo

Conforme visto acima, o tipo de índice de um arranjo pode ser qualquer tipo ordinal, exceto o tipo **integer**. O menor índice do tipo de índice de um arranjo é denominado **limite inferior** do arranjo; de modo semelhante, o maior índice do tipo de índice é denominado de **limite superior** do arranjo. Sejam i e s os limites inferior e superior de um arranjo unidimensional, respectivamente. Então, o número de elementos do arranjo pode ser determinado através da fórmula:

ord(s) - **ord**(i) + 1

Considere, por exemplo, as seguintes declarações:

```
type
    tFruta = (MANGA, PITOMBA, CAJU, MORANGO)
    tEstacao = (PRIMAVERA, VERAO, OUTONO, INVERNO)
var
    epoca : array [tFruta]  of   tEstacao;
```

Os limites inferior e superior do arranjo `epoca` acima são, respectivamente, `MANGA` e `MORANGO`. Tem-se ainda que **`ord`**`(MANGA) = 0` e **`ord`**`(MORANGO) = 3` (por que?). Portanto, o número de elementos do arranjo `epoca` é: `3 - 0 + 1 = 4`.

A fórmula acima pode facilmente ser generalizada para arranjos multidimensionais, nos quais cada dimensão tem seus limites inferior e superior. Sejam i_j e s_j os limites inferior

e superior da j-ésima dimensão de um arranjo n-dimensional. Então, o número de elementos do arranjo pode ser determinado através da fórmula:

$$[\mathbf{ord}(s_1) - \mathbf{ord}(i_1) + 1] \times [\mathbf{ord}(s_2) - \mathbf{ord}(i_2) + 1] \times ... \times [\mathbf{ord}(s_n) - \mathbf{ord}(i_n) + 1]$$

Utilizando essa última fórmula, verifique que o arranjo `arranjoExemplo` declarado a seguir:

```
var
    arranjoExemplo : array [1..10, 4..6, -1..7] of real;
```

possui 270 elementos do tipo **real**.

Exercício: Quantos elementos possui o arranjo `preco` na declaração abaixo?

```
var
    preco : array [tFruta, tEstacao] of  real;
```

4.5 Cadeias de Caracteres

O tipo de dados **string** representa cadeias de caracteres em Turbo Pascal (e em outras implementações de Pascal) embora não seja especificado em Pascal padrão. Para declarar-se uma variável do tipo string, deve-se especificar o comprimento máximo (i.e., o número máximo de caracteres) da cadeia de caracteres que a variável pode conter. Este número em Turbo Pascal é limitado a 255 (i.e., não se pode ter uma variável do tipo string com mais de 255 caracteres em Turbo Pascal). A declaração de uma variável string em Turbo Pascal segue a seguinte sintaxe:

var <nome da variável> : **string**[<comprimento

Por exemplo,

```
var meuString : string[10]
```

declara uma variável, chamada `meuString`, capaz de conter no máximo 10 caracteres.

Uma cadeia de caracteres constante em Pascal é representada por seu conteúdo entre apóstrofos ('). Por exemplo, `'Bom dia.'` é uma cadeia de caracteres constante em Pascal, cujo comprimento é 8 (o espaço em branco entre "Bom" e "dia" também é levado em consideração).

Apesar da similaridade com arranjos, o tipo de dados string é tratado de forma diferente em Pascal e, de fato, é muito mais fácil manipular cadeias de caracteres como strings do que como arranjos de caracteres em Turbo Pascal. Por exemplo, se a variável `meuArranjo` fosse declarada como:

```
var  meuArranjo : array [1..10]  of  char
```

ela seria capaz de conter 10 caracteres, da mesma forma que a variável `meuString` do exemplo precedente, mas Turbo Pascal trata estas duas variáveis de formas bem diferentes. Suponha que se quisesse ler no meio de entrada um valor (seqüência de caracteres) para estas duas variáveis. No caso da variável `meuString`, seria necessária apenas uma instrução **read** (ou **readln**) como:

```
read(meuString)
```

No caso da variável `meuArranjo`, entretanto, seria necessário um laço **for** para ler um caracter de cada vez (seguido de <ENTER>) no meio de entrada, como visto a seguir:

```
for i := 1 to 10 do
   read(meuArranjo[i]);
```

Existem outras diferenças entre variáveis do tipo string e variáveis do tipo arranjo de caracteres, que serão discutidas mais adiante. Apesar dessas diferenças, pode-se pensar numa variável string como um tipo especial de arranjo de caracteres. Existem duas espécies de dados associados com uma variável string: (1) seu *conteúdo*, e (2) seu *comprimento*. O comprimento de uma variável string é **dinâmico** no sentido de que ele é determinado pelo valor da cadeia de caracteres correntemente armazenada na variável. Este comprimento pode variar de 0 (quando não há nenhuma cadeia de caracteres armazenada na variável) até o valor máximo especificado na declaração da variável. Assim, por exemplo, a variável `meuString` declarada acima poderia conter (em diferentes instantes, obviamente) as seqüências de caracteres: 'Bom dia.' e 'Boa noite.'; no primeiro caso, o comprimento da variável seria 8, enquanto que no segundo caso o comprimento seria 10. A variável `meuString` não poderia armazenar uma cadeia de caracteres com mais de 10 caracteres (como 'Good Morning.') pois o comprimento máximo da variável foi declarado como sendo 10. Em Turbo Pascal, a função predefinida **length** recebe uma variável string como entrada e retorna o valor

correntemente armazenado para o comprimento da variável. Por exemplo, no trecho de programa abaixo,

```
meuString := 'Bom dia.';
write(length(meuString));
```

o valor impresso pela instrução **write** seria 8.

Na realidade, o compilador Turbo Pascal reserva uma posição a mais em memória do que o valor especificado para o comprimento máximo de uma variável string. Por exemplo, a variável `meuString`, cujo comprimento máximo foi especificado como sendo 10 teria 11 posições de memória, cada uma capaz de conter um caracter. A posição adicional, com índice 0, é utilizada por Turbo Pascal para armazenar o comprimento da variável. Como o espaço reservado para conter cada caracter é de 8 bits e o maior inteiro que pode ser armazenado em 8 bits é 255, o maior comprimento permitido para uma variável string é 255. Isto explica a limitação de comprimento máximo encontrada em Turbo Pascal (e em algumas outras implementações de Pascal). Em resumo, pode-se obter o comprimento de uma variável string através da função **length**, como visto acima, ou pode-se, alternativamente, acessar o elemento de índice 0 da variável. Em alguns problemas envolvendo strings, o conhecimento deste último fato é essencial para a resolução dos mesmos. Por exemplo, se você utilizar um laço **for** para copiar uma variável string em outra, caracter por caracter, você vai precisar também atribuir o comprimento de uma variável à outra, o que seria equivalente, neste caso específico, a começar o laço **for** em 0, ao invés de 1.

O **string nulo** é um string cujo comprimento é 0; este string é representado em Pascal por '' (i.e., duas aspas). Pode-se atribuir nulo a uma variável string através de uma atribuição como[23]:

```
meuString := '';
```

Outra semelhança entre arranjos e strings é que elementos de uma variável string também podem ser referenciados individualmente. Por exemplo, a instrução abaixo:

```
meuString[8] := '!';
```

armazena o caracter '!' na oitava posição da variável `meuString`. Portanto, se esta variável contivesse antes desta instrução a seqüência 'Bom dia.', ela passaria a conter

[23] Alternativamente, pode-se *enganar* o compilador Turbo Pascal através da atribuição `meuString[0] := 0`, mas não há nenhuma vantagem em fazer isso.

após esta instrução a seqüência 'Bom dia!' (note que o caracter '.' foi substituído por '!')[24].

Uma importante diferença entre variáveis do tipo string e arranjos de caracteres é que Turbo Pascal permite que strings possam ser utilizados como tipos de retorno de funções, enquanto que arranjos não podem ser utilizados como tipos de retorno de funções. Aliás, strings são os únicos tipos de dados estruturados que podem ser utilizados como tipos de retorno de funções em Pascal.

Exercícios:

1. Escreva um trecho de programa que substitui todos os caracteres correntemente armazenados na variável `meuString` com '*'. (**Sugestão**: use um laço **for** com o limite superior igual a **length**(`meuString`) para acessar todos os elementos da variável e atribua o valor '*' a cada elemento.)

2. Suponha que o tipo `tString20 = string[20]` tenha sido declarado num programa Pascal. Escreva uma função chamada `Inverso` que recebe uma cadeia de caracteres do tipo `tString20` como entrada e retorna a cadeia escrita ao inverso. Por exemplo, se a cadeia de entrada fosse 'roma', a função deveria retornar 'amor'. (**Sugestão**: Primeiro, você deve atribuir à variável string de saída, cujo identificador corresponde ao nome da função, o comprimento da variável de entrada - veja acima como fazer isso. Depois, escreva um laço **for** que atribua o primeiro elemento do string de entrada ao último elemento do string de saída, o segundo elemento do string de entrada ao penúltimo elemento do string de saída, e assim por diante.)

Um fato importante referente à compatibilidade de tipos é que os tipos **char** e **string** são compatíveis (isso se refere a variáveis do tipo string como um todo; a compatibilidade entre elementos de strings e **char**s é óbvia porque os elementos de um string são do tipo **char**). Por exemplo, suponha a existência de uma variável `meuCaracter` do tipo **char**. Então, a atribuição:

```
meuString := meuCaracter;
```

atribuiria à variável `meuString` uma cadeia de caracteres consistindo apenas do caracter armazenado na variável `meuCaracter`. Entretanto, a atribuição:

```
meuCaracter:= meuString;
```

[24] Observe que os elementos de uma variável string são caracteres e não cadeias de caracteres como a própria variável. Isto significa, por exemplo, que uma atribuição como `meuString[8] := '!!!!'` seria ilegal.

faria sentido apenas se, nesse instante, a variável `meuString` contivesse apenas um caracter; caso contrário, a atribuição resultaria em erro.

Os operadores relacionais de Pascal (=, <, >, etc.) podem ser utilizados para comparar strings. Dois strings são iguais quando eles têm o mesmo comprimento e contêm exatamente os mesmos caracteres como elementos. Um string é menor do que outro (ou precede o outro) se, quando eles são comparados elemento a elemento, encontra-se um caracter no primeiro string que precede o correspondente caracter no segundo string de acordo com a ordem de precedência entre caracteres vista na *Seção 3.5.3*; caso contrário, e quando os strings não são iguais, o primeiro string é maior do que o segundo. Note que quando strings contendo apenas letras maiúsculas ou apenas letras minúsculas são comparados, o resultado corresponde à ordem alfabética usual. Por exemplo, `'copo' < 'corpo'` e `'JOSE' < 'JOSUE'` resultam em verdadeiro (**true**). Lembre-se que, como visto na *Seção 3.5.3*, letras maiúsculas precedem as minúsculas, portanto, `'Maria'` precede `'maria'`. Quando todos os caracteres de um string coincidem com os caracteres iniciais de um outro string de comprimento maior, o string de comprimento menor é considerado menor. Assim, por exemplo, `'Carol'` é menor do que `'Carolina'`[25].

Sumário de Diferenças entre Strings e Arranjos:

1. O comprimento de um string é dinâmico e varia de acordo com a cadeia de caracteres armazenada no mesmo, enquanto que o comprimento de um arranjo é fixo.

2. Uma variável do tipo string pode ser lida ou apresentada através de uma única instrução **read** ou **write**, respectivamente. Um arranjo pode ser lido ou apresentado apenas elemento a elemento através de laços **for**.

3. Variáveis string de comprimentos diferentes são compatíveis (isto não é válido para qualquer implementação de Pascal!), enquanto que apenas arranjos do mesmo tipo são compatíveis.

4. Strings podem ser utilizados como tipos de retorno de funções, enquanto que arranjos não podem.

[25] **Cuidado**: Quando comparam-se strings consistindo de dígitos alguns resultados inesperados podem surgir. Por exemplo, `'10' < '8'` resulta em verdadeiro apesar de o número 10 ser obviamente maior do o número 8. Isto ocorre porque os dois strings diferem nos primeiros caracteres correspondentes e `'1'` é menor do que `'8'`.

4.6 Operações sobre Cadeias de Caracteres

O Turbo Pascal oferece algumas funções e procedimentos predefinidos para manipulação de strings. Estes procedimentos e funções, que serão resumidos a seguir, não estão disponíveis no Pascal padrão (embora a maioria das implementações de Pascal ofereçam procedimentos de manipulação de strings equivalentes aos apresentados aqui).

4.6.1 Conversão entre Números e Strings

O procedimento **Val** serve para converter um string numérico num número. A forma de chamada do procedimento **Val** é:

Val(<string numérico>, <número>,

onde <*string numérico*> é o string a ser convertido, <*número*> é uma variável cujo tipo é **integer** ou **real**, de acordo com o número resultante na conversão, e <*erro*> é um valor inteiro retornado pelo procedimento que indica se houve erro na conversão e, em caso afirmativo, em que posição no string o erro ocorreu. Por exemplo, suponha que `N` e `E` são do tipo **integer**. Então, tem-se que a chamada:

```
Val('123', N, E)
```

retorna os valores `123` para `N` e `0` para `E`, este último indicando que não houve erro na conversão. Por outro lado, a chamada:

```
Val('1.23', N, E)
```

retornaria o valor `2` para `E`, indicando que houve um erro de conversão na segunda posição do string `1.23`, e o valor de `N` seria indefinido (lembre-se que supõe-se que `N` é do tipo **integer**). É sempre importante verificar o valor retornado em <*erro*> (através de uma instrução **if**) antes de utilizar o valor retornado em <*número*>.

O procedimento **Str** funciona do modo inverso de **Val**. Isto é, este procedimento é utilizado para converter números em strings. Uma chamada do procedimento **Str** tem a seguinte forma:

Str(<número : formato>, <string numérico>)

onde, <*número : formato*> especifica o número a ser convertido com um formato que segue as especificações de formato vistas na *Seção 3.10*, e <*string numérico*> é o string resultante da conversão. Exemplo de uma chamada ao procedimento **Str**:

```
Str(32.5 : 7:2, meuString)
```

Esta chamada resultaria no retorno do string `' 32.50'` (de comprimento igual a 7) na variável `meuString`. (Note que existem dois espaços em branco antes do `3` no string. Isso é devido ao formato que especifica que o comprimento total deve ser 7 e o número de casas decimais 2.)

4.6.2 Substrings e a Função Copy

A função **Copy** serve para se *extrair* um substring contido em um string maior. A função Copy tem o seguinte formato de chamada:

Copy(<string fonte>, <índice de início>, <tamanho>)

onde, <*string fonte*> é o string (variável ou constante) do qual se deseja extrair o substring, <*índice de início*> é o índice que especifica onde o substring começa no string fonte, e <*tamanho*> informa qual é o tamanho do substring; <*índice de início*> e <*tamanho*> devem ser do tipo **integer**. Por exemplo, a chamada:

```
meuString := Copy('21 de Dezembro de 1995', 7, 8)
```

atribui à variável `meuString` o substring `'Dezembro'`.

4.6.3 Concatenação de Strings

A função predefinida **Concat** permite a concatenação de vários strings num único string. A forma de chamada desta função é:

Concat(string1, string2, ..., stringN)

onde `string1`, `string2`, ..., `stringN` são os strings (variáveis ou constantes) que serão concatenados. Por exemplo, a chamada:

```
meuString := Concat('Bom dia,', ' ', 'senhores', '.')
```

atribui à variável `meuString` o string `'Bom dia, senhores.'`.

Em Turbo Pascal, o operador "+" pode também ser utilizado para concatenação de strings. Por exemplo, a atribuição:

```
meuString := 'Bom dia,' + ' ' + 'senhores' + '.'
```

produziria exatamente o mesmo efeito da chamada da função **Concat** do exemplo anterior.

4.6.4 Localização de Substrings

A função **Pos** retorna o índice da primeira ocorrência de um dado substring num dado string. O formato de chamada da função **Pos** é:

Pos(<substring>, <string>)

onde <*substring*> é o substring a ser procurado no string <*string*>. Qualquer um dos parâmetros pode ser variável ou constante. Quando chamada, a função Pos examina o string <*string*> da esquerda para direita até encontrar a primeira ocorrência de <*substring*>. Se este for realmente encontrado, o valor retornado será o índice no string <*string*> do primeiro caracter do <*substring*>. Caso o substring não seja encontrado, o valor retornado será `0`. Por exemplo, a chamada:

```
minhaVarInteira := Pos('lina', 'Carolina')
```

retornaria o valor `5`.

4.6.5 Procedimentos Delete e Insert

O procedimento **Delete** serve para remover um substring de um string que o contém. Este procedimento tem a seguinte forma:

Delete(<string fonte>, <índice de início>, <tamanho>)

onde os parâmetros <*string fonte*>, <*índice de início*> e <*tamanho*> têm as mesmas interpretações que os parâmetros da função **Copy** vista acima. Note, entretanto, que existe uma diferença fundamental: o string <*string fonte*> ***deve*** ser uma variável (no caso da função **Copy** ele pode ser constante). Por exemplo, no final do trecho de programa:

```
meuString := 'Carolina'
Delete(meuString, 6, 3)
```

a variável `meuString` conteria o valor `'Carol'`. (O mesmo resultado poderia ser obtido neste exemplo utilizando a chamada: `Delete(meuString, Pos('ina', meuString), 3)`. Qual é a vantagem deste segundo exemplo de chamada?)

O procedimento **Insert** serve para inserir um substring num string. Este procedimento tem o seguinte formato de chamada:

Insert(<string>, <string de destino>, <índice>)

onde, <*string*> é o string (constante ou variável) a ser inserido, <*string de destino*> é o string (***deve*** ser variável) que receberá o acréscimo e <*índice*> (do tipo **integer**) especifica a posição no string de destino a partir da qual a inserção será feita. Por exemplo, no final do trecho de programa:

```
meuString := 'Katerine'
Insert('h', meuString, 4)
```

a variável `meuString` conteria o valor `'Katherine'`. (O mesmo resultado poderia ser obtido neste exemplo utilizando a chamada: `Insert('h', meuString, Pos('e', meuString))`. Qual é a vantagem deste segundo exemplo de chamada?)

4.7 Exercícios de Revisão

1. Quais são as principais diferenças entre variáveis de um tipo arranjo e variáveis comuns?

2. Que condição deve ser satisfeita por todos os elementos de um arranjo?

3. Como são identificados os elementos individuais de um arranjo multidimensional?

4. O que são índices e como eles são escritos? Que restrições aplicam-se aos valores que podem ser atribuídos aos subscritos de um arranjo?

5. Qual é a principal vantagem obtida ao se definir os índices de um arranjo através de uma constante simbólica ao invés de um valor inteiro fixo?

6. De que modo os arranjos são normalmente manipulados em Pascal? Qual é a única exceção a essa regra de manipulação?

7. Suponha que o tipo `tArranjoReal` é definido como `type tArranjoReal = array [1 .. 100] of real` na seção de declarações de um programa Pascal. Diga o que há de errado nos cabeçalhos das seguintes declarações de funções nesse programa:

(a) `function Funcao1(var vetor : array [1 .. 100] of real) : char;`
(b) `function Funcao2 : tArranjoReal;`

8. Encontre o erro no seguinte fragmento de programa em Pascal:

```
type
   tArranjoInteiro = array [1..9] of integer;

var
   X : tArranjoInteiro;
   i : integer;

begin
    for i := 1 to 5 do
       X[2*i] := i*i
end.
```

Quando este erro será detectado (i.e., durante a compilação ou durante a execução do programa)?

9. Escreva uma declaração de um arranjo que utilize como subscrito os dias da semana, começando com `DOMINGO`, como subscrito.

10. Escreva um programa em Pascal que imprima o índice e o valor do maior número num arranjo X de 10 inteiros. O arranjo X dever ser lido no meio de entrada.

4.8 Exercícios de Programação

EP4.1) Suponha a existência das seguintes declarações num programa Pascal:

```
const
    MAX = 20;

type
    tArranjoDeCaracteres = array [1..MAX] of char;

var
    X : tArranjoDeCaracteres;
    Y : tArranjoDeCaracteres;
```

(a) Escreva um procedimento com três parâmetros `X`, `MAX` (entrada) e `Y` (saída) que copia os elementos do arranjo `X` no arranjo `Y`, mas na ordem inversa (i.e., o primeiro elemento de `X` deverá ser o último de `Y`, o segundo elemento de `X` deverá ser o penúltimo de `Y`, e assim por diante). (b) Este procedimento poderia ser escrito como uma função cujo valor retornado fosse o arranjo copiado? (c) Complete o programa acima de modo que ele leia o arranjo `X`, chame o procedimento especificado em (a) para copiar o arranjo `X` em `Y`, e finalmente, imprima os arranjos `X` e `Y`. (d) Que modificação seria necessária em seu programa para processar um arranjo de 100 caracteres (ao invés de 20)?

EP4.2) (a) Escreva um programa em Pascal que leia `N` inteiros em dois arranjos `X` e `Y`. Então, esse programa deve comparar os elementos correspondentes de `X` e `Y` (i.e., `X[1]` com `Y[1]`, `X[2]` com `Y[2]`, etc.) e atribuir valores aos elementos correspondentes de um terceiro arranjo `Z` da seguinte forma:

$$\begin{cases} Z[i] = 1, & \text{se } X[i] > Y[i] \\ Z[i] = 0, & \text{se } X[i] = Y[i] \\ Z[i] = -1, & \text{se } X[i] < Y[i] \end{cases}$$

O programa deve, finalmente, imprimir uma tabela de três colunas contendo os valores de `X`, `Y` e `Z` em cada coluna. (b) O valor de `N` pode ser lido no meio de entrada? Por que? (c) Teste seu programa para `N = 10`.

EP4.3) Suponha que um dado tipo arranjo represente matrizes `NxM` de números reais. (a) Escreva as declarações necessárias para implementar este tipo de dados; (b) Escreva um procedimento que lê as matrizes linha por linha. (c) Escreva um procedimento que soma duas matrizes e armazena o resultado numa terceira matriz. (d) Escreva um procedimento que multiplica duas matrizes e armazena o resultado numa terceira matriz. (Lembre-se que existem condições que devem ser satisfeitas para que duas matrizes possam ser multiplicadas. Seu procedimento (d) deve verificar se essas são realmente satisfeitas antes de efetuar a multiplicação.)

EP4.4) Escreva um programa Pascal que converte cada letra minúscula num arranjo de N caracteres na letra maiúscula equivalente. (**Sugestão**: Utilize a função predefinida **Upcase** do Turbo Pascal que retorna a letra maiúscula equivalente a uma dada letra minúscula.)

EP4.5) Um palíndromo é uma cadeia de caracteres que representa a mesma palavra nos sentidos direto e inverso (por exemplo, “6996” é um palíndromo). Escreva uma função booleana em Pascal que recebe uma cadeia de caracteres como entrada e retorna **true** se as ela é um palíndromo.

Capítulo 5

REGISTROS

5.1 Introdução

Um **registro** é um tipo de dados estruturado cujos elementos (ou componentes) podem ser diferentes (portanto, ao contrário dos arranjos, os registros constituem uma estrutura de dados **heterogênea**). Os elementos de um registro são usualmente denominados de **campos**. Uma declaração de um tipo registro em Pascal tem o seguinte formato:

```
type <nome do registro> = record
                              <campo1> : <tipo1>;
                              <campo2> : <tipo2>;
                                  .
                                  :
                              <campoN> : <tipoN>
                          end;
```

onde, *<nome do registro>* é o identificador do tipo registro sendo declarado, *<campo1>*, ..., *<campoN>* são os identificadores associados aos campos do registro, e *<tipo1>*, ..., *<tipoN>* são, respectivamente, os tipos de *<campo1>*, ..., *<campoN>*. Não existe restrição quanto ao tipo de dados de um campo de um registro (exceto, é claro, que o tipo seja conhecido pelo compilador Pascal antes da declaração do registro). Se dois ou mais campos de um registro têm o mesmo tipo, estes campos podem ser declarados juntos separados por vírgulas. Considere, por exemplo, as seguintes declarações abaixo:

```
type tNotas = record
                  nomeDoAluno : string[30];
                  matricula : string[8];
                  nota1, nota2, nota3, nota4, notaFinal, media: real
              end;

var notasDeIP : tNotas;
```

A variável `notasDeIP` do tipo `tNotas` possui 8 campos. Os dois primeiros campos, `nomeDoAluno` e `matricula` são strings, mas de tipos diferentes (por que?), enquanto que os seis últimos campos são todos do tipo **real**. A variável `notasDeIP` poderia conter, por exemplo, o nome e a matrícula de um aluno juntamente com todas as notas obtidas pelo mesmo na disciplina Introdução à Programação. Um arranjo cujos elementos fossem do tipo `tNotas` (por exemplo, **`array`** `[1..36]` **`of`** `tNotas`) poderia conter os nomes, matrículas e notas de todos os alunos de uma turma de Introdução à Programação.

5.2 Manipulações de Registros

Como todo tipo estruturado, os registros permitem que seus elementos sejam acessados individualmente. O acesso a um campo de um registro é feito através de um **seletor de campo**, que consiste simplesmente do nome da variável do tipo registro, seguido de ponto (“.”), e seguido finalmente pelo nome do campo. Por exemplo, o acesso ao campo `nota1` da variável `notasDeIP`, declarada no exemplo acima, poderia ser feito através da referência `notasDeIP.nota1`. Usando esta referência poder-se-ía, por exemplo, atribuir a nota `10.0` ao campo `nota1` da variável `notasDeIP` como:

```
notasDeIP.nota1 := 10.0;
```

Exercício: Como você poderia acessar (referenciar) o primeiro caracter do campo `nomeDoAluno` da variável `notasDeIP`?

Como ocorre com arranjos, quase todas as operações envolvendo registros referem-se a elementos (i.e., campos) de registros e não a registros como um todo. Existe, entretanto, uma exceção que é a operação de **atribuição de registro**, na qual pode-se copiar todos os valores dos campos de um registro nos campos de outro registro *do mesmo tipo*. Por exemplo, suponha que `notasDeIP2` seja uma variável do tipo `tNotas`; então, a instrução de atribuição de registro a seguir copiaria os valores dos campos da variável `notasDeIP` nos campos da variável `notasDeIP2`.

```
notasDeIP2 := notasDeIP;
```

Registros inteiros (i.e., como um todo) podem ser utilizados como parâmetros de procedimentos e funções, mas um registro *não pode ser utilizado como tipo de retorno de uma função*. Na passagem de parâmetros do tipo registros, um parâmetro real do tipo registro deve ser do *mesmo* tipo do parâmetro formal correspondente.

Não existem procedimentos de entrada ou saída capazes de ler ou apresentar registros inteiros, como um todo. Portanto, entrada e saída de registros devem ser feitas campo a campo. Por exemplo, o procedimento a seguir, poderia ser utilizado para ler as duas primeiras notas de um aluno utilizando um parâmetro do tipo `tNotas`:

```
procedure Le2Notas(var notas : tNotas);

begin
    writeln('Introduza as notas do aluno(a)', notas.nomeDoAluno);
    writeln('1a. Nota: ');
    read(notas.nota1);
    writeln('2a. Nota: ');
    read(notas.nota2)
end;
```

5.3 A Instrução WITH

Existe uma outra forma de acesso aos campos de um registro: através de uma instrução **with**, que tem o seguinte formato:

with <registro> **do**
<instrução>

Onde <*registro*> é uma variável de um tipo registro cujos campos se deseja referenciar, <*instrução*> é uma instrução (ou mais comumente uma seqüência de instruções delimitadas por **begin** e **end**) contendo campos da variável <*registro*>. O efeito da instrução **with** é que seu uso dispensa o uso do nome da variável registro e do ponto na referência a um campo da variável. Em outras palavras, os campos da variável podem ser referenciados sem precisarem ser precedidos pelo nome da variável ou do ponto. O uso da instrução **with** é vantajoso quando vários campos de uma variável registro precisam ser acessados de uma vez. Por exemplo, o trecho de programa:

```
notasDeIP.nomeDoAluno := 'José da Silva';
notasDeIP.matricula : '97100000';
notasDeIP.nota1 := 3.5;
notasDeIP.nota2 := 4.0;
notasDeIP.nota3 := 4.5;
notasDeIP.nota4 := 5.0;
notasDeIP.media:= (notasDeIP.nota1 + notasDeIP.nota2 +
                        notasDeIP.nota3 + notasDeIP.nota4) / 4;
```

é equivalente a:

```
with notasDeIP do begin
    nomeDoAluno := 'José da Silva';
    matricula : '97100000';
    nota1 := 3.5;
    nota2 := 4.0;
    nota3 := 4.5;
    nota4 := 5.0;
    media := (nota1 + nota2 + nota3 + nota4) / 4
end;
```

Apesar de sua utilidade, o uso (e abuso) de instruções **with** pode comprometer a legibilidade dos programas e levar a ambigüidade (do ponto-de-vista do programador). Por exemplo, suponha que a variável `notasDeIP2` tenha sido declarada como sendo do tipo `tNotas`, então nas instruções **with** aninhadas abaixo:

```
with notasDeIP do
    with notasDeIP2 do  begin
        .
        :
        nota1 := 0.0
        .
        :
    end;
```

podem obscurecer (pelo menos à primeira vista) para uma pessoa lendo o programa: (1) que `nota1` é um campo de um registro; ou (2) se este não for o caso, a que variável `nota1` se refere (i.e., `notasDeIP` ou `notasDeIP2`?). Num caso de possível ambigüidade, como no exemplo acima, Pascal considera que a referência é feita à instrução **with** mais próxima do campo sendo referenciado. Portanto, neste exemplo, `nota1` refere-se a `notasDeIP2.nota1`. A mesma observação acima é válida quando há colisão entre identificadores de variáveis comuns e identificadores de campos de registros dentro de uma instrução **with**. Por exemplo, suponha que `nota1` é uma variável do programa do exemplo acima. Neste último caso, a atribuição:

```
nota1 := 0.0
```

dentro da instrução **with** acima continua referindo-se ao campo `notasDeIP2.nota1`.

Deve-se ainda observar que existe uma abreviação para instruções **with** aninhadas. Isto é, uma instrução aninhada envolvendo as variáveis do tipo registro `v1`, `v2`, ..., `vn`, como:

```
with v1 do
    with v2 do
       .
       :
            with vn do
               .
               :
```

é equivalente a:

```
with v1, v2, ..., vn do
      .
      :
```

5.4 Registros Variantes

Todas as variáveis de tipos registros vistas até aqui têm a mesma forma e estrutura. Entretanto, é possível definir registros que possuem alguns campos que são os mesmos para todas as variáveis (a **parte fixa** do registro) e outros campos que podem ser diferentes (a **parte variante** do registro). Num registro de empregados, por exemplo, pode-se desejar adicionar os campos `nomeDoConjuge` e `numeroDeFilhos` baseado no fato de o empregado ser casado; se o empregado não for casado, a variável registro contendo seus dados não precisa conter estes campos uma vez que os mesmos serão vazios. Pode-se obter o resultado desejado em Pascal utilizando-se **registros variantes**. Uma declaração de registro variante tem (usualmente) duas partes: (1) uma **parte fixa** (opcional), e (2) uma **parte variante**. A parte fixa, que contém os campos comuns a todas as variáveis do tipo, é declarada no início da declaração do registro conforme foi visto anteriormente. A parte variante, que contém os campos que podem ser diferentes para diferentes variáveis do tipo, é declarada no final da declaração do registro utilizando a seguinte sintaxe:

```
case <indicador> : <tipo do indicador> of
     <rótulo 1> : (<lista de campos 1>);
     <rótulo 2> : (<lista de campos 2>);
                  .
                  :
     <rótulo N> : <(lista de campos N>);
```

onde *<indicador>* é um campo do registro, denominado de **campo indicador** (*tag*); *<tipo do indicador>* é o tipo do campo indicador que pode ser qualquer tipo não-estruturado previamente definido; *<rótulo 1>*, *<rótulo 2>*, ..., *<rótulo N>* são valores que podem ser assumidos pelo campo indicador; *<lista de campos i>* constituem os campos, bem como os tipos destes, que farão parte do registro quando o campo indicador assumir o valor *<rótulo i>*. Deve-se observar ainda que: (1) todos os identificadores de campos devem ser únicos, i.e., um nome de campo não pode aparecer mais de uma vez na declaração de um tipo registro; (2) uma lista de campos vazia é indicada por um par de parênteses vazio (); e (3) não existe nenhum **end** delimitando o **case**. Considere o seguinte exemplo:

```
type
   tEndereco = record
                    rua : string[20];
                    numero : string[4];
                    cidade, estado : string[20];
                    cep : string[5]
               end;

   tEstadoCivil = (CASADO, SOLTEIRO, DIVORCIADO);

   tData = record
              dia : 1..31;
              mes : 1..12;
              ano : 1900..1999;
           end;

   tEmpregado = record
                   nome : string[30];
                   sexo : (M, F);
                   cpf : string[11];
                   endereco : tEndereco;
                   case  estadoCivil : tEstadoCivil of
                        CASADO : ( nomeDoConjuge : string[30];
                                   numeroDeFilhos : integer );
                        SOLTEIRO : ( moraSozinho : boolean );
                        DIVORCIADO : ( dataDoDivorcio : tData )
                end;
```

No exemplo acima, para cada variável do tipo `tEmpregado`, o compilador irá alocar memória suficiente para acomodar o maior registro variante possível (i.e., aquele no qual o campo indicador `estadoCivil` assume o valor `CASADO`). Entretanto, apenas uma lista de campos variantes, determinada pelo valor do campo indicador `estadoCivil`, é definida num dado instante. Por exemplo, se num dado instante o valor de `estadoCivil` é `CASADO`, os únicos campos variantes que podem ser referenciados são `nomeDoConjuge` e `numeroDeFilhos` (i.e., `moraSozinho` e `dataDoDivorcio` não são definidos neste caso). Portanto, o programador deve sempre consultar o valor do campo indicador para assegurar-se que está se referindo aos campos variantes adequados. Isto é usualmente feito através de instruções **case** que usam o campo indicador da parte variante como seletor. Deve-se observar ainda que, quando se atribui dados iniciais a um registro variante, o campo indicador deve ter seu valor atribuído antes dos valores dos campos variantes. Considere, por exemplo, o trecho de programa abaixo que manipula corretamente a parte variante de uma variável `novoEmpregado` do tipo `tEmpregado` declarado acima:

```
case novoEmpregado.estadoCivil of
  CASADO : begin
             writeln('O nome do cônjuge é ', novoEmpregado.nomeDoConjuge);
              writeln('O número de filhos é ', novoEmpregado.numeroDeFilhos)
           end;
  SOLTEIRO :  if novoEmpregado.moraSozinho then  writeln('Mora sozinho');
  DIVORCIADO :writeln('Data de divórcio ', novoEmpregado.dataDoDivorcio.dia,
                              '/', novoEmpregado.dataDoDivorcio.mes, '/',
                              '/', novoEmpregado.dataDoDivorcio.ano)
end;
```

Exercício: Escreva o trecho de programa acima utilizando instruções **with**.

O campo indicador de um registro variante é usado para determinar os campos variantes que uma variável de um tipo registro variante está assumindo num dado instante. Entretanto, o campo indicador não é requerido (i.e., ele pode ser omitido) numa declaração de um tipo registro variante, apesar de o tipo do indicador ser sempre exigido. Por exemplo, a seguinte declaração de tipo é perfeitamente válida:

```
type
   tRegVariante = record
                     campo1 : real;
                     case 1..3 of
                        1 : (campo10, campo11 : integer);
                        2 : ();
                        3 : (campo21 : char)
                  end;
```

No exemplo acima, mesmo não havendo campo indicador, os campos variantes são especificados pelos valores `1..3` do tipo indicador. Nos casos em que o campo indicador não é utilizado, como no exemplo acima, qualquer campo de qualquer lista de campos variantes pode ser acessado em qualquer instante. O uso de registros variantes sem campo indicador, como no exemplo acima, pode ser perigoso porque, uma vez que são atribuídos valores aos campos de uma lista de campos variantes, não existe nenhuma garantia de que os campos de uma outra lista de campos variantes irão reter seus antigos valores. Por exemplo, suponha que a variável `rv` seja uma variável do tipo `tRegVariante` declarado no exemplo anterior. Então, as instruções:

```
rv.campo10 := 1;
rv.campo11 := 25;
writeln(rv.campo10, rv.campo11);
```

resultariam na impressão dos valores `1` e `25` no meio de saída. Agora, suponha que em seguida, se tenham as instruções:

```
rv.campo21 := 'a';
writeln(rv.campo10, rv.campo11);
```

No final destas últimas instruções, apesar de não se ter explicitamente modificado os valores dos campos `campo10` e `campo11`, estes campos poderão não mais conter os valores antigos, e o resultado impresso não mais será `1` e `25` como antes. Resumindo, a moral da estória é: cuidado com o uso de registros variantes sem campos indicadores. Se um registro variante não possui campo indicador, não existe nenhuma forma de se determinar que lista de campos variantes de uma variável do tipo registro variante está correntemente em uso. Portanto, o programador deve acompanhar a variante atual e certificar-se de não referenciar um campo de uma outra lista de campos variantes enquanto os valores da lista de campos variantes corrente ainda estão em uso. Em qualquer situação, é sempre melhor utilizar um campo indicador na declaração de um tipo registro variante.

5.5 Exercícios de Revisão

1. (a) O que é um registro? (b) Qual é a principal diferença entre arranjos e registros?

2. (a) Como são denominados os componentes de um registro? (b) Explique como são referenciados os componentes de um registro.

3. O tipo de dados **string** de Turbo Pascal é um arranjo ou um registro? Explique.

4. (a) Qual é a diferença entre registros e registros variantes? (b) Como deve ser declarado um registro variante? (c) Como deve ser inicializado um registro variante?

5. É possível a coexistência, num mesmo escopo, de dois registros de tipos diferentes com um nome de campo coincidente? Explique.

6. Suponha a existência das seguintes declarações:

```
type
   tEndereco = record
                  rua : string[20];
                  numero : string[4];
                  cidade, estado : string[20];
                  cep : string[5]
               end;

   tEmpregado = record
                   nome : string[30];
                   sexo : (M, F);
                   cpf : string[11];
                   dependentes : integer;
                   endereco : tEndereco
                end;

var novoEmpregado : tEmpregado;
```

(a) Escreva uma instrução **if** que verifique o valor contido no campo `sexo` da variável `novoEmpregado`.
(b) Escreva uma instrução que imprime o `cpf` contido na variável `novoEmpregado`.
(c) Como você poderia acessar o campo `rua` através da variável `novoEmpregado`?
(d) Como você poderia acessar o primeiro dígito do campo `cep` através da variável `novoEmpregado`?

7. Suponha a existência das seguintes declarações num programa Pascal:

```
type
  tMesesDoAno = (JAN, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SEP, OUT,NOV, DEZ);
  tDiasDaSemana = (SEG, TER, QUA, QUI, SEX, SAB, DOM);
  tData1 = record
               dia : 1..31;
               mes : 1..12;
               ano : 1900..2000
           end;
  tData2 = record
               dia : tDiasDaSemana;
                diaDoMes : 1..31;
                mes : tMesesDoAno
            end;

  var
    data1 : tData1;
    data2 : tData2;
```

Diga qual das instruções **with** aninhadas abaixo é inválida e explique porque:

```
with data2, data1 do begin
   dia := 16;
   mes :=  7;
   ano :=  1996;
   diaDoMes := 16
end;
```

```
with data1, data2 do begin
   diaDoMes := 16;
   ano :=  1996;
   mes :=  7;
   dia := 16
end;
```

8. Suponha a existência das seguintes declarações num programa Pascal:

```
type
   tPonto = record
                coordX,  coordY : real
            end;
var
   centro : tPonto;
   coordX,  coordY : real;

procedure Proc1(ponto :  tPonto; var x, y : real);

begin
    with ponto do begin
       coordX := coordX + x;
       coordY := coordY + y;
    end (* with *)
end; (* Proc1 *)
```

Quais serão os valores das variáveis `coordX` e `coordY` após a chamada do procedimento `Proc1` no trecho de programa abaixo?

```
coordX := 0.0;
coordY := 1.0;

centro.coordX := 4.0;
centro.coordY := 2.0;

Proc1(centro, coordX, coordY);
```

9. Suponha que `A` seja um arranjo unidimensional de registros. O tipo registro de cada elemento contém os campos `campo1`, `campo2`, ..., `campo10`. A instrução **with** abaixo:

```
with A[2*i DIV j + k]  do  begin
   campo1 := valor1;
   campo2 := valor2;
            .
            :
   campo10 := valor10;
 end;
```

é mais eficiente do que a seguinte seqüência de instruções equivalentes? Explique.

```
A[2*i DIV j + k].campo1 := valor1;
A[2*i DIV j + k].campo2 := valor2;
                  .
                  .
                  .
A[2*i DIV j + k].campo10 := valor10;
```

10. Suponha a existência das seguintes declarações num programa Pascal:

```
type
   tParDeInteiros = record
                        primeiro,  segundo : integer
                    end;

   tArranjoDeParesInteiros = array [1..15]  of   tParDeInteiros;

var
   A : tArranjoDeParesInteiros;
   i : integer;
```

Suponha ainda que o arranjo `A` tenha sido inicializado e que se deseja imprimir o segundo campo de cada elemento do arranjo cujo valor do primeiro campo seja maior do que `10`. Uma forma correta de se implementar isso é através do trecho de programa a seguir:

```
for i := 1 to 15 do
   with A[i] do
       if primeiro > 10 then
           writeln(segundo);
```

Uma forma aparentemente correta, mas na realidade ***incorreta***, de se traduzir o trecho de programa acima usando uma instrução **while** seria:

```
i := 1;
with A[i] do
    while i <= 15 do begin
         if primeiro > 10 then
               writeln(segundo);
         i := i + 1
    end; (* while *)
```

Explique porque o segundo trecho de programa é incorreto e apresente uma forma de contornar o erro.

5.6 Exercícios de Programação

EP5.1) Um número complexo possui duas partes: (i) real, e (ii) imaginária, sendo ambas números reais. (a) Implemente números complexos através de uma declaração de tipo registro (`tComplexo`) contendo as partes real e imaginária de um número complexo. (b) Escreva um procedimento para somar dois números complexos. (c) Escreva um procedimento para multiplicar dois números complexos. (d) Escreva um programa que lê dois números complexos introduzidos pelo teclado e imprime os números complexos resultantes da soma e da multiplicação dos dois números complexos de entrada. Os resultados devem ser impressos na forma usual (por exemplo, 2 + 3i). (Para simplificar, leia as partes reais e imaginárias dos números complexos separadamente.)

EP5.2) Escreva um programa em Pascal que utiliza um arranjo de registros para armazenar as notas dos alunos de uma turma de Introdução à Programação. Este programa deverá utilizar:

(a) Um procedimento para ler o nome, matrícula, e as notas parciais e final de cada aluno. Este procedimento deverá emitir uma mensagem de erro (por exemplo, `'Matrícula inválida'`) se o número de matrícula contiver menos de 8 caracteres. Também, este procedimento não deverá solicitar ao usuário a nota da prova final de um aluno aprovado por média (i.e., cuja média das provas parciais é maior do que ou igual a 7.0).

(b) Um procedimento que calcula a média final de cada aluno. A média final é calculada da seguinte forma: (i) se a média das provas parciais for maior do que ou igual a 7.0 (i.e., o aluno foi aprovado por média), esta média será também a média final; (ii) caso contrário, a média final será obtida da seguinte forma: multiplica-se a média das notas parciais por 6, soma-se com a nota da prova final multiplicada por 4, e divide-se o resultado por 10.

(c) Um procedimento que imprime, em cada linha, o nome do aluno, sua média final e seu status final, i.e., se o aluno foi aprovado ou reprovado.

(d) Um procedimento que imprime, após a execução do procedimento (c), o número de alunos aprovados e reprovados, e a média final da turma.

Teste seu programa com uma turma de cinco alunos. (**Sugestão**: utilize o tipo `tNotas` definido acima como o tipo dos elementos do arranjo.)

EP5.3) Num arranjo de registros, um campo de um registro cujo valor nunca se repete é denominado de **chave**. (Em outras palavras, cada registro tem seu próprio valor de chave, de modo que dois registros diferentes não podem ter o mesmo valor de chave.) O número de matrícula de um aluno é um exemplo de uma chave, uma vez que dois alunos não podem ter o mesmo número de matrícula. Modifique o programa escrito em **EP5.2** da seguinte maneira:

(a) O procedimento que lê os dados do aluno (procedimento (a) no programa anterior) deve verificar cada número de matrícula introduzido para verificar se ele já existe. Em caso afirmativo, o procedimento deve emitir uma mensagem comunicando o fato ao usuário, e rejeitar os dados para aquele aluno. (**Sugestão**: construa uma função booleana JaExiste que recebe um número de matrícula e o arranjo de registros como entrada e retorna **true** se a matrícula já existe. É importante, neste caso, que se inicialize todos os números de matrícula com o string vazio ('') no início do programa. Pode ser que o Turbo Pascal faça isto automaticamente, mas você não deve esperar que outras implementações de Pascal o façam.)

(b) Os procedimentos de apresentação dos dados de saída (procedimentos (c) e (d) no programa anterior) não deve mais apresentá-los automaticamente como antes. Em vez disso, este procedimento deve solicitar ao usuário para digitar o número de matrícula de um aluno, encontrar o registro do aluno utilizando seu número de matrícula e apresentar o nome do aluno, sua média final e seu status final. Este procedimento deve executar um laço que faz isso até que o usuário digite um número de matrícula inválido ou que não possa ser encontrado.

Capítulo 6

ARQUIVOS

6.1 Introdução

Arquivos podem ser utilizados para armazenar dados permanentemente num meio de armazenamento secundário como, por exemplo, um disco rígido. Pascal provê facilidades para manipulação de arquivos, cujos componentes devem ser todos do mesmo tipo. (Os componentes de um arquivo são comumente denominados **registros**, embora nem sempre "registro" aqui corresponda à estrutura de dados registro vista no capítulo precedente.)

Existe um tipo de arquivo predefinido em Pascal, chamado **text**, que consiste de seqüências de caracteres, (usualmente) agrupadas em linhas. Pode-se declarar uma variável deste tipo simplesmente da mesma forma que uma variável de qualquer outro tipo (i.e., através de uma declaração: **`var`** `<nome da variável> :` **`text`**). Por exemplo:

```
var meuArquivoTexto : text;
```

Deve-se notar que o nome da variável é um identificador válido em Pascal, i.e., ele não segue as convenções para nomes de arquivos de nenhum sistema operacional. Portanto, é necessário que se associe um nome de arquivo real (i.e., tal como o sistema operacional utilizado o identifica) a uma variável de arquivo. A instrução **assign** é utilizada em Turbo Pascal para associar uma variável de um tipo arquivo com um nome de arquivo (usualmente, em disco) a ser lido ou gravado. A instrução **assign** tem a seguinte sintaxe:

assign (<variável do tipo arquivo>, <nome de arquivo>)

onde, *<variável do tipo arquivo>* é o nome de uma variável do tipo arquivo, e *<nome de arquivo>* é um string (entre aspas) representando o nome de um arquivo (usualmente) em disco a ser associado com a variável. Este nome de arquivo em disco

deve seguir as normas do sistema operacional sendo utilizado[26]. Exemplo de uso da instrução **assign**:

```
assign(meuArquivoTexto, 'c:\dados.txt')
```

A instrução do exemplo acima, associa a variável `meuArquivoTexto` ao arquivo `dados.txt` no diretório-raiz do disco **c** (segundo as normas do sistema operacional DOS). Deve-se observar que esta é a ***única*** situação onde é necessário referir-se a um nome de arquivo real em Pascal; i.e., quaisquer manipulações do arquivo serão feitas através da variável associada ao nome real do arquivo (e não através deste).

Um dado arquivo pode ser utilizado para entrada, quando seus componentes são utilizados como dados para o programa, ou saída, quando os resultados do programa são escritos no arquivo. Entretanto, num dado instante, um dado arquivo pode ser utilizado apenas como entrada ou apenas como saída (i.e., num dado instante, um arquivo não pode ser simultaneamente de entrada e de saída).

Utilizar um arquivo através de uma variável do tipo arquivo em Pascal normalmente envolve três passos:

> **1. Preparar o arquivo** para leitura (no caso de arquivo de entrada) ou gravação (no caso de arquivo de saída). Todo arquivo possui um **apontador de posição** que indica (i.e., aponta para) a posição dentro do arquivo na qual o próximo dado será lido ou gravado. Portanto, preparar um arquivo para leitura significa mover o apontador de posição para o início do arquivo, de modo que o primeiro dado a ser lido corresponda ao dado da primeira posição do arquivo. De modo semelhante, preparar um arquivo para gravação significa mover o apontador de posição para o início do arquivo, de modo que o primeiro dado será gravado no início do arquivo (neste último caso, se o arquivo já contiver dados quando ele for preparado, estes dados serão perdidos na primeira operação de gravação).
>
> Os procedimentos **reset** e **rewrite** preparam um arquivo para leitura e gravação respectivamente. As formas de chamada dos procedimentos **reset** e **rewrite** são:
>
> **reset**(<variável de arquivo de entrada>)
>
> **rewrite**(<variável de arquivo de saída>)

[26] A instrução **assign** existe em Turbo Pascal, mas não faz parte do Pascal padrão.

onde <*variável de arquivo de entrada*> e <*variável de arquivo de saída*> são variáveis de arquivos.

2. Ler ou gravar no arquivo. Operações de leitura e gravação de arquivos podem ser efetuadas através dos procedimentos de entrada e saída **read**, **readln**, **write**, e **writeln** vistos antes (v. *Seção 3.10*). A única diferença com relação ao que foi visto anteriormente sobre estes procedimentos é que, no caso de entrada ou saída utilizando um arquivo, o primeiro argumento (parâmetro) desses procedimentos deve ser o nome da variável associada ao arquivo[27].

A forma mais geral da instrução **readln** (ou **read**) é:

readln(<variável de arquivo de entrada>, <lista de variáveis de entrada>)

onde <*variável de arquivo de entrada*> denota uma variável representando o arquivo de onde os dados serão lidos; os valores presentes na linha corrente do arquivo são lidos nas variáveis contidas na <*lista de variáveis de entrada de entrada*>. Após a leitura dos dados da linha corrente com **readln**, a próxima linha do arquivo torna-se a linha corrente. Isto é, o apontador de posição passa a apontar para o primeiro dado na próxima linha. No caso da instrução **read**, o apontador passa a apontar para o próximo dado, mas não *pula* para a próxima linha. Se a <*variável de arquivo*> não for especificada, assume-se que esta variável é **input** (i.e., o teclado do computador).

A forma mais geral da instrução **writeln** (ou **write**) é:

writeln(<variável de arquivo de saída>, <lista de variáveis de saída>)

onde <*variável de arquivo de saída*> denota uma variável representando o arquivo para onde os dados contidas na <*lista de variáveis de saída*> serão

[27] Os procedimentos **readln** e **writeln** podem ser utilizados apenas com arquivos do tipo **text**, como será visto mais adiante.

escritos (gravados). O procedimento **writeln** escreve automaticamente uma marca de final de linha após a escrita do último valor da <*lista de variáveis de saída*>. Se a <*variável de arquivo de saída*> não for especificada, assume-se que esta variável é **output** (i.e., a tela do computador). A diferença entre **writeln** e **write** é que esta última instrução não escreve automaticamente nenhuma marca de final de linha.

3. Fechar o arquivo. Quando lê-se ou grava-se um arquivo, parte do arquivo (ou mesmo todo o arquivo) é mantido numa porção de memória principal do computador (esta porção de memória é denominada **área de buffer**, ou simplesmente, **buffer**). Portanto, "fechar um arquivo" significa liberar este espaço em memória ocupado pelo arquivo. Fechar um arquivo é especialmente importante quando este se trata de um arquivo de saída, pois esta operação resulta na escrita da porção do arquivo ainda em memória principal para o arquivo no meio de armazenamento externo. Em Pascal, a operação de fechamento de um arquivo (de entrada ou de saída) é efetuada através do procedimento **close**, cuja forma de chamada é:

close(<variável de arquivo>)

Parâmetros de procedimentos e funções podem ser de um tipo arquivo (incluindo do tipo **text**). A única restrição feita neste caso é que o parâmetro seja variável (i.e., parâmetros de um tipo arquivo *devem* ser precedidos pela palavra **var**; um parâmetro de um tipo arquivo não pode ser um parâmetro de valor, mesmo que ele seja apenas um parâmetro de entrada).

Se um programa faz referência a um arquivo externo, o nome da variável de arquivo que o representa deve fazer parte do cabeçalho do programa. Por exemplo, se um programa acessasse os arquivos `a1`, `a2`, e `a3`, então o cabeçalho do programa seria algo como:

```
program  MeuPrograma(a1, a2, a3);
```

Esta é uma exigência do Pascal padrão, mas não existe em Turbo Pascal.

6.2 Arquivos do Tipo Text

O arquivo do tipo **text** é predefinido em Pascal. Os componentes de um arquivo do tipo **text** são caracteres de Pascal (i.e., objetos do tipo **char**) mais um caracter especial, denominado **final-de-linha**, que indica o final de cada linha de caracteres no arquivo (i.e., o caracter final-de-linha é utilizado para agrupar caracteres em linhas). O teclado e a tela do computador são duas espécies predefinidas de arquivos do tipo **text** representadas, respectivamente, pelas variáveis (implicitamente declaradas em Pascal) **input** e **output**.

Arquivos do tipo **text** podem ser criados usando um editor de texto comum (sem formatação especial de texto). Os caracteres comuns são introduzidos digitando-se as teclas correspondentes, enquanto que o caracter especial final-de-linha é introduzido quando a tecla `<ENTER>` é pressionada. Arquivos do tipo **text** são **arquivos seqüenciais** no sentido de que eles devem ser processados em ordem seqüencial, do primeiro caracter até o caracter desejado (em outras palavras, se você deseja acessar um dado caracter no arquivo, não existe nenhuma forma de fazê-lo diretamente, como, por exemplo, num arranjo de caracteres; neste caso, você terá que *percorrer* todos os caracteres que antecedem, na seqüência do arquivo, o caracter desejado).

Uma vantagem do uso de arquivos do tipo **text** como entrada de dados para um programa é que, uma vez digitados, estes dados podem ser utilizados quantas vezes se desejar pelo programa sem que o programador precise digitar os dados novamente. Isto economiza tempo e é muito importante durante a fase de depuração de um programa. Por outro lado, uma vantagem do uso de um arquivo do tipo **text** como saída de um programa (ao invés de meramente usar a tela ou a impressora) é que os dados resultantes do programa podem ser processados por outro programa ou impressos quantas vezes se desejar.

O programa-exemplo a seguir lê um conjunto de registros de alunos contidos num arquivo de entrada. Cada registro contém o nome do aluno, e duas notas numa dada disciplina. Então, o programa calcula a média de cada aluno e escreve num arquivo de saída o nome e a média do aluno. Tanto o arquivo de entrada (`alunos.txt`) quanto o arquivo de saída (`medias.txt`) são arquivos do tipo **text**. O final do arquivo de entrada é marcado com '*' no nome do aluno.

```
program ExemploDeUsoDeArquivosText;

const
    MARCA_FINAL = '*';

type
    tAluno = record
                  nome : string[20];
                  nota1,
                  nota2,
                  media : real
             end;
var
    arquivoIN,  arquivoOUT : text;
    aluno : tAluno;

begin
   assign(arquivoIN, 'alunos.txt');  (* Associa variáveis a arquivos reais *)
   assign(arquivoOUT, 'medias.txt');
   reset(arquivoIN);             (* Prepara arquivos para entrada/saída *)
   rewrite(arquivoOUT);

   with aluno do begin
       readln(arquivoIN, nome);          (* Lê o nome do primeiro aluno *)
       writeln(arquivoOUT, nome);    (* Escreve o nome do primeiro aluno *)
       while (nome <> MARCA_FINAL) do
           begin
                readln(arquivoIN, nota1, nota2); (* Lê duas notas do aluno *)
                media := (nota1 + nota2)/2;
                writeln(arquivoOUT, media :4:1);  (* Escreve média do aluno *)
                readln(arquivoIN, nome);  (* Lê próximo nome *)
                writeln(arquivoOUT, nome);  (* Escreve próximo nome *)
           end (* while *)
   end (* with *)

   close(arquivoIN);       (* Fecha arquivos *)
   close(arquivoOUT);
end. (* Programa *)
```

Antes de tentar entender o programa do exemplo acima, é melhor examinar um exemplo de um arquivo de entrada para este programa e o arquivo de saída resultante da execução do mesmo. O arquivo "`alunos.txt`" contém apenas três registros de alunos e é organizado como a seguir (o símbolo "◊" denota final de linha; "*" é a marca de final do arquivo adotada aqui):

```
Jose da Silva◊
5.5  3.5◊
Joao Gaia◊
8.0  7.0◊
Maria das Dores◊
6.5  4.5◊
*◊
```

Conforme visto no exemplo de arquivo acima, cada registro de aluno é dividido em duas linhas separadas por uma caracter de final-de-linha. A razão para esta separação em duas linhas é simples: o nome do aluno é um string e como tal deve ser encerrado sempre com um `<ENTER>` (que é representado pelo caracter *invisível* de final-de-linha); outros tipos de dados não precisam ser terminados por `<ENTER>` (por exemplo, se você tiver cinco valores do tipo **real** para serem lidos através de uma instrução **readln**, você não precisa digitar `<ENTER>` depois de cada um deles; a tecla `<ENTER>` precisa ser pressionada apenas depois da entrada de todos os cinco valores).

Qual seria a diferença entre a entrada de dados através do arquivo do exemplo acima e a entrada dos mesmos dados através do teclado? A resposta é: essencialmente, não existe nenhuma diferença! Se você tivesse que entrar os dados do arquivo acima através do teclado (de acordo com as instruções **readln** do programa), você digitaria primeiro o nome do primeiro aluno (`Jose da Silva`), seguido de `<ENTER>`, então digitaria, na próxima linha, suas notas (`5.5` e `3.5`), e assim por diante. Isto está de acordo com o que foi dito anteriormente a respeito do teclado que é considerado pelo Pascal como um tipo especial de arquivo de entrada do tipo **text** (denominado **input**). No entanto, o teclado (ou melhor, o arquivo **input**) tem um *tratamento especial* que outros arquivos de entrada do tipo **text** não têm (esse "tratamento especial" será discutido mais adiante).

Supondo que o programa processasse o arquivo de entrada do exemplo acima, ele produziria o seguinte arquivo de saída (arquivo `medias.txt` no disco):

```
Jose da Silva◊
 4.5◊
Joao Gaia◊
 7.5◊
Maria das Dores◊
 5.5◊
*◊
```

Note que existe um espaço em branco antes de cada média neste arquivo em decorrência do formato `:4:1` nas instruções **writeln** do program. (Por que, então, não foi adotado o formato `:3:1` que evitaria esse espaço adicional em branco?) Qual seria a diferença entre a apresentação de dados neste arquivo de saída e a apresentação equivalente na tela do computador? Novamente, a resposta é: *essencialmente nenhuma*. Isto ocorre porque a tela do computador é um arquivo de saída do tipo **text** (denominado **output**). Da mesma forma que o teclado, a tela do computador (ou melhor, o arquivo **output**) também recebe *tratamento especial* de Pascal.

Em que consiste esse tratamento especial recebido pelo teclado e pela tela do computador? A resposta a esta questão aparece quando você modifica o programa acima para receber entradas pelo teclado e apresentar os resultados na tela. Que mudanças seriam necessárias?

Examinando o programa a partir de seu início, tem-se que:

(1) Não seriam necessárias quaisquer variáveis de arquivos, visto que as variáveis **input** e **output**, representando o teclado e a tela, respectivamente, são implícitas em qualquer programa Pascal.

(2) As instruções **assign** não se aplicam aos arquivos **input** ou **output**, uma vez que não existe nenhum arquivo em disco (ou qualquer outro meio de armazenamento secundário) associado a esses arquivos.

(3) As instruções **reset** e **rewrite** são desnecessárias para os arquivos **input** ou **output**, porque Pascal prepara estes arquivos para entrada e saída (respectivamente) automaticamente.

(4) As instruções **readln** e **read** não precisam incluir o nome do arquivo de entrada quando a entrada é feita pelo teclado, pois o Pascal considera o teclado como meio de entrada padrão (*default*). O mesmo se aplica às instruções **writeln** e **write** quando o arquivo de saída é a tela.

(5) Nem o arquivo de entrada pelo teclado nem o arquivo de saída pela tela precisa ser explicitamente fechado através de instruções **close**, porque o Pascal faz isso automaticamente.

Existe uma forma de se colocarem todos os dados referentes a um aluno numa única linha no arquivo de entrada, ao invés de duas como no arquivo de entrada do exemplo acima. Simplesmente, lê-se o nome do aluno após suas notas. Desta forma, o arquivo de entrada equivalente ao arquivo do exemplo acima teria o seguinte formato:

```
5.5  3.5  Jose da Silva◊
8.0  7.0  Joao Gaia◊
6.5  4.5  Maria das Dores◊
*◊
```

A média do aluno também poderia ser escrita antes do nome do aluno no arquivo de saída, obtendo-se um arquivo com o seguinte formato:

```
4.5  Jose da Silva◊
7.5  Joao Gaia◊
5.5  Maria das Dores◊
*◊
```

Exercício: Modifique o programa acima para levar em consideração estes novos formatos de arquivos de entrada e saída.

O programa do exemplo acima utiliza um asterisco ('*') para sinalizar o final do arquivo de entrada. Na realidade, isso não era necessário, pois Pascal oferece um método mais fácil para se detectar o final de um arquivo. A função booleana predefinida **EOF** recebe como entrada o nome de uma variável de arquivo e retorna **true** se o final do arquivo foi atingido; caso contrário, **false** será retornado. Utilizando a função **EOF** ao invés da marca '*' no trecho principal do programa do exemplo acima, tem-se o seguinte trecho de programa mais simples:

```
with aluno do
    while not EOF(arquivoIN) do begin
         readln(arquivoIN, nome);  (* Lê o nome do aluno *)
         writeln(arquivoOUT, nome);  (* Escreve o nome do aluno *)
         readln(arquivoIN, nota1, nota2);  (* Lê duas notas do aluno *)
         media := (nota1 + nota2)/2;
         writeln(arquivoOUT, media :4:1);  (* Escreve a média do aluno *)
    end (* while *)
```

Utilizando o trecho de programa acima, o arquivo de entrada não deve mais conter o caracter '*' para indicar final da entrada. Compare este último trecho de programa com aquele apresentado anteriormente e convença-se de que entende as mudanças que foram efetuadas.

Saídas de dados do tipo **char** podem ser enviado para impressão de dentro de um programa Pascal por meio de uma instrução **write** ou **writeln** na qual o nome da variável de arquivo é **lst**. Por exemplo, se você substituísse `arquivoOUT` nas instruções **writeln** do programa-exemplo acima por "**lst**", os nomes e médias dos alunos seriam impressos na impressora acoplada ao computador (supondo que existisse uma) ao invés de gravados no arquivo `medias.txt` no disco. Na realidade, **lst** é uma outra variável de arquivo **text** predefinida em Pascal. Existem outras variáveis que, como **lst**, representam **dispositivos lógicos** de entrada e saída. Uma completa discussão de dispositivos lógicos está além do escopo deste livro.

A Função EOLN

Suponha que se desejasse fazer copia de um arquivo do tipo **text** de entrada para um outro arquivo **text** de saída através de um programa Pascal (obviamente, isso poderia ser trivialmente realizado usando o sistema operacional ao invés de um programa Pascal!). Suponha ainda que as variáveis associadas aos arquivos de entrada e saída sejam, respectivamente, `arquivoIN` e `arquivoOUT`, e que `linha` é uma variável do tipo **string** suficiente para conter qualquer linha do arquivo de entrada. Então, o trecho de programa a seguir poderia ser utilizado para realizar a tarefa desejada:

```
while not EOF(arquivoIN)  do  begin
   readln(arquivoIN, linha); (* Lê cada linha do arquivo de entrada *)
   writeln(arquivoOUT, linha);(* Escreve cada linha no arquivo de saída *)
end (* while *)
```

Agora, para complicar um pouco a situação, suponha que se deseje fazer a cópia desejada caracter por caracter, ao invés de linha por linha. (Isso seria talvez necessário se o arquivo de entrada contivesse linhas com mais de 255 caracteres, que é o máximo comprimento permitido para uma variável do tipo **string**.) Sabe-se que cada elemento de um arquivo do tipo **text** é do tipo **char** (ou uma marca de final de linha), mas não se deve esperar que a solução seja simplesmente substituir a variável `linha` (do tipo **string**) por uma variável do tipo **char** no trecho de programa acima. Fazer isto significa que apenas o primeiro caracter de cada linha será lido e copiado, visto que a instrução **readln** *pula* para a próxima linha após ler os dados correspondentes às variáveis de sua lista de variáveis de entrada. (Do mesmo modo, a instrução **writeln** *pula* para a próxima linha após escrever os valores das variáveis em sua lista de variáveis de saída.) A solução parece ser simplesmente usar instruções **read** e **write** (ao invés de **readln** e **writeln**), pois estas instruções não causam *pula-linhas*. No entanto, isso não seria suficiente. Como se poderia detectar o final de uma linha? Como poder-se-ía passar de uma linha para outra no arquivo de entrada? E, finalmente, como poder-se-ía escrever o arquivo de saída de modo a manter a mesma divisão em linhas do arquivo de entrada? O uso de instruções **readln** e **writeln** sem variáveis de entrada ou saída como argumentos (v. *Seção 3.10*) em conjunção com as instruções **read** e **write**, como será mostrado no programa mais adiante, respondem às duas últimas questões. A resposta para a primeira questão está no uso da função **EOLN**.

A função booleana **EOLN** serve para determinar se o apontador de posição de um arquivo está apontando para uma marca de final de linha. A função **EOLN** tem o seguinte formato de chamada:

EOLN(<variável de arquivo do tipo text>)

onde, <*variável de arquivo do tipo text*> é uma variável de arquivo do tipo **text**. Esta função retorna **true** se o apontador de posição do arquivo representado pela variável está apontando para uma marca de final de linha; caso contrário, ela retorna **false**. Note que a função **EOLN** não pode ser utilizada com arquivos que não sejam do tipo **text**.

Finalmente, o trecho de programa a seguir poderia ser utilizado para fazer a cópia desejada caracter a caracter (suponha que `caracter` é uma variável do tipo **char**):

```
while not EOF(arquivoIN)  do  begin
   while  not  EOLN(arquivoIN)  do  begin
      read(arquivoIN, caracter);(* Lê cada caracter do arquivo de entrada *)
      write(arquivoOUT, caracter);(* Grava caracter no arquivo de saída *)
   end  (* while *);

   readln(arquivoIN);  (* Pula para a próxima linha do arquivo de entrada *)
   writeln(arquivoOUT);  (* Pula para a próxima linha no arquivo de saída *)
end  (* while *)
```

Compare este último trecho de programa com aquele que copia o arquivo de entrada linha por linha e convença-se de que entendeu as mudanças efetuadas.

Exercício: O que aconteceria se a instrução **`readln`**`(arquivoIN)` fosse deixada de fora deste último trecho de programa? E se a instrução **`writeln`**`(arquivoOUT)` fosse deixada de fora?

6.3 Tipos de Arquivos Definidos pelo Programador

Pascal permite que o programador defina tipos de arquivos cujos componentes podem ser de qualquer tipo (predefinido ou definido pelo programador). A única exigência feita por Pascal é que todos os componentes de um arquivo sejam de um mesmo tipo. Arquivos cujos componentes não são do tipo **char** (i.e., arquivos que não sejam do tipo **text**) são denominados **arquivos binários**. Arquivos do tipo **text** têm seus conteúdos (que já não são do tipo **char**) convertidos para seqüências de caracteres, separados por

linhas, antes de serem gravados. Por outro lado, arquivos binários são gravados em *estado natural* (i.e., na mesma forma binária em que se encontram na memória do computador). Em contraste com arquivos do tipo **text**, arquivos binários não podem (ou, pelo menos, isso não é prático!) ser criados ou modificados utilizando-se um editor de texto; isto é, arquivos binários podem apenas ser criados ou modificados como resultado da execução de um programa. Será visto, entretanto, que existem certas vantagens do uso de arquivos binários em detrimento de arquivos do tipo **text**.

Um tipo de dados arquivo pode ser declarado em Pascal utilizando-se a seguinte sintaxe:

type <nome do tipo arquivo> = **file of** <tipo dos componentes>

onde, <*nome do tipo arquivo*> é o identificador associado ao tipo arquivo e <*tipo dos componentes*> é o tipo de dados dos elementos que compõem um arquivo daquele tipo. Por exemplo, nas declarações a seguir,

```
type
   tAluno = record
                 nome : string[30];
                 curso : string[10];
                 CRE : real
            end;

   tArquivoDeAlunos =  file  of  tAluno;

var  alunos : tArquivoDeAlunos;
```

a variável `estudantes` é um arquivo do tipo `tArquivoDeEstudantes`, cujos componentes são registros do tipo `tEstudante`.

Os procedimentos **read** e **write** podem ser utilizados para ler de um arquivo binário e gravar num arquivo binário, respectivamente, da mesma forma como foi visto anteriormente. A diferença com relação a essas operações de leitura e gravação com arquivos do tipo **text** é que, com arquivos do tipo **text**, apenas um caracter de cada vez pode ser manipulado por uma dada operação de leitura ou gravação, enquanto que, no caso de arquivos binários, um componente inteiro é manipulado, não importando o

tamanho do componente (i.e., a complexidade de seu tipo de dado)[28]. Lembre-se, entretanto, que os procedimentos **readln** e **writeln** podem apenas ser utilizados com arquivos do tipo **text**[29]. Em resumo, as instruções **read** e **write** podem ser utilizadas para ler e gravar arquivos binários utilizando o mesmo formato utilizado para arquivos do tipo **text**. Existe, entretanto, um detalhe importante quando se utilizam estes procedimentos com arquivos binários: as variáveis na lista de variáveis de entrada ***devem*** ser todas do mesmo tipo dos componentes do arquivo, e as variáveis (ou expressões) na lista de saída ***devem*** ser todas do mesmo tipo dos componentes do arquivo.

Pelo que foi exposto acima, percebe-se que uma grande vantagem de arquivos binários é quando precisa-se realizar operações de entrada ou saída com estruturas de dados complexas.

O programa-exemplo a seguir lê um conjunto de registros de alunos contidos num arquivo de entrada do tipo **text**. Cada registro contém o nome do aluno, e duas notas numa dada disciplina. Então, o programa calcula a média de cada aluno e escreve num arquivo binário de saída cada registro incluindo média do aluno. (Compare este programa-exemplo com aquele apresentado na *Seção 6.2*.)

[28] Isso é verdadeiro mesmo que se tenha mais de uma variável na lista de entrada ou saída, pois, se `arquivo` é uma variável de arquivo e `v1`, `v2`, ..., `vN` é uma lista de variáveis de entrada, **`read`**`(arquivo, v1, v2, ..., vN)` é uma abreviação equivalente a **`read`**`(arquivo, v1)`; **`read`**`(arquivo, v2)`, ..., **`read`**`(arquivo, vN)`. O mesmo vale para outras instruções de entrada/saída que utilizam listas de variáveis.

[29] A razão pela qual os procedimentos **readln** e **writeln**, bem como a função **EOLN**, não podem ser utilizados com arquivos binários é simples: arquivos deste tipo não possuem nenhuma marca de final de linha.

```
program ExemploDeUsoDeArquivos;

type
   tAluno = record
                  nome : string[20];
                  nota1,
                  nota2,
                  media : real
            end;
tArquivoDeAlunos =  file  of  tAluno;

var
   arquivoIN: text;
   arquivoOUT : tArquivoDeAlunos;
   aluno : tAluno;

begin
  assign(arquivoIN, 'alunos.txt'); (* Associa variáveis a  arquivos reais *)
  assign(arquivoOUT, 'alunos.bin');
  reset(arquivoIN);        (* Prepara arquivos para entrada/saída *)
  rewrite(arquivoOUT);

  while not EOF(arquivoIN)  do begin
     with aluno  do begin
        readln(arquivoIN, nome);              (* Lê o nome do aluno *)
        readln(arquivoIN, nota1, nota2);  (* Lê as duas notas do aluno *)
        media := (nota1 + nota2)/2;     (* Calcula a média do aluno *)
     end (* with *);

     write(arquivoOUT, aluno)     (* Escreve todo o registro do aluno *)
  end (* while *)

  close(arquivoIN); (* Fecha arquivos *)
  close(arquivoOUT);
end. (* Programa *)
```

Os registros do arquivo binário de saída do exemplo acima poderiam ser lidos e processados subseqüentemente por outro programa através de instruções simples como:

```
while not EOF(arquivoIN) do begin
   read(aluno);

   <seqüência de instruções que processam campos da variável aluno>

end (* while *)
```

Note que a instrução **`read`**`(aluno)` lê um registro inteiro de cada vez. Entretanto, se o arquivo de saída do programa-exemplo acima fosse do tipo **text**, seria necessário que cada campo do registro fosse lido separadamente.

6.4 Arquivos de Acesso Aleatório

Freqüentemente, deseja-se acessar apenas um dado registro de um arquivo. Como foi visto anteriormente, arquivos seqüenciais (como os arquivos do tipo **text**) não permitem que um dado componente seja acessado diretamente (i.e., em ordem arbitrária ou aleatória). Turbo Pascal e outras implementações de Pascal (mas não Pascal padrão) permitem o **acesso aleatório** de componentes de arquivos binários.

Antes de um componente de um arquivo binário poder ser acessado aleatoriamente, o apontador de posição do arquivo deve ser movido para este componente. Turbo Pascal provê o procedimento **Seek** para esta finalidade. A forma de chamada deste procedimento é:

Seek(<variável de arquivo>, <número de ordem do componente>)

onde <*variável de arquivo*> é uma variável representando um arquivo binário de entrada ou saída, e <*número de ordem do componente*> é o índice do componente dentro do arquivo[30]. O efeito da execução de uma chamada do procedimento **Seek** é mover o apontador de posição do arquivo para a posição onde se encontra o componente especificado por seu índice.

Turbo Pascal também provê a função **Filesize** que fornece o número de registros de um arquivo. A forma de chamada desta função é:

Filesize(<variável de arquivo>)

Esta função é útil, por exemplo, quando se deseja mover o apontador de posição para o final de um arquivo com a finalidade de inserir (i.e., acrescentar) um novo registro naquela posição. Uma chamada do procedimento **Seek** com o seguinte formato:

```
Seek(<variável de arquivo>, Filesize(<variável de arquivo>))
```

produziria o resultado desejado.

[30] **Cuidado**: esta ordenação começa com 0, e não com 1.

6.5 Exercícios de Revisão

1. Aonde arquivos são armazenados?

2. Descreva três vantagens advindas do uso de arquivos para entrada e saída em relação aos dispositivos-padrão de entrada e saída (i.e., teclado e tela).

3. Qual é a única situação na qual é necessário referir-se a um nome de arquivo real (externo) num programa Pascal?

4. Para que serve o apontador de posição de um arquivo?

5. (a) Um parâmetro de procedimento ou função em Pascal pode ser de um tipo arquivo? (b) Em caso afirmativo, existe alguma restrição com relação ao uso de parâmetros deste tipo?

6. (a) Descreva o significado das variáveis **input** e **output** em Pascal. (b) Estas variáveis precisam ser declaradas? Explique. (c) Que tratamento especial (em relação a outras variáveis *normais* de arquivo) Pascal reserva para as variáveis **input** e **output**?

7. (a) Quais são as principais características de um arquivo do tipo **text**? (b) Como se pode criar um arquivo do tipo **text**?

8. (a) O que é um arquivo binário? (b) Como se pode criar um arquivo binário? (c) Existe alguma vantagem em utilizar-se um arquivo binário para armazenar dados ao invés de um arquivo do tipo **text**? Explique.

9. Por que os procedimentos **readln** e **writeln** não podem ser utilizados com arquivos binários?

10. Descreva os propósitos das funções booleanas (a) **EOF**, e (b) **EOLN**.

11. Qual é a finalidade da área de *buffer* quando se está manipulando um arquivo de entrada ou saída?

12. (a) O que significa preparar (i.e., abrir) um arquivo para entrada ou saída? (b) Como isto é efetuado em Pascal?

13. (a) O que significa fechar um arquivo? (b) Como se fecha um arquivo em Pascal? (c) Por que fechar um arquivo de *saída* de um programa é imprescindível?

14. Compare acesso seqüencial com acesso aleatório de arquivos.

15. Um programa em Pascal padrão tem seguinte o cabeçalho:

```
program  MeuPrograma(input, output, f1, f2);
```

O que significam os identificadores entre parênteses?

6.6 Exercícios de Programação

EP6.1) Suponha a existência de um arquivo do tipo **text** (cujo nome real é '`nomes.dat`') contendo em cada linha o primeiro e o último nomes de uma pessoa no seguinte formato: <*último nome*>, <*primeiro nome*> (por exemplo, Oliveira, Ulysses). Escreva um programa em Pascal que:

(a) lê cada nome de pessoa deste arquivo;

(b) calcula o comprimento do primeiro nome da pessoa; e

(c) para cada nome de pessoa lido, escreve para um arquivo de saída o primeiro nome e o comprimento deste calculado em (b). (**Sugestão**: Utilize a função de manipulação de strings **Pos** para encontrar a vírgula que separa o último nome do primeiro nome.)

EP6.2) Um arquivo do tipo **text** (denominado externamente como '`alunos.dat`') contém nomes de alunos (um nome em cada linha). Infelizmente, quando este arquivo foi digitado, a tecla de letras maiúsculas estava defeituosa, de modo que todos os nomes tiveram de ser digitados em letras minúsculas. Escreva um programa em Pascal que:

(a) leia cada nome no arquivo original;

(b) transforme a letra inicial de cada palavra componente de cada nome de aluno em maiúscula (para simplificar não se preocupe em transformar a primeira letra de palavras de ligação, tal como 'de' e 'e' em 'De' e 'E');

(c) grave cada nome transformado em (b) num arquivo denominado (externamente) '`rascunho.tmp`';

(d) copie o arquivo denominado (externamente) '`rascunho.tmp`' no arquivo original '`alunos.dat`'.

Capítulo 7

RECURSÃO

7.1 Definições e Processos Recursivos

Muitas entidades em Matemática são definidas por meio de processos que produzem aquela entidade. Por exemplo, o fatorial de um número inteiro não-negativo n é definido como o produto de todos os inteiros entre 1 e n. Em outras palavras, esta definição é equivalente à instrução: dado um número n, multiplique todos os números entre 1 e n, e denomine o resultado de "o fatorial de n". Essa definição pode ainda ser representada por um algoritmo que recebe como entrada um número n e retorna o valor de n! numa variável de saída `fat`:

```
x ← n;
prod ← 1;
enquanto x <> 0 faça
        prod ← x*prod;
        x ← x - 1;
fat ← prod
```

O algoritmo apresentado acima é chamado de **iterativo** (ou repetitivo) porque ele usa a repetição explícita de algum processo até que uma dada condição é satisfeita. O algoritmo acima pode facilmente ser transformado numa função em Pascal que calcula o fatorial de um parâmetro de entrada inteiro[31].

É fácil verificar que, para n >0, n! = n*(n-1)! (por exemplo, 4! = 4.3.2.1 = 4.3!). Sabe-se também, por definição, que 0! = 1. Portanto, o fatorial de um número não-negativo qualquer pode ser definido por meio das sentenças:

[31] Dada as limitações dos computadores reais (nunca esqueça que qualquer computador é uma máquina finita), uma função Pascal não pode servir como definição matemática precisa de fatorial.

n! = 1, se n = 0
n! = n*(n-1)!, se n > 0

Esta última definição pode parecer estranha pois define fatorial em termos de fatorial em si. Entretanto, esta definição pode ser interpretada da seguinte maneira: dado o fatorial de um número inteiro não-negativo, pode-se obter o fatorial de seu sucessor simplesmente multiplicando o valor daquele fatorial pelo dado sucessor. Por exemplo, 0! = 1 (por definição), logo 1! = 1*0! = 1*1 = 1; similarmente, 2! = 2*1! = 2*1 = 2, e assim por diante. Existem muitas definições semelhantes a esta última em Matemática e Ciência da Computação. Uma definição como tal, que define uma entidade (por exemplo, n!) em termos de um caso mais simples da própria entidade (por exemplo, (n-1)!) é chamada uma **definição recursiva**.

Antes de prosseguir, veja como se pode calcular o fatorial de 5 de acordo com a definição recursiva acima. De acordo com a definição 5! = 5*4!; portanto, para calcular 5!, deve-se primeiro calcular 4!. Novamente, de acordo com a definição 4! = 4*3!; seguindo novamente a definição, 3! = 3*2!, etc. Este processo pode ser ilustrado através do seguinte esquema:

(1) 5! = 5*4!
(2) 4! = 4*3!
(3) 3! = 3*2!
(4) 2! = 2*1!
(5) 1! = 1*0!
(6) 0! = 1

Note que cada caso é reduzido a um caso mais simples até que o caso trivial 0! é atingido. Neste ponto, pode-se retroceder da linha (6) até a linha (1), retornando o valor calculado numa linha para avaliar o resultado da linha anterior. Isto produz o seguinte:

(6') 0! = 1
(5') 1! = 1*0! = 1*1 = 1
(4') 2! = 2*1! = 2*1 = 2
(3') 3! = 3*2! = 3*2 = 6
(2') 4! = 4*3! = 4*6 = 24
(1') 5! = 5*4! = 5*24 = 120

A definição recursiva de fatorial pode ser representada através do seguinte algoritmo:

```
1.    se n = 0 então
2.         fat ← 1
3.    senão
4.         calcule o valor de (n-1)! e chame este valor de y
5.         fat ← n*y
```

Note que o passo 4 do algoritmo acima requer que o próprio algoritmo seja reexecutado com a entrada n-1. Por causa disso, este algoritmo é denominado **recursivo**. Note que, qualquer que seja $n \geq 0$, o algoritmo irá eventualmente parar, pois cada a cada execução do passo 4, o algoritmo é chamado com um valor de entrada que é um a menos que o valor da chamada anterior. Portanto, o valor de entrada 0 será eventualmente atingido e o processo irá terminar com o retorno de 1. Este valor será retornado na linha 4 na variável y. Então multiplicando-se este valor por n (que quando da chamada com entrada 0 valia 1), obtém-se 1! Este valor também seria retornado em y na linha 4 e seria então multiplicado por n (que neste caso seria 2) e obter-se-ía 2!. E assim por diante.

Claramente, este algoritmo é um pouco mais complicado e menos intuitivo de ser acompanhado do que a versão iterativa apresentada antes. O último algoritmo foi apresentado apenas com a intenção de introduzir recursão utilizando uma função matemática bem conhecida.

Um importante requerimento para uma definição ou algoritmo recursivo ser correto é que ele não gere uma seqüência infinita de referências a si mesmo (caso contrário, o algoritmo não terminaria nunca). Por causa disso, uma definição (ou algoritmo) recursiva deve ter pelo menos uma parte que não envolve a definição (ou algoritmo) em si. Isto é, deve sempre haver um **ponto de saída** da recursão. Este ponto de saída é também conhecido como **condição de parada** ou **condição trivial**. Também, cada chamada recursiva deve reduzir cada entrada a um caso mais simples até que a condição de parada seja eventualmente atingida.

Em resumo, para resolver um problema usando recursão, o algoritmo que resolve o problema deve ser dividido em duas partes:

> (1) **Parte recursiva**, na qual a solução para o problema é definida recursivamente (i.e., através de chamadas ao próprio algoritmo); e
>
> (2) **Condição de parada** (ou **caso trivial**) do problema, que corresponde a um caso (ou situação) do problema que se sabe resolver trivialmente.

7.2 Recursão em Pascal

A linguagem Pascal permite que o programador escreva procedimentos e funções que chamam a si mesmos. Tais procedimentos ou funções são ditos serem **recursivos**. Será visto em seguida como algoritmos recursivos podem ser traduzidos em Pascal.

7.2.1 Exemplo de Função Recursiva

Uma função que implementa o algoritmo recursivo para cálculo de fatorial será apresentada em seguida. Mas, antes, suponha a existência da seguintes declarações de tipos:

```
type
    tInteiroNaoNegativo = 0..MAXINT;
    tInteiroPositivo = 1..MAXINT;
```

onde `MAXINT` é uma constante predefinida que representa o valor do maior inteiro possível numa dada implementação de Pascal. Pode-se, então, facilmente traduzir o algoritmo recursivo para cálculo de fatorial na seguinte função Pascal:

```
function  Fatorial(n  :  tInteiroNaoNegativo) : tInteiroPositivo;
var
   x  :  tInteiroNaoNegativo;
   y  :  tInteiroPositivo;

begin
    if  n = 0  then
       Fatorial := 1                      (* Condição de parada *)
    else
       begin
           x := n - 1;
           y := Fatorial(x);        (* Caso recursivo *)
           Fatorial := n * y
    end  (* if *)
end;
```

Na instrução: `y := Fatorial(x)` da função acima, a função chama a si própria, com um argumento cada vez menor a cada chamada. A condição de parada: `Fatorial := 1` garante que a série de chamadas terminará em algum instante.

Exercício: O que aconteceria se o tipo do parâmetro n fosse inteiro (ao invés de `tInteiroNaoNegativo`) e fosse feita uma chamada com n < 0 (por exemplo, `Fatorial(-1)`)?

É interessante examinar a execução de uma chamada da função acima. Por exemplo, suponha que, no programa principal, se tenha a seguinte chamada:

```
z := Fatorial(4);
```

Neste caso, quando da chamada, o parâmetro n assume o valor `4`. Como n é diferente de 0, o caso recursivo da função é executado com a chamada `Fatorial(3)`. Quando esta nova chamada da função Fatorial ocorre, um novo conjunto de variáveis locais (i.e., `x` e `y`) e o parâmetro da função (`n`) são novamente alocados. Entretanto, isso não significa que as duas variáveis locais e o parâmetro anteriores são abandonados, uma vez que a execução da primeira chamada ainda não terminou neste ponto. Na verdade, estes dois conjuntos de variáveis e parâmetros *coexistem*, mas apenas o mais recentemente alocado pode ser referenciado. Aliás, isto é um fato que acontece em geral com chamadas recursivas de funções e procedimentos em Pascal (e em outras linguagens): cada vez que uma função ou procedimento é chamado recursivamente, novos conjuntos de variáveis locais e parâmetros são criados, e apenas os objetos (i.e., variáveis e parâmetros) mais recentemente alocados têm validade num dado instante. Quando uma chamada de procedimento ou função recursiva termina, estes objetos são liberados (i.e., o espaço que ocupavam em memória é liberado) e os objetos que foram alocados imediatamente antes deles tornam-se ativos novamente. Para entender melhor como a execução de uma função ou um procedimento recursivo funciona, é necessário primeiro que se entenda o conceito de **pilha** em programação.

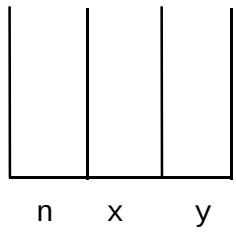
Uma pilha é uma estrutura de dados na qual os dados são colocados (i.e., **empilhados**) uns sobre os outros de acordo com a ordem de chegada. Os itens (de dados) são retirados (i.e., **desempilhados**) na ordem inversa de chegada (i.e., o primeiro item a ser retirado é sempre o último que foi empilhado, depois retira-se o penúltimo, e assim por diante). O que é que isso tudo tem a ver com recursão? A resposta é simples: quando é feita uma chamada recursiva (de função ou procedimento), ela não é executada imediatamente; em vez disso, ela é colocada numa pilha até que a condição que termina a recursão seja eventualmente satisfeita. Quando a condição de parada nunca é encontrada (i.e. quando a recursão é infinita), ou quando a recursão é muito longa, esgota-se a capacidade de armazenamento da pilha e obtém-se um erro de execução do tipo ***stack overflow***.

A pilha é utilizada por Pascal para manter a alocação/liberação de memória a cada chamada/retorno de procedimento ou função. Cada vez que se chama um procedimento ou função recursivo, um novo conjunto de variáveis e parâmetros é empilhado na pilha.

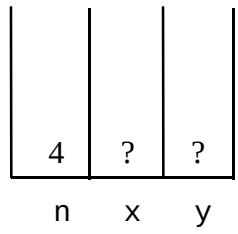
Apenas objetos que estão no topo da pilha podem ser referenciados. Cada vez que se retorna de um procedimento ou função, o conjunto de variáveis e parâmetros correspondente é desempilhado da pilha, e o conjunto anterior (se houver algum) fica no topo da pilha e torna-se ativo. É extremamente útil entender o uso de pilha neste processo, pois a simulação de uso de pilha é a estratégia mais indicada para entendimento e depuração de programas recursivos em Pascal.

As figuras a seguir ilustram várias situações na pilha utilizada na execução da chamada `z := Fatorial(4)` (um ponto de interrogação significa que o valor da respectiva variável é desconhecido). Inicialmente (*Figura (a)*), a pilha está vazia. Logo após a chamada `Fatorial(4)`, `n` vale `4`, e `x` e `y` têm seus valores desconhecidos (*Figura (b)*). Como `n` ≠ `0`, `3` é atribuído a `x` e é feita a chamada `Fatorial(3)` (*Figura (c)*). Neste ponto, o novo valor de `n` é `3`, que é diferente de `0`; portanto, `x` recebe o valor `2` e é feita a chamada `Fatorial(2)` (*Figura (d)*). Este processo continua até que `n` atinge o valor `0` (*Figura (f)*). Então, o valor 1 é retornado e atribuído a `y` (*Figura (g)*). (Veja a implementação da função `Fatorial` acima e note que a atribuição do valor retornado da função a `y` é a próxima instrução após a execução da chamada.) Note que, quando este retorno ocorre, os valores correspondentes à chamada `Fatorial(0)` (topo da pilha na *Figura(f)*) são desempilhados e o próximo conjunto de valores torna-se ativo, com `1` sendo atribuído a `y`.

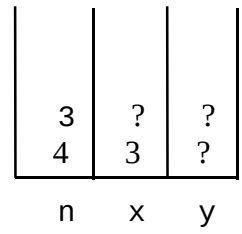
Prosseguindo, a instrução `Fatorial := n*y` é executada, multiplicando-se os valores de `n` e `y` no topo da pilha (*Figura (g)*) resultando em `1`. Este é o valor de retorno de `Fatorial(1)`. Os valores correspondentes a Fatorial(1) são desempilhados e o valor de retorno é então atribuído a `y` (*Figura (h)*). Esse processo prossegue até que a pilha é esvaziada (*Figura (l)*). Neste ponto o valor final retornado será `24` (`n*y = 4*6 = 24` na *Figura (j)*) e este valor será atribuído à variável `z` na instrução `z := Fatorial(4)` que fez a chamada original.



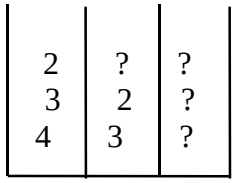
(a) Início



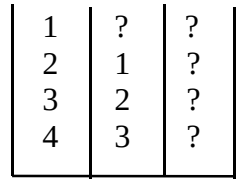
(b) Fatorial(4)



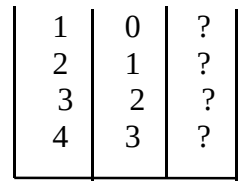
(c) Fatorial(3)



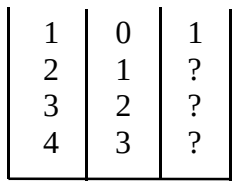
(d) Fatorial(2)



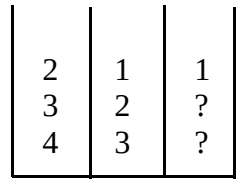
(e) Fatorial(1)



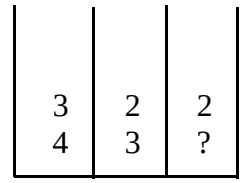
(f) Fatorial(0)



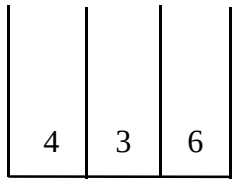
(g) y := Fatorial(0)



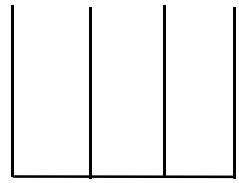
(h) y := Fatorial(1)



(i) y := Fatorial(2)



(j) y := Fatorial(3)



(l) z := Fatorial(4)

Note que, quando uma função é chamada, ela retorna na mesma linha em que foi feita a chamada, uma vez que o valor retornado será certamente utilizado em seguida na mesma linha (no último exemplo, o valor retornado seria atribuído a `z`). Quando um procedimento é chamado (v. exemplo adiante), o ponto de retorno usualmente é na linha seguinte àquela onde foi feita a chamada, uma vez que não existe valor de retorno que possa ser processado na mesma linha de instrução (não confunda valor de retorno de uma função com parâmetro de saída!).

Pode-se ainda observar neste exemplo que as variáveis locais `x` e `y` são absolutamente desnecessárias para implementação da função `Fatorial` apresentada aqui. Isto é, esta função poderia ser mais compactamente declarada como:

```
function  Fatorial(n  :  tInteiroNaoNegativo) : tInteiroPositivo;
begin
    if  n = 0  then
        Fatorial := 1                          (* Condição de parada *)
    else
        Fatorial := n * Fatorial(n - 1);       (* Caso recursivo *)
end;
```

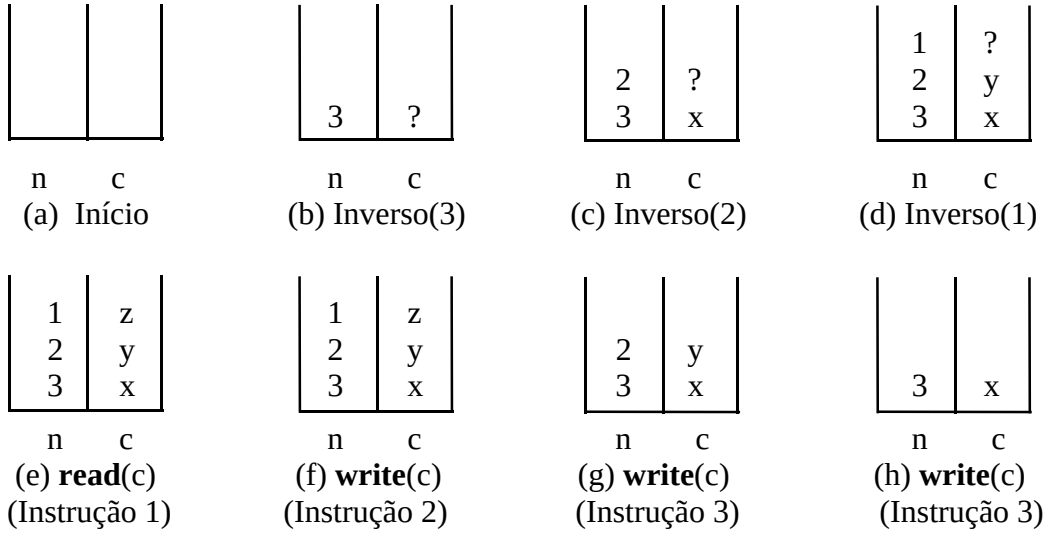
Entretanto, apesar de mais compacta, esta última versão de `Fatorial` não é mais *econômica*, pois Pascal utiliza variáveis temporárias *implícitas* para conter os valores intermediários contidos em `x` e `y` na versão anterior da função `Fatorial`. Além disso, estas variáveis temporárias (implícitas) recebem exatamente o mesmo tratamento que as variáveis `x` e `y` *explicitamente* declaradas (o programador não pode evidentemente referir-se a variáveis temporárias implícitas, pois estas são invisíveis para ele). O uso de variáveis temporárias explicitamente declaradas, tais como `x` e `y` no exemplo anterior, muitas vezes torna mais fácil o acompanhamento (*trace*) de procedimentos e funções recursivos. Tente traçar a execução da chamada `z := Fatorial(4)` utilizando pilha, como foi feito anteriormente, para esta última versão da função `Fatorial`.

7.2.2 Exemplo de Procedimento Recursivo

O procedimento `Inverso` a seguir lê um número n de caracteres no meio de entrada (teclado) e imprime estes caracteres na ordem inversa (em Turbo Pascal, a diretiva `{$B-}` deve ser utilizada; v. *Seção 3.10*). As instruções **read** e **write** foram numeradas para facilitar a discussão que segue.

```
procedure   Inverso(n :  integer);
var   c  :  char ;

begin
    if  n = 1 then
        begin                       (* Caso de Parada *)
            read(c);                (* 1 *)
            write(c)                (* 2 *)
        end
    else
        begin                       (* Caso Recursivo *)
            read(c);
            Inverso(n - 1);
            write(c)                (* 3 *)
        end
end;
```

Suponha que seja feita a chamada `Inverso(3)` no corpo do programa principal e que o usuário tenha digitado os caracteres `xyz` como entrada no teclado. As figuras a seguir ilustram diversas situações na pilha de execução para esta chamada.



A *Figura (b)* ilustra o instante em que o procedimento `Inverte` foi acionado; nesta situação, `n` recebe o valor 3, enquanto c ainda é indeterminado. Como `n` ≠ 1, o controle passa para o caso recursivo (**else**) do procedimento. Então, o primeiro caracter ('x') é lido e atribuído a `c`. Em seguida, é feita a chamada recursiva `Inverso(2)` (*Figura(c)*). O parâmetro `n` nesta chamada vale 2, e entra-se novamente no caso recursivo; o segundo caracter ('y') é lido e atribuído a c, e então feita a chamada `Inverso(1)` (*Figura(d)*). Agora, como `n` assume o valor 1, a condição de parada é atingida; o valor do terceiro caracter ('z') é lido na instrução 1 (*Figura(e)*) e escrito na instrução 2 (*Figura(f)*). Neste instante, ocorre a primeira operação de desempilhamento com o subseqüente retorno do procedimento; o ponto de retorno é a instrução 3 (**`write`**`(c)`) e o caracter 'y' é impresso (*Figura(g)*). O segundo desempilhamento e respectivo retorno ocorre. Novamente, o ponto de retorno é a instrução 3 (**`write`**`(c)`) e o caracter 'x' é impresso (*Figura(h)*). Finalmente, a pilha é então esvaziada com o último retorno. O ponto de retorno agora será a instrução que segue a chamada original `Inverso(3)` no corpo do programa principal.

7.2.3 Cadeias Recursivas

Para um procedimento ser considerado recursivo, não é necessário que ele chame a si próprio *diretamente*. Isto é, um procedimento `P1` pode chamar um outro procedimento `P2` que, por sua vez, chama `P1` de volta. Isto significa que `P1` está chamando

indiretamente a si mesmo através de P2. Neste caso, P1 e P2 formam uma **cadeia recursiva**. Uma cadeia recursiva pode envolver mais do que dois procedimentos (por exemplo, P1 chama P2, que chama P3, que chama P1). O programador deve ter cuidado para que cadeias recursivas não sejam infinitas (i.e., nunca terminem).

Considere o seguinte exemplo esquemático:

```
procedure  P1(<parâmetros formais>);
begin
    .
    :
    P2(<parâmetros reais>);
    .
    :
end;
```

```
procedure P2(<parâmetros formais>);
    begin
        .
        :
        P1(<parâmetros reais>);
        .
        :
    end;
```

Note que a cadeia recursiva representada acima pode comprometer a legibilidade do programa que a contém. Por exemplo, examinando apenas o procedimento P1, não é claro que este procedimento é (indiretamente) recursivo (o mesmo argumento aplica-se ao procedimento P2, que também é recursivo).

Supondo que, na situação apresentada acima, o procedimento P2 é declarado após o procedimento P1 no programa, uma pergunta que surge é: Como se pode referir (i.e., chamar) ao procedimento P2 no procedimento P1 se P2 ainda não foi declarado neste ponto do programa? (Lembre-se que isso é ilegal em Pascal.) A linguagem Pascal oferece uma solução para isso que é o uso da palavra reservada **forward** da seguinte maneira:

```
procedure  P2(<parâmetros formais>);  forward;

procedure  P1(<parâmetros formais>);

      <corpo de P1>

procedure  P2;

      <corpo de P2>
```

A palavra reservada **forward** na primeira declaração acima indica que o corpo do procedimento P2 será especificado subseqüentemente. Note que os parâmetros formais de P2 são declarados na declaração contendo **forward**; quando o corpo de P2 é definido, eles não aparecem na declaração.

7.3 Escrevendo Programas Recursivos

Apesar de ser uma ferramenta poderosa, recursão nem sempre é a melhor forma de se resolver um problema e, em geral, quando depara com um problema, o programador não deve imediatamente procurar uma solução recursiva para o mesmo. Muitos (talvez a maioria) dos problemas práticos que um programador encontra podem ser resolvidos sem o uso de recursão. Entretanto, alguns problemas são resolvidos de forma mais lógica e elegante através do uso de recursão. Como descobrir se uma solução recursiva é a melhor solução para um dado problema? A resposta para esta questão vem de um exame das características de um problema inerentemente recursivo e o que faz uma solução recursiva funcionar.

Em primeiro lugar, num programa recursivo, existem um grande número de casos distintos para resolver (no caso do fatorial, estes casos eram 1!, 2!, 3!, etc.). Em segundo lugar, pode-se encontrar um caso trivial para o qual uma solução imediata pode ser facilmente obtida (no caso de fatorial, este caso era 0! = 1). O próximo passo é encontrar um método para resolver um caso complexo do problema em termos de um caso mais simples do mesmo. Esta redução (contínua) de complexidade do problema deve, em última instância, resultar no caso trivial do problema. No caso do fatorial, o problema complexo n! é reduzido para o problema mais simples n*(n-1)!, e esta redução resulta eventualmente em (n-1)! tornando-se 0!, que é o caso trivial do problema.

O problema do fatorial não constitui um caso cuja solução mais simples ou elegante seja por meio de recursão, mas ele ilustra bem os ingredientes necessários para uma solução recursiva. Estes ingredientes podem ser resumidos em: (1) um caso complexo que pode ser reduzido a um caso mais simples, e (2) um caso trivial que pode ser resolvido diretamente sem usar recursão.

7.4 Iteração x Recursão

Quando um procedimento (ou função) contém variáveis locais, um conjunto diferente de variáveis locais será criado durante cada chamada recursiva do procedimento (ou função). Os valores dos parâmetros envolvidos (ou seus endereços, no caso de parâmetros variáveis) também são copiados para a pilha a cada chamada recursiva. Por causa disso (e também porque a própria montagem da pilha leva tempo), o uso de recursão não é necessariamente a melhor (i.e., mais eficiente em termos computacionais) forma de se resolver um problema, mesmo quando a definição do problema tem natureza recursiva. Uma implementação não-recursiva (i.e., **iterativa**) pode ser muito mais eficiente em termos de utilização de memória e velocidade de

execução. Por causa disso, o uso de recursão geralmente envolve um conflito entre simplicidade de programação e performance computacional.

Portanto, recursão geralmente é ineficiente no uso de recursos computacionais, e em casos onde uma solução iterativa pode ser implementada facilmente, esta solução é a preferida. Nos exemplos apresentados aqui, é muito fácil escrever soluções iterativas. As soluções recursivas apresentadas aqui tinham o propósito ***didático*** de apresentar a técnica de recursão com problemas fáceis de ser entendidos. Os exemplos particulares de soluções recursivas apresentados aqui não seriam utilizados na prática.

Algumas vezes, uma solução recursiva é a forma mais natural e lógica de se resolver um problema. Apesar disso, conforme visto acima, soluções recursivas usualmente custam caro em termos de tempo e espaço de armazenamento (por causa das atividades de empilhar e desempilhar variáveis locais e/ou temporárias, e parâmetros a cada chamada recursiva). Existe, portanto, um conflito entre **eficiência de execução** e **eficiência de programação**. Como o custo de programação tem aumentado e o custo de computadores cada vez mais potentes tem decrescido constantemente, na maioria dos casos, a escolha da solução recursiva (se esta for a escolha mais natural) é melhor do que requerer que o programador gaste seu tempo procurando por uma solução iterativa mais eficiente (em termos de processamento), mas mais difícil e demorada de ser programada. Isto significa que, se um programador encontra uma solução recursiva como sendo a forma mais simples de resolver um dado problema, não vale a pena tentar descobrir um método mais eficiente em termos de computação (execução do programa). (Evidentemente, um programador "*muito esperto*" pode encontrar uma solução recursiva complicada para um problema que é mais facilmente resolvido sem recursão.) Entretanto, essa escolha não é o caso onde eficiência de computação deve ser perseguida a todo custo (por exemplo, na programação de um sistema operacional). Nestas situações, e quando a solução recursiva é fácil de ser encontrada, o melhor caminho pode ser a transformação desta solução recursiva numa solução não-recursiva utilizando um algoritmo de simulação de recursão. A apresentação de um tal algoritmo está além do escopo deste livro (v. referência abaixo).

(Parte do material deste capítulo é derivada de Tenenbaum, A. M. e Augenstein, M. J., *Data Structures Using Pascal*. Prentice-Hall, 1981.)

7.5 Exercícios de Revisão

1. O que é uma definição recursiva?

2. O que é recursão em Pascal e qual é a vantagem advinda de seu uso?

3. (a) O que é uma pilha? (b) Em que ordem dados são acrescentados e retirados de uma pilha?

4. O que acontece com as variáveis locais de um procedimento (ou função) quando o mesmo é chamado recursivamente? O que ocorre com as mesmas variáveis quando acontece um retorno do procedimento (ou função)?

5. O que é uma cadeia recursiva?

6. Qual é o significado da palavra reservada **forward** em Pascal?

7. Por que um programa iterativo é geralmente mais eficiente em termos de utilização de recursos computacionais do que um programa recursivo equivalente (i.e., que realiza a mesma tarefa)?

8. O que acontece quando um programa gera uma seqüência infinita (ou excessivamente longa) de chamadas recursivas de um procedimento ou função?

9. Escreva cada uma das fórmulas algébricas a seguir numa forma recursiva:
   (a) $y = x_1 + x_2 + ... + x_n$.
   (b) $g = f_1 * f_2 * ... * f_n$.

10. A seqüência de Fibonacci pode ser definida recursivamente da seguinte forma:

$$\begin{cases} Fib(n) = n, \text{ se } n = 1 \text{ ou } n = 0 \\ Fib(n) = Fib(n-2) + Fib(n-1), \text{ se } n \geq 2 \end{cases}$$

onde n representa a ordem de cada termo. (a) Escreva uma função recursiva em Pascal que compute o n-ésimo termo da seqüência de Fibonacci utilizando a definição recursiva acima. (b) Acompanhe, por meio de pilhas de recursão, a execução da chamada Fib(6).

11. Seja A um arranjo de N números inteiros. Apresente algoritmos recursivos que calculem:
(a) O maior elemento do arranjo.
(b) O menor elemento do arranjo.
(c) A soma dos elementos do arranjo.
(d) A média aritmética dos elementos do arranjo.

12. Dada a seguinte função recursiva em Pascal (v. definição do tipo `tInteiroNaoNegativo` acima):

```
function   Fun1(n : tInteiroNaoNegativo) : tInteiroNaoNegativo;
begin
    if   n = 0   then
       Fun1 := 0
    else
       Fun1 := n + Fun1(n-1)
end;
```

(a) Determine o que ela faz.
(b) Escreva uma função iterativa em Pascal que realize o mesmo objetivo.

13. Determine a saída do seguinte programa em Pascal:

```
programa   MeuProg;
var  n :  integer;

function   F1(n  :  integer)  :  integer
begin
    if  n > 0   then
        F1 := n + F1(n - 2)
    else
        F1 := 0
end;

begin
    n := 10;
    writeln(F1(n))
end.
```

14. O máximo divisor comum (**mdc**) de dois inteiros x e y pode ser definido da seguinte forma (*Algoritmo de Euclides*):

$$\begin{cases} mdc(x, y) = y, & \text{se } y \leq x \text{ e } x \textbf{ mod } y = 0 \\ mdc(x, y) = mdc(y, x), & \text{se } x < y \\ mdc(x, y) = mdc(y, x \textbf{ mod } y), & \text{em qualquer outra situação.} \end{cases}$$

(a) Escreva uma função recursiva em Pascal que calcule o mdc de dois números inteiros.
(b) Escreva uma função iterativa em Pascal que realize o mesmo objetivo.

7.6 Exercícios de Programação

EP7.1) (a) Escreva uma função booleana *recursiva* que recebe dois arranjos unidimensionais de um mesmo tipo (escolha!) e retorna **true** se os dois arranjos são iguais. (b) Escreva um programa em Pascal que teste este procedimento.

EP7.2) (a) Um palíndromo é uma palavra que se lê da mesma forma, seja no sentido direto ou no sentido inverso. Escreva uma função booleana *recursiva* que retorne **true** se uma palavra, passada como parâmetro, for um palíndromo. (b) Escreva um programa em Pascal que teste este procedimento. (Limite o tamanho da palavra a 10 caracteres.)

Capítulo 8

PONTEIROS E ESTRUTURAS DINÂMICAS

8.1 Introdução

Os tipos de dados vistos até aqui são denominados **tipos de dados estáticos** porque o espaço reservado para uma variável de qualquer um daqueles tipos não varia durante a execução do programa[32]. Por outro lado, o tamanho de uma variável de um **tipo de dados dinâmico** varia à medida em que o programa é executado, de acordo com as necessidades do mesmo. Uma estrutura de dados dinâmica consiste de uma coleção de elementos chamados **nós** (normalmente estes nós são registros).

Estruturas de dados dinâmicas servem para superar as limitações encontradas nas estruturas estáticas que têm que ser dimensionadas previamente, e portanto, estão sujeitas a subdimensionamento (quando o tamanho previsto deveria na realidade ter sido maior) ou superdimensionamento (quando o tamanho previsto poderia ter sido menor). Considere, por exemplo, o caso de um programa para manutenção de uma lista de espera de uma companhia aérea. Esta lista de espera, contendo dados dos passageiros, poderia ser implementada como um arranjo de registros que teria que ser previamente especificado. Qual deveria então ser o tamanho pré-especificado para esta estrutura estática? A resposta mais óbvia parece ser: este tamanho deveria corresponder ao número máximo de passageiros esperado. Mas, mesmo que fosse possível prever este *máximo*, existiriam situações (por exemplo, quando não há nenhum passageiro na lista de espera durante uma época do ano) onde haveria grande desperdício de memória e tempo para alocação de espaço que, na realidade, não seria necessário. Por outro lado, uma estrutura de dados dinâmica resolveria este problema de dimensionamento pois simplesmente não haveria necessidade para tal dimensionamento prévio: as variáveis representando os registros dos passageiros seriam alocadas à medida em que aparecessem passageiros para ser adicionados à lista. Também, um registro poderia ser retirado da lista e o espaço ocupado pelo mesmo seria liberado. Portanto, uma estrutura assim poderia crescer ou diminuir dependendo das necessidades do programa, como seria desejável no caso de uma lista de passageiros.

[32] Isto inclui strings e registros variantes, embora possa parecer o contrário.

8.2 Variáveis do Tipo Ponteiro

Variáveis do tipo **ponteiro** são utilizadas para a implementação de estruturas de dados dinâmicas[33]. Usualmente, uma variável do tipo ponteiro representa o **endereço** em memória de uma variável que não foi explicitamente declarada no programa. Este tipo de variável que não é declarada é denominada **variável anônima** e, como não existe identificador associado a ela, a mesma pode ser acessada apenas através de ponteiros. Uma declaração de um tipo ponteiro tem a seguinte sintaxe (note a existência do símbolo "^" na declaração abaixo; este símbolo corresponde ao acento circunflexo no teclado):

type <nome do tipo ponteiro> = ^<tipo de dado do elemento>

onde, <*nome do tipo ponteiro*> é o identificador do tipo ponteiro, e <*tipo de dado do elemento*> é o tipo de dados do objeto para o qual o ponteiro ***pode*** apontar. O <*tipo de dado do elemento*> pode ser de qualquer tipo que já tenha sido definido (pelo programador ou predefinido) ou a ser definido posteriormente pelo programador. Note que Pascal permite que se faça referência ao tipo para o qual o ponteiro aponta antes deste último tipo ter sido definido. Esta é a única situação em Pascal na qual uma entidade pode ser referenciada antes de ser definida. Note ainda que nenhum tipo ponteiro é estruturado, mas um ponteiro pode obviamente apontar para um tipo que pode ser estruturado ou não.

Considere, por exemplo, as seguintes declarações:

```
type
   tPassageiro = record
                     nome : string[20];
                     id  :  integer;
                     voo : string[5];
                     valorDaPassagem  :  real
                 end;
   tPonteiroParaPassageiro = ^tPassageiro;

var  meuPonteiro : tPonteiroParaPassageiro;
```

[33] Não faça confusão: ponteiros em si ***não são estruturas de dados dinâmicas***! Eles são apenas um instrumento utilizado na construção de tais estruturas.

No exemplo acima, o tipo `tPonteiroParaPassageiro` é um tipo ponteiro para o tipo `tPassageiro`, enquanto que a variável `meuPonteiro` *pode apontar*[34] para (i.e., conter o endereço de) uma variável (anônima) do tipo `tPassageiro`.

Uma variável anônima não pode ter seu espaço alocado convencionalmente como visto anteriormente, uma vez que a mesma não é declarada no programa. Esta espécie de variável é criada através de instruções explícitas dentro do programa. A instrução que cria uma tal variável anônima é a instrução **new**, que tem a seguinte sintaxe:

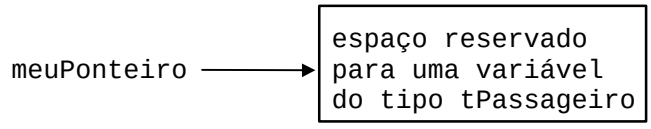
new(<variável do tipo ponteiro>)

onde *<variável do tipo ponteiro>* é uma variável do tipo ponteiro para a variável anônima que se deseja criar. O efeito da instrução **new** é duplo: (1) ela aloca espaço em memória suficiente para conter a variável anônima para qual a *<variável do tipo ponteiro>* irá apontar (i.e., conter o endereço), e (2) ela atribui o endereço desta variável anônima à *<variável do tipo ponteiro>*.

Considere novamente as declarações do exemplo acima juntamente com a instrução:

```
new(meuPonteiro)
```

Esta instrução solicita ao compilador Pascal para fazer o seguinte: (1) reservar espaço em memória para conter uma variável (anônima) do tipo `tPassageiro`, que é o tipo para o qual a variável `meuPonteiro` pode apontar; e (2) atribuir o endereço deste espaço reservado em memória à variável `meuPonteiro`. Esquematicamente, ter-se-ía o seguinte após a execução desta instrução **new**:



O "*espaço reservado para uma variável do tipo `tPassageiro`*" no esquema acima corresponde exatamente à variável anônima que foi criada com a instrução **new**. A seta na ilustração deve ser interpretada como "*contém o endereço de*". Isto é, na mesma

[34] Note a ênfase na expressão "*pode apontar*"; isto significa que, neste ponto do programa, a variável `meuPonteiro` não está ainda apontando para nenhum endereço válido.

ilustração, a seta diz que a variável `meuPonteiro` contém o endereço de um espaço reservado para uma variável do tipo `tPassageiro`.

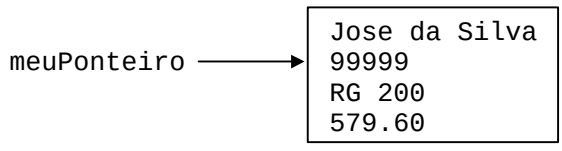
Como uma variável anônima não possui nenhum identificador associado, ela não pode ser referenciada pelo nome como variáveis comuns. Entretanto, cada variável anônima criada através de uma instrução **new** possui um ponteiro apontando para a mesma. Este ponteiro pode ser utilizado para fazer referência à variável anônima para a qual ele está apontando através de um **operador de referenciação** que, em Pascal, é representado pelo símbolo "^"[35]. Uma variável de um tipo ponteiro seguida pelo operador de referenciação "^" (acento circunflexo no teclado) representa o conteúdo (i.e., o valor) de uma variável anônima. Isto é, um ponteiro seguido pelo operador de referenciação (chamado **ponteiro referenciado**) tem o mesmo significado de uma variável do tipo para o qual ele aponta (i.e., pode-se considerar `p^` como sendo a variável para a qual p aponta). Por exemplo, `meuPonteiro^` representa a própria variável anônima criada pela instrução **`new`**`(meuPonteiro)` do exemplo acima. Assim, as instruções abaixo seriam legais:

```
meuPonteiro^.nome := 'Jose da Silva';
meuPonteiro^.voo := 'RG 200';
meuPonteiro^.valorDaPassagem := 534.10;
meuPonteiro^.id := 99999;
meuPonteiro^.valorDaPassagem := meuPonteiro^.valorDaPassagem + 25.50;
```

ou, alternativamente:

```
with  meuPonteiro^  do
       nome := 'Jose da Silva';
       voo := 'RG 200';
       valorDaPassagem := 534.10;
       id := 99999;
       valorDaPassagem := valorDaPassagem + 25.50
end (* with *);
```

A ilustração a seguir representa os valores armazenados nos campos da variável anônima após a execução das instruções acima:



[35] Este símbolo é o mesmo utilizado na declaração de ponteiros; outras linguagens utilizam símbolos diferentes para evitar alguma confusão.

Deve-se observar que a única operação definida sobre ponteiros em Pascal é a **operação de referenciação** (i.e., não se pode, por exemplo, somar ponteiros).

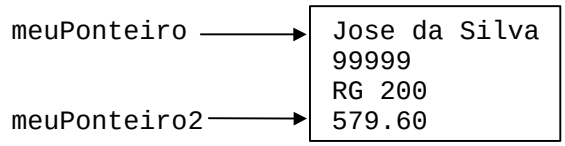
Um aspecto interessante é que dois ponteiros diferentes podem apontar para uma mesma variável anônima. Por exemplo, suponha que, além das declarações do exemplo acima, também se tenha:

```
var   meuPonteiro2 : tPonteiroParaPassageiro;
```

Então, se for feita a atribuição:

```
meuPonteiro2 := meuPonteiro;
```

a variável `meuPonteiro2` passaria a apontar para a mesma posição de memória apontada pela variável `meuPonteiro`. Atribuição entre ponteiros pode apenas ocorrer com ponteiros que apontam para o mesmo tipo de dados. A atribuição do último exemplo poderia ser representado esquematicamente como:



Note que as duas únicas maneiras de se atribuir um endereço válido a uma variável do tipo ponteiro em Pascal são através de instruções de atribuição e de instruções **new**. Da mesma forma que o valor de uma variável comum vista anteriormente tem seu valor (i.e., conteúdo) indeterminado quando a mesma é declarada, variáveis do tipo ponteiro possuem valores (i.e., endereços) indeterminados quando as mesmas são declaradas. É ***extremamente perigoso*** em programação fazer-se referência a um ponteiro ao qual não foi ainda atribuído um endereço válido em memória. As conseqüências podem ser desastrosas se esta regra não for obedecida pois geralmente nenhuma linguagem de programação verifica se um ponteiro contém um endereço válido antes de utilizá-lo. Isto significa que um ponteiro não-inicializado (i.e., ao qual não se atribuiu nenhum endereço válido) pode virtualmente conter o endereço de qualquer porção de memória (inclusive aqueles que não foram designados para a execução de seu programa). A solução para isto (ou a proteção contra acontecimentos desagradáveis) é sempre inicializar ponteiros no início do programa com o valor nulo, que a linguagem Pascal reconhece ser inválido. O valor nulo para um ponteiro é representado em Pascal pela constante predefinida **nil**.

Suponha, por exemplo, que a primeira referência à variável meuPonteiro2, logo após sua declaração, seja a atribuição:

```
meuPonteiro2^.nome := 'Joao da Silva';
```

Como a variável `meuPonteiro2` não estava apontando para nenhum endereço ***conhecido*** neste instante, a execução desta instrução poderia modificar aleatoriamente o valor de uma posição de memória que talvez estivesse sendo utilizada por outro programa ou pelo sistema operacional e o resultado seria imprevisível. Um tal ponteiro é comumente denominado de **ponteiro pendente** (*dangling pointer*, em inglês). Isto poderia ser evitado se o programador tivesse colocado, logo no início do programa, a atribuição:

```
meuPonteiro2 := nil;
```

Neste caso, quando ele tivesse feito acidentalmente a atribuição: `meuPonteiro2^.nome := 'Joao da Silva'`, o compilador Pascal teria acusado o erro.

Outra característica de ponteiros com a qual o programador deve estar bem atento é exemplificada a seguir. Suponha que `p` e `q` são ponteiros para o mesmo objeto. Então, as variáveis (anônimas) `p^` e `q^` são as mesmas (ou, em outras palavras, `p^` e `q^` representam o mesmo objeto). Portanto, uma atribuição a `p^` modifica o valor de `q^`, apesar do fato de nem `q` nem `q^` terem sido mencionados.

8.3 Alocação Dinâmica de Memória

Viu-se na seção precedente como posições de memória podem ser alocadas dinamicamente com o uso de ponteiros e da instrução **new**. Mas, aonde estas posições de memória são alocadas? Com quanto de memória dinamicamente alocada um programa pode contar durante sua execução? Estas e outras questões pertinentes serão respondidas nesta seção.

Pascal mantém um *armazém* de células de memória disponíveis que são alocadas (i.e., separadas para uso) sempre que uma instrução **new** é executada. Este armazém de memória é comumente conhecido como **heap**, e seu tamanho inicial é determinado pelo programador (normalmente através de uma diretiva ou opção do compilador que não faz parte do programa em si). Ter uma quantidade total de memória disponível no *heap* maior do que a quantidade requerida para alocar uma dada porção de memória é uma

condição necessária, mas pode não ser suficiente para a execução bem sucedida de um programa. Ou seja, é preciso ainda que este espaço necessário esteja disponível em ***posições contíguas*** de memória. Por exemplo, suponha que se tenha num programa um tipo registro cuja alocação demande 50 células de memória. Pode ser que, no instante em que se deseje alocar uma porção de memória para um registro deste tipo, o *heap* possua muito mais do que as 50 células necessárias, mas as posições disponíveis de memória podem estar *espalhadas* no *heap* (por exemplo, podem-se ter 10 células contíguas aqui, mais 30 ali, mais 38 acolá, etc., mas nunca as 50 células contíguas necessárias). Isto é chamado de **fragmentação de memória** (ou, mais apropriadamente, **fragmentação de heap**) e ocorre porque, muitas vezes, não existe uniformidade no tamanho dos objetos alocados[36].

Outro fator que pode impedir a alocação dinâmica de memória para um dado objeto é o fato de não se liberar espaço alocado anteriormente quando este não é mais necessário no programa. Isto ocorre porque, quando se faz alocação de memória através da instrução **new**, o espaço alocado torna-se indisponível, mesmo que o programador nunca o utilize efetivamente.

Freqüentemente, num programa, aloca-se espaço apenas para um processamento local temporário e, nestes casos, ***deve-se*** liberar este espaço; caso contrário, o mesmo ficará reservado (indisponível) até o final da execução do programa. Pascal oferece a instrução **dispose** para esta finalidade. A sintaxe da instrução **dispose** é:

dispose(<variável do tipo ponteiro>)

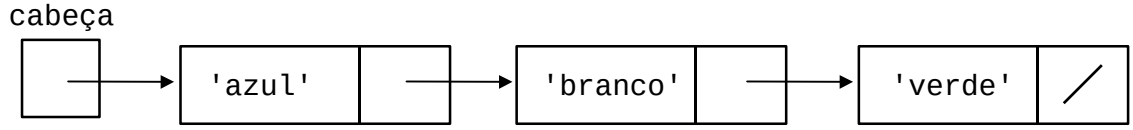
O efeito da instrução **dispose** é o contrário daquele da instrução **new**. Isto é, **dispose** anula o efeito de **new** através da liberação do espaço ocupado cujo endereço é fornecido pela *<variável do tipo ponteiro>* (i.e., o espaço apontado pelo ponteiro torna-se disponível novamente no heap). Constitui-se boa norma de programação utilizar **dispose** para liberar espaço em memória sempre que não se precisa mais dele. Mas tenha cuidado, o uso de **dispose** com um ponteiro que não foi na realidade alocado pode ter efeitos desastrosos (pior: não espere que o compilador identifique este tipo de erro). Após o uso de **dispose** o ponteiro torna-se indefinido. Isto é, após o uso de **dispose** o ponteiro não deve ser utilizado a não ser que o mesmo seja alocado novamente ou

[36] Alguns sistemas adotam mecanismos, denominados **coleta de lixo** ou **garbage collection**, para tentar eliminar os efeitos de fragmentação. A discussão desses mecanismos está além do escopo deste livro.

atribuído uma posição já alocada em memória[37]. (**Discipline-se**: A cada **new** deve corresponder um **dispose**, mesmo que seja no final do programa!)

8.4 Introdução às Listas Encadeadas

Uma **lista encadeada** é uma seqüência de registros, denominados **nós**, os quais são conectados (ou **encadeados**) uns com os outros. Um exemplo de uma lista encadeada com três nós é apresentada a seguir:



Na lista ilustrada no diagrama acima, cada nó (registro) possui dois campos: o primeiro campo é um string e representa o verdadeiro conteúdo (i.e., a informação) contida no nó; o segundo campo, representado por setas é um apontador (i.e., ponteiro) para o próximo elemento na lista. O símbolo "/" no segundo campo do último nó indica que aquele nó não aponta para nenhum outro nó. A **cabeça da lista** é um ponteiro para o primeiro elemento (nó) da lista. Cada elemento da lista pode ser acessado começando-se pela cabeça da lista e seguindo-se os ponteiros para os nós seguintes.

Como ilustração do uso de ponteiros e alocação dinâmica, será examinada a implementação de listas encadeadas como estruturas dinâmicas[38]. O primeiro passo para a criação de uma lista encadeada é a definição do tipo de cada nó da lista. Para assegurar o acesso a todos os elementos da estrutura, é necessário que cada elemento inclua uma indicação de onde o próximo elemento localiza-se em memória. Assim, cada elemento deve consistir de duas partes: a informação que deve ser armazenada no elemento e o endereço em memória do próximo elemento. Portanto, para implementação da lista

---

[37] O verdadeiro efeito da instrução **dispose** não é especificado em Pascal, o que significa que este efeito varia de implementação para implementação da linguagem. O efeito desejado seria que o espaço apontado pelo ponteiro utilizado como parâmetro de **dispose** fosse realmente liberado e o que o próprio ponteiro recebesse o valor **nil**, indicando que o mesmo se tornaria inválido a partir dali. Infelizmente, algumas implementações de Pascal não fazem nem uma coisa nem outra; outras implementações liberam o espaço, mas não atribuem **nil** ao ponteiro. Verifique qual é o efeito de **dispose** na implementação que você está utilizando. Caso o efeito do **dispose** não inclua a atribuição de **nil** ao ponteiro, ou em caso de dúvida, faça esta atribuição explicitamente após cada instrução **dispose**.

[38] Listas encadeadas podem também ser implementadas por meio de arranjos, mas isso não é de interesse aqui.

encadeada ilustrada no exemplo acima, podem-se utilizar as seguintes declarações de tipos:

```
type
   tApontador = ^no;
   string10 = string[10];
   no = record
            dado    :  string10;
            proximo :  tApontador
        end;
```

Existem duas coisas interessantes nas declarações de tipo acima. Primeiro, o tipo `tApontador` é um ponteiro para o tipo `no` que é definido após `tApontador`. Segundo, a definição dos dois tipos é circular, ou seja, `tApontador` faz referência a `no` que por sua vez faz referência a `tApontador`. Numa situação como esta, Pascal ***requer*** que o tipo ponteiro (no caso, `tApontador`) seja declarado primeiro.

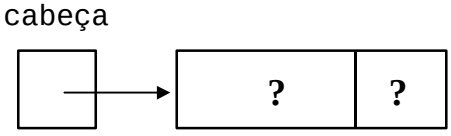
Para criar a lista, são necessárias três variáveis do tipo ponteiro para o tipo `no`: o primeiro ponteiro representa exatamente a cabeça da lista; o segundo e o terceiro ponteiros facilitam a inserção de novos nós na lista como será visto em seguida. Estes ponteiros são declarados abaixo:

```
var
   cabeca, ponteiroAux1, ponteiroAux2 : tApontador;
```

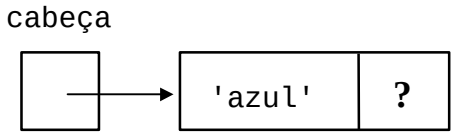
Procede-se agora à criação da lista ilustrada no início desta seção. Os nós serão criados por meio de alocação dinâmica. O primeiro nó (v. ilustração no início desta seção) pode ser criado por meio das seguintes instruções (as instruções foram numeradas para facilitar a discussão que se segue):

```
new(cabeca);                (* 1 *)
cabeca^.dado := 'azul';     (* 2 *)
```

Após a execução da instrução 1, tem-se a seguinte situação (o símbolo “?” indica que o respectivo campo do nó é indeterminado no instante considerado):



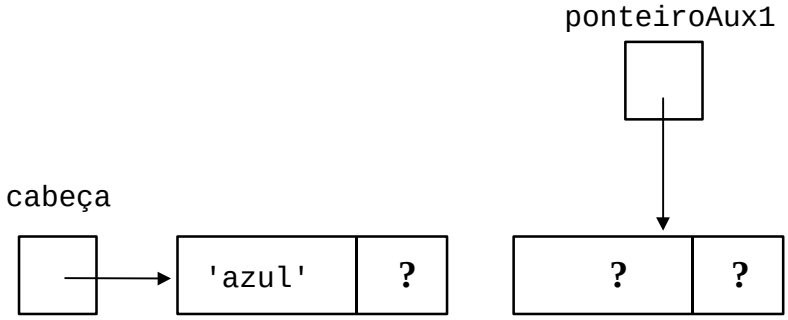
Após a execução da instrução 2, a situação passa a ser a seguinte:



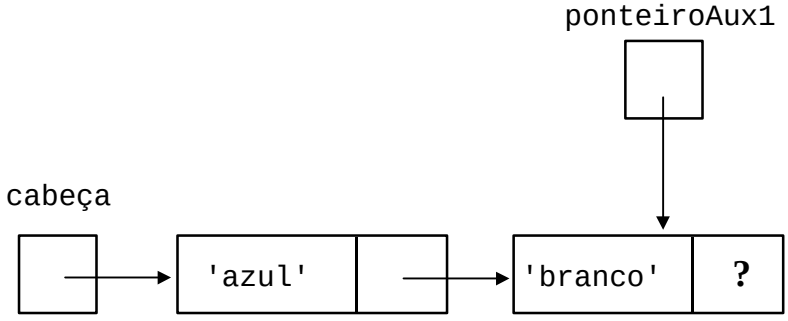
O segundo nó da lista é criado e acrescentado à lista da seguinte maneira:

```
new(ponteiroAux1);                    (* 3 *)
ponteiroAux1^.dado := 'branco';       (* 4 *)
cabeca^.proximo := ponteiroAux1;      (* 5 *)
```

Após a execução da instrução 3 tem-se a seguinte situação (note que o novo nó ainda não está encadeado na lista):



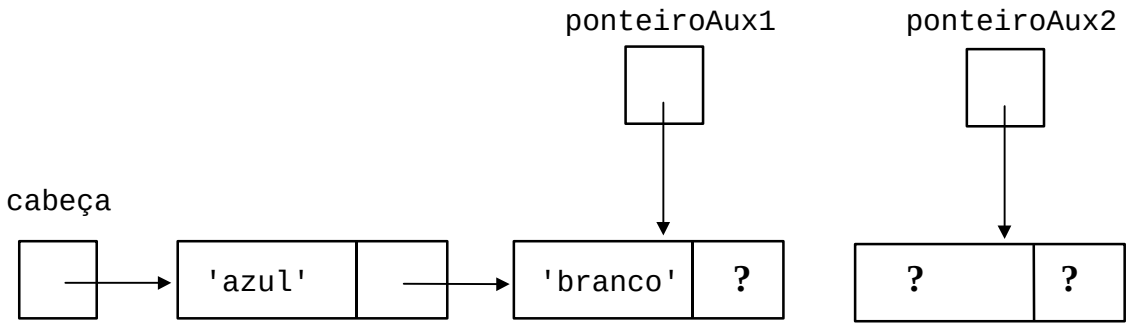
A instrução 4 preenche o valor do primeiro campo do novo nó, enquanto que a instrução 5 faz a devida anexação deste nó à lista. Assim, após a execução da instrução 5 a situação é seguinte:



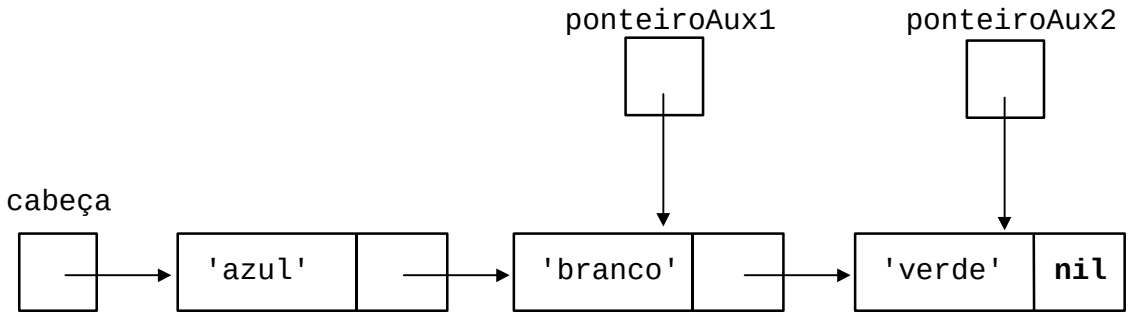
O último conjunto de instruções, responsável pela inserção do último elemento da lista, é:

```
new(ponteiroAux2);                          (* 6 *)
ponteiroAux2^.dado := 'verde';              (* 7 *)
ponteiroAux1^.proximo := ponteiroAux2;      (* 8 *)
ponteiroAux2^.proximo := nil;               (* 9 *)
```

Após a execução da instrução 6 tem-se a seguinte situação (note, novamente, que o novo nó ainda não está encadeado na lista):



A instrução 7 preenche o campo de dado do nó apontado por `ponteiroAux2`, enquanto que a instrução 8 faz a ligação do nó apontado por `ponteiroAux1` para o nó apontado por `ponteiroAux2`. A última instrução simplesmente indica que o último elemento não aponta para nenhum outro nó. A situação final é apresentada a seguir.



Pode parecer que foram utilizados três ponteiros (`cabeca`, `ponteiroAux1` e `ponteiroAux2`) devido ao fato de a lista possuir três elementos, mas isto não corresponde à realidade. Se quiséssemos acrescentar um quarto elemento à lista, não necessitaríamos de um quarto ponteiro auxiliar. Neste caso, o que teríamos que fazer seria o seguinte: (1) alocar o novo nó utilizando o ponteiro `ponteiroAux1`; (2) atribuir os valores dos campos do novo nó (ainda através do ponteiro `ponteiroAux1`); (3)

fazer a ligação do nó apontado pelo ponteiro `ponteiroAux2` para o nó apontado por `ponteiroAux1`.

Exercício: Escreva um conjunto de instruções para acrescentar um quarto nó na lista acima, supondo que o dado contido no novo nó seja o string `'vermelho'`.

Suponha, agora, que você deseje imprimir os valores contidos nos campos `dado` de cada nó da lista encadeada acima. Para fazer isso, você precisaria *percorrer* (ou **atravessar**) toda a lista do início ao final. O procedimento a seguir realiza isto:

```
procedure  ImprimeListaEncadeada(p : tApontador);

begin
    while  (p <>  nil)  do  begin
        writeln(p^.dado);           (* Imprime o valor do nó *)
        p := p^.proximo       (* Avança para o próximo nó *)
    end  (* while *)
end;  (* ImprimeListaEncadeada *)
```

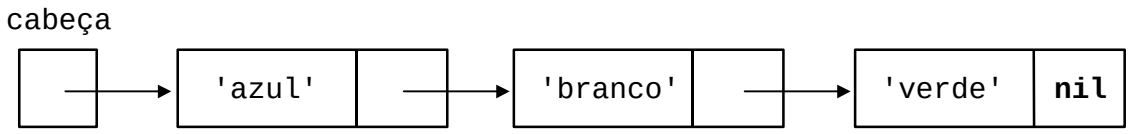
Uma chamada `ImprimeListaEncadeada(cabeca)`, no programa principal, imprimiria todos os valores armazenados nos campos `dado` dos nós da lista apontada por `cabeca`. O funcionamento do procedimento `ImprimeListaEncadeada` é simples. Ele recebe como entrada um ponteiro para uma lista encadeada. Esta lista pode estar vazia, ou conter um ou mais elementos. No primeiro caso, o ponteiro representado pelo parâmetro formal p é **nil** e o laço **while** não é executado. No segundo caso, entra-se no laço **while**, imprime-se o valor do campo `dado` do primeiro elemento, e faz-se `p` apontar para o próximo nó. Se este último tiver o valor **nil**, a iteração pára; caso contrário, repete-se o processo até que `p` assuma eventualmente o valor **nil**.

Exercício: O que aconteceria se o parâmetro `p` do procedimento acima fosse um parâmetro variável e fosse feita a chamada `ImprimeListaEncadeada(cabeca)`, onde `cabeca` aponta para o primeiro nó da lista do exemplo acima? Para onde estaria apontando `cabeca` no final do procedimento?

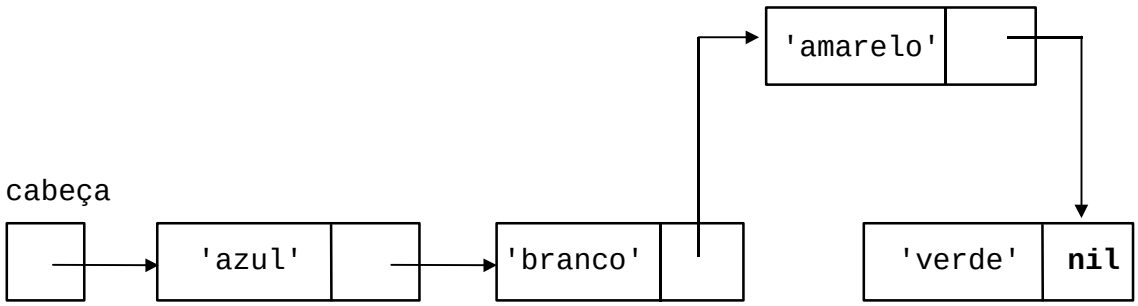
Elementos (*nós*) podem ser inseridos numa lista encadeada de várias maneiras. Pode-se, por exemplo, requerer que um nó seja inserido no início ou no final da lista; ou pode-se desejar que a inserção seja feita imediatamente antes ou imediatamente depois de um nó contendo um determinado valor. Como exemplo, o procedimento apresentado em

seguida pode ser utilizado para inserir um nó imediatamente depois de um nó contendo um determinado valor para uma lista cujos nós são do tipo do exemplo anterior.

Como ilustração do processo, suponha, por exemplo, que se deseje inserir um nó cujo valor do campo `dado` seja 'amarelo' imediatamente depois do nó contendo `'branco'` na lista abaixo:



A situação final deveria então ser a seguinte:

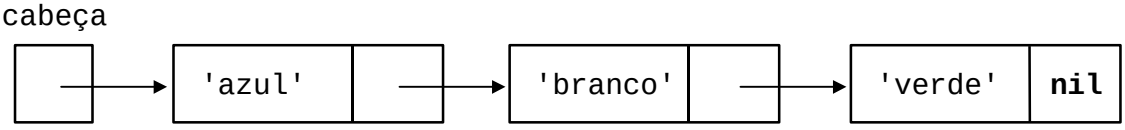


As mudanças feitas na lista para a inserção do nó "`amarelo`" são simples: o nó "`branco`" passa a apontar para o novo nó ("`amarelo`"), que por sua vez passa a apontar para o nó "`verde`". Note, entretanto, que não se pode fazer as novas atribuições de endereços nesta ordem, pois, se o endereço do nó "`amarelo`" for atribuído ao campo `proximo` do nó "`branco`" antes da atribuição do endereço do nó verde ao campo `proximo` do nó "`amarelo`", o nó "`verde`" estará perdido para sempre. (No diagrama, isto significa que se você desenhar primeiro a seta do nó "`branco`" para o nó "`amarelo`", você não será nunca capaz de desenhar a seta do nó "`amarelo`" para o nó "`verde`", simplesmente porque não saberá aonde o nó "`verde`" se encontra. Convença-se de que realmente entendeu isso!)

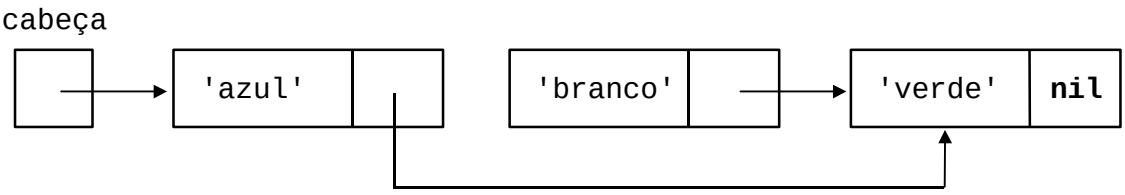
```
procedure InsereDepois(p : tApontador; stringNovo,stringNaLista : string10);
(* Insere um novo nó cujo conteúdo é stringNovo após o nó cujo conteúdo é *)
(* stringNaLista na lista apontada por p (cabeça da lista)                *)
var
   posicaoDoNo, noNovo : tApontador;

begin
    posicaoDoNo := nil;
    while  (p <> nil)  and  (posicaoDoNo = nil)  do  (* Procura o nó *)
         if  (p^.dado = stringNaLista)  then
             posicaoDoNo := p           (* Nó procurado foi encontrado *)
         else
              p := p^.proximo;          (* Passa para o próximo nó *)

        if (posicaoDoNo <> nil) then begin (* posicaoDoNo foi encontrado e*)
                                        (* aponta para o nó após o qual *)
                                        (* a inserção será feita. *)
          new(noNovo);    (* Aloca o novo nó *)
          noNovo^.dado := stringNovo;  (* Preenche conteúdo do novo nó *)
          noNovo^.proximo := posicaoDoNo^.proximo; (* Faz novo nó apontar *)
                                                   (* para o nó apontado *)
                                                (*antes pelo nó encontrado *)
          posicaoDoNo^.proximo := noNovo; (* Faz nó encontrado apontar *)
                                             (*para  nó  novo  *)
        end (* if *)
end;  (* InsereDepois *)
```

Exercício: Transforme o procedimento acima numa função booleana que retorna **true** se a inserção foi bem sucedida (i.e., se o nó foi realmente inserido na lista) e **false** em caso contrário.

Como último exemplo, o procedimento a seguir remove um nó, cujo conteúdo é especificado, de uma lista do tipo apresentado acima. Como ilustração, suponha que se deseje remover o nó cujo valor do campo `dado` seja `'branco'` na lista a seguir:



A situação final deveria então ser a seguinte:



Conforme pode ser observado nesta última ilustração, a remoção do nó desejado envolve apenas o desvio do endereço do campo `proximo` do nó anterior àquele sendo removido para o nó que era anteriormente apontado pelo nó a ser removido. Na prática, deve-se também liberar o espaço reservado para o nó removido, mas em princípio, o desvio representado pelo esquema acima é suficiente para a remoção do nó. O procedimento a seguir produz o efeito de remoção desejado:

```
procedure   RemoveNo(var  p : tApontador; stringNaLista : string10);
(* Remove o primeiro nó encontrado cujo conteúdo é stringNaLista na *)
(* lista apontada por p (cabeça da lista).                          *)

var
   noAnterior, posicaoDoNo : tApontador;
   noFoiEncontrado : boolean;

begin
   posicaoDoNo := p;       (* Começa a busca a partir da cabeça da lista *)
   noAnterior := nil;      (* No início da lista não existe nó anterior  *)
   noFoiEncontrado := false; (* Indica que nó ainda não foi encontrado  *)

   while (posicaoDoNo <> nil) and (not  noFoiEncontrado) do (* Procura nó *)
         if   (posicaoDoNo^.dado = stringNaLista)   then
           noFoiEncontrado:= true (* Nó procurado foi encontrado *)
         else  begin
          noAnterior := posicaoDoNo; (*Nó anterior passa a ser o atual e...*)
          posicaoDoNo := posicaoDoNo^.proximo (* atual passa a ser próximo *)
         end;    (* if *)

   if  noFoiEncontrado  then
       if (noAnterior = nil) then (* Nó a ser removido é 1o. da lista *)
          p := p^.proximo   (* O 1o. nó agora será o antigo 2o. nó *)
       else (noAnterior <> nil) (* Nó a ser removido NÃO é o primeiro *)
          noAnterior^.proximo := posicaoDoNo^.proximo; (* Faz o desvio *)

       (* Neste ponto, o nó apontado por posicaoDoNo não mais pertence  *)
       (* à lista e  a remoção está logicamente concluída. Resta apenas *)
       (* liberar o espaço ocupado pelo nó removido.              *)

   dispose(posicaoDoNo); (* Libera alocação de memória do nó removido *)
end;
```

Note que a remoção do primeiro nó da lista é um caso que deve ser tratado à parte pelo procedimento acima. Note ainda que se houver na lista mais de um nó com o mesmo conteúdo especificado para remoção, apenas o primeiro nó com este conteúdo será removido utilizando-se o procedimento acima.

Exercícios:

1. Por que o parâmetro `p` no procedimento `RemoveNo` é um parâmetro variável, enquanto que no procedimento `InsereDepois` o parâmetro `p` foi definido como um parâmetro de valor? O parâmetro p no procedimento `RemoveNo` poderia ser um parâmetro de valor? E o parâmetro `p` no procedimento `InsereDepois` poderia ser um parâmetro variável?

2. Modifique o procedimento `RemoveNo` acima de modo que ele seja capaz de remover todas as ocorrências de nós com o conteúdo especificado, e não apenas o primeiro nó encontrado.

8.5 Exercícios de Revisão

1. O que é uma estrutura de dados dinâmica? Diferencie estruturas de dados dinâmica e estática.

2. Que espécie de valor pode estar contido numa variável do tipo ponteiro?

3. Para que serve o operador de referenciação "^" em Pascal?

4. (a) O que é uma variável anônima? (b) Como variáveis anônimas podem ser acessadas?

5. Para que serve a instrução **new** em Pascal?

6. Qual é o significado da palavra reservada **nil** em Pascal?

7. Por que a inicialização de ponteiros é importante?

8. O que significa *fragmentação de memória* (ou, mais precisamente, *fragmentação de heap*)?

9. Por que é importante utilizar a instrução **dispose** no programa para liberar uma porção de memória alocada dinamicamente quando não mais se precisa dela?

10. Suponha que você tenha um ponteiro como variável local de um procedimento em Pascal. Se você alocar espaço dinamicamente utilizando este ponteiro em conjunção com a instrução **new**, este espaço alocado dinamicamente será automaticamente liberado quando do retorno do procedimento? Explique.

11. (a) Defina lista encadeada. (b) Como ponteiros são utilizados para estabelecer ligações entre os nós? (c) Para que serve a cabeça de uma lista encadeada?

8.6 Exercícios de Programação

EP8.1) Escreva uma função em Pascal que retorne o comprimento (i.e., o número de nós) de uma lista simplesmente encadeada L, onde L é um ponteiro para o primeiro nó da lista e o tipo de cada nó é aquele definido na *Seção 8.4*. O campo **proximo** do último nó tem valor **nil**. A função deverá ter o seguinte cabeçalho:

```
function   ComprimentoDeListaEncadeada(L : tApontador) : integer;
```

EP8.2) Escreva um procedimento, denominado `InsereAntes`, em Pascal para inserir um nó imediatamente antes de um nó contendo um determinado valor (do tipo `string10`) para uma lista cujos nós são do tipo do exemplo da *Seção 8.4*.

EP8.3) (a) Escreva um programa em Pascal que crie a lista de três nós do exemplo da *Seção 8.4*. (b) Utilize, em seu programa, o procedimento `ImprimeListaEncadeada` encontrado neste capítulo para imprimir os conteúdos dos nós da lista. (c) Utilize o procedimento `ComprimentoDeListaEncadeada` (**EP8.1**) para calcular e imprimir o comprimento desta lista. (d) Utilize o procedimento `InsereAntes` (**EP8.2**) para inserir um nó cujo conteúdo é o string 'preto' antes do nó contendo o string `'branco'`. (e) Utilize o procedimento `RemoveNo` encontrado na *Seção 8.4* para remover o nó cujo conteúdo é o string `'verde'`.

Apêndice A

NOÇÕES DE SISTEMAS OPERACIONAIS

A.1 Introdução

Para que se possa programar efetivamente é necessário que se conheça bem o **sistema operacional** do computador assim como o próprio **ambiente de programação** com o qual se irá trabalhar. Os apêndices A e B são dedicados ao conhecimento elementar destes assuntos para que o aluno menos experiente com sistemas computacionais possa a partir daqui estar apto a escrever e executar seus programas.

A.2 O Que É Um Sistema Operacional?

Um sistema operacional é conjunto de programas básicos que controlam o *hardware* do computador. O *hardware* dota o computador de poder computacional em estado bruto, que é praticamente inútil se não existir um sistema operacional capaz de gerenciá-lo. Sistemas operacionais são responsáveis, entre outras coisas, pelo gerenciamento dos seguintes recursos: memórias principal e secundárias, dispositivos periféricos de entrada/saída, e dispositivos de comunicação. Eles também são responsáveis pela execução de inúmeras funções, dentre as quais podem-se enumerar: implementação da interface com o usuário, compartilhamento de recursos de hardware entre usuários em sistemas multi-usuários, gerenciamento de comunicações em redes de computadores, segurança de dados, recuperação de erros, e estatística de uso do sistema.

Como exemplos de sistemas operacionais podem ser citados os seguintes:

- *Disk Operating System* (***DOS***) - usado pela maioria dos computadores pessoais do mundo; desenvolvido pela Microsoft.

• **UNIX** - Desenvolvido pela AT&T e utilizado principalmente em estações de trabalho, mas tem implementações para virtualmente todas as plataformas[39].

• **OS/2** - Desenvolvido pela IBM/Microsoft para uso na linha PS/2 da IBM.

• **VM** - Desenvolvido pela IBM para uso em sua linha de *mainframes*.

• **MacOS** - Desenvolvido pela Apple para uso em sua linha de computadores *Macintosh*.

• **Windows95** - Desenvolvido pela Microsoft para uso em computadores pessoais (PCs)[40].

O desenvolvimento de programas em sistemas operacionais antigos (desenvolvidos até meados da década de 60), denominados **sistemas de batch**, era uma tarefa muito lenta e árdua. Normalmente, o usuário não estava presente nas instalações computacionais aonde seus programas deveriam ser compilados e executados. Os programas eram tipicamente submetidos em cartões perfurados e permaneciam à espera para serem carregados no computador. Esta espera poderia significar vários dias. Ainda pior: um programa que contivesse um erro, por mais insignificante que fosse este erro (por exemplo, um ponto-e-vírgula ausente), causava uma frustração adicional para o programador que teria que corrigir o erro (perfurando um novo cartão) e submeter novamente o programa (que teria novamente que esperar para ser compilado e executado). O aparecimento de sistemas operacionais de **tempo compartilhado** (***timesharing systems***) no final da década de 60 representou uma grande evolução nesse sentido para o programador. O tempo decorrido entre a submissão de um programa e o retorno do resultado foi reduzido para minutos (ou mesmo segundos). O programador podia digitar seu programa, compilá-lo, receber uma lista de erros sintáticos (se fosse o caso) corrigir estes erros, e recompilar o programa até que o mesmo estivesse livre de erros sintáticos. Então, o programa podia ser executado e testado até que o programa estivesse (idealmente) livre de erros.

Do ponto-de-vista de programação de alto nível, sistemas operacionais livram o programador da preocupação com detalhes de manipulação de hardware, tais como

---

[39] Como "*UNIX*" é marca registrada muitas implementações deste sistema aparecem com nomes diferentes como, por exemplo, *POSIX* e *LINUX*.

[40] Deve-se observar que versões anteriores do Windows não são consideradas sistemas operacionais, mas simplesmente programas de interface gráfica (**shells**) que tentam imitar a interface do Macintosh.

gerenciamento de memória, e execução de entrada/saída. Um sistema operacional geralmente oferece várias **chamadas de sistema** que o programador utiliza para efetuar manipulação de hardware. Sistemas mais avançados também oferecem **chamadas de sistema de alto nível** para manipulação de interface. Por exemplo, com um pequeno número de chamadas de sistema do **MacOS**, o programador pode apresentar uma janela no seu programa com as características desejadas; caso o sistema não oferecesse esta facilidade (ou o programador a desconhecesse), o programador, certamente, teria muito mais trabalho para apresentar sua janela (i.e., ele teria que dimensionar e instruir o computador a desenhar os vários componentes da janela, estabelecer o funcionamento de seus controles, e muitos, muitos outros detalhes de implementação necessários para a construção de uma janela). Por isso, é importante que o programador seja ciente das facilidades oferecidas pelo sistema operacional com o qual está trabalhando[41]. Entretanto, visto que este é um livro de programação introdutório, os programas vistos aqui não requerem conhecimento destas chamadas (na realidade, o compilador Pascal traduz algumas instrução da linguagem em chamadas de sistema).

A.3 O Disk Operating System (DOS)

Devido à sua popularidade, o sistema operacional **DOS** será visto brevemente aqui com alguns detalhes práticos. O primeiro aspecto importante para entender o funcionamento do **DOS** é a forma como ele organiza os arquivos no computador (ou, mais precisamente, num disco rígido/flexível do computador).

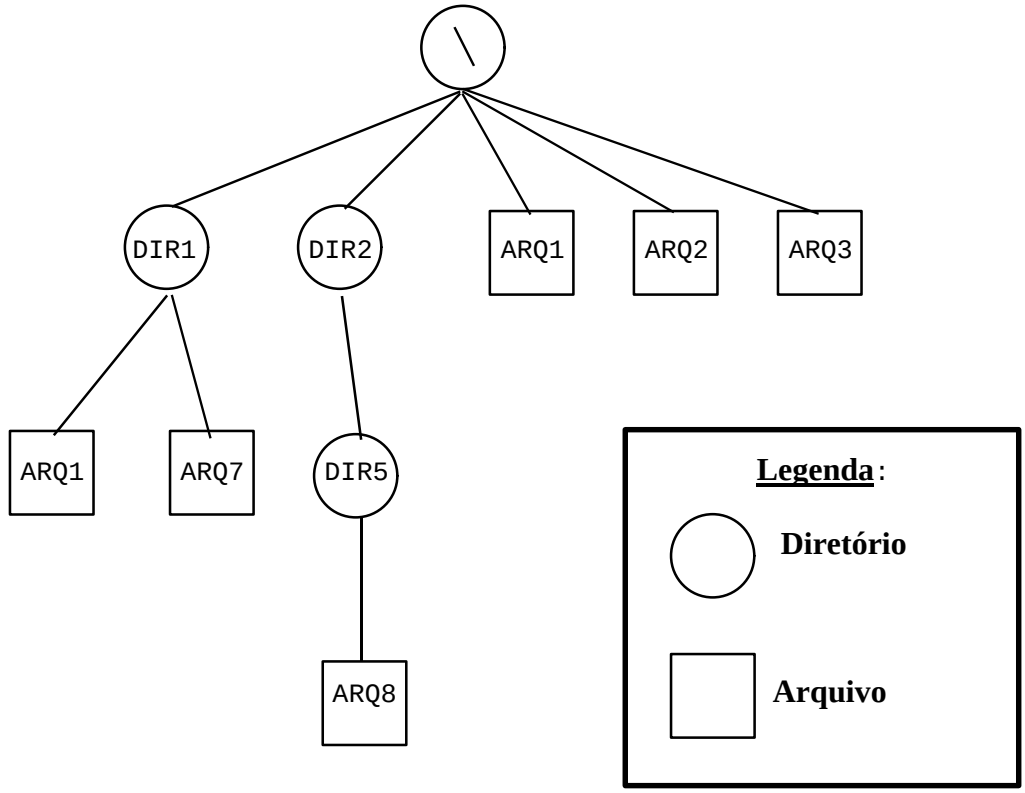
A.3.1 Estrutura de Arquivos

O DOS organiza arquivos por meio de uma **estrutura hierárquica**. Esta estrutura de organização pode ser dispensável quando o meio de armazenamento tem baixa capacidade, mas é essencial para facilitar a localização (busca) de arquivos em unidades de armazenamento com capacidade de armazenar milhares de arquivos, como as unidades de disco rígido atuais. Na estrutura hierárquica do DOS, o volume do disco é dividido em diretórios, cada diretório pode conter arquivos ou outros diretórios, e assim por diante. O próprio volume do disco é considerado um diretório - denominado **diretório raiz**. Cada arquivo possui um nome constituído por, no máximo, oito caracteres alfanuméricos; o primeiro caractere deve ser uma letra. Pode-se ainda

[41] Isto nem sempre é uma tarefa fácil; para se ter uma idéia, o **MacOS** possui no momento mais de quatro mil chamadas de sistema, e este número aumenta a cada versão nova do sistema.

acrescentar uma **extensão** ao nome de um arquivo. A extensão pode conter no máximo três caracteres alfanuméricos. Dois arquivos num mesmo disco podem possuir um mesmo nome, desde que estes arquivos estejam localizados em diferentes diretórios.

Um arquivo é *unicamente* identificado por um **caminho** (***pathname***) seguido do nome do arquivo. O caminho é constituído pelos diretórios que devem ser atravessados desde a raiz até que o arquivo seja alcançado. Um exemplo ilustra melhor estes conceitos. Considere a estrutura de arquivos esquematizada na figura a seguir:



Na estrutura de arquivos esquematizada acima, "\" representa o diretório raiz. Subordinados à raiz, existem dois diretórios (`DIR1` e `DIR2`) e três arquivos (`ARQ1`, `ARQ2` e `ARQ3`). O diretório `DIR1` possui dois arquivos subordinados a ele, enquanto que o diretório `DIR2` possui um outro diretório (`DIR5`) sob sua subordinação. O diretório `DIR5` possui apenas um arquivo (`ARQ8`). Como se poderia referenciar o arquivo `ARQ8` a partir do diretório raiz, ou em outras palavras, qual é o **nome absoluto** do arquivo `ARQ8`? O nome absoluto de um arquivo é composto por seu *pathname* seguido de seu próprio nome, e é formado da seguinte maneira: o nome do *drive* aonde se encontra o arquivo, seguido de barra invertida ("\"), seguida dos diretórios que vão desde a raiz até

o arquivo, estes últimos também são separados por barras invertidas. Assim, supondo que o arquivo `ARQ8` está localizado no drive `C`, tem-se que seu nome absoluto é dado por:

```
C:\DIR2\DIR5\ARQ8
```

Se o diretório corrente fosse aquele que contém o arquivo `ARQ8`, não seria necessário escrever o nome absoluto para referir-se ao arquivo, o nome do arquivo, que é o seu **nome relativo** a este diretório, seria suficiente.

A.3.2 Comandos Essenciais do DOS

Esta seção apresenta alguns comandos essenciais do DOS, mas não pode ser considerado como um manual de DOS.

Existem dois tipos de comandos no DOS: (1) **internos**, e (2) **externos**. Os comandos internos são mantidos em memória e processados pelo processador de comandos contido no arquivo "`COMMAND.COM`"; este arquivo é carregado automaticamente quando o DOS inicia. Comandos externos são processados por outros arquivos mantidos em disco e carregados quando são necessários. O arquivo `COMMAND.COM` também é responsável pela execução de programas. Programas executáveis no DOS devem ter uma das seguintes extensões: "`.COM`", "`.EXE`" ou "`.BAT`".

Manipulação de Discos

- **FORMAT** - formata um disco para ser usado com o DOS.

  Exemplo: `FORMAT A:` - formata um disco contido numa unidade de disco identificada pela letra `A`.

- **LABEL** - Modifica, cria ou apaga o rótulo de um disco.

  Exemplo: `LABEL A:` - quando este comando é digitado, o DOS apresenta o seguinte ao usuário:

```
O volume da unidade A não tem nome
Nome de volume (11 caracteres, ENTER para nenhum)?
```

Você pode então digitar o nome desejado para o disco na unidade `A` (com o máximo de 11 caracteres).

• **SYS** - Copia arquivos de sistema para o disco

Exemplo: `SYS A:` - copia os arquivos de sistema para o disco `A`, que poderá então ser utilizado para iniciar o computador.

• **VOL** - Apresenta o rótulo de um disco.

Exemplo: `VOL A:` - quando este comando é digitado, o DOS apresenta uma mensagem informando ao usuário o rótulo do disco especificado, como:

```
O volume da unidade A é MEUDISCO
```

Manipulação de Diretórios

• **CD** - Torna o diretório especificado o diretório corrente.

Exemplo: `CD DIR1` - quando este comando é digitado, o DOS passa a considerar o diretório `DIR1` como sendo o diretório corrente.

• **DIR** - Apresenta informação sobre o conteúdo do diretório especificado. Se nenhum diretório for especificado, apresenta informação sobre o diretório corrente.

Exemplo: `DIR A:` - quando este comando é digitado, o DOS apresenta conteúdo do disco `A`, como:

```
ARQ1                 86   03/09/96   17:50
DIR1       <DIR>          04/09/96   12:14
DIR2       <DIR>          04/09/96   14:06
ARQ2                168   03/09/96   18:10
ARQ3                425   03/09/96   19:40
```

Na primeira coluna, tem-se o nome do diretório ou arquivo; no caso de diretório, a segunda coluna indica isto por meio de `<DIR>`; a terceira coluna indica o tamanho de arquivos em *bytes*; a quarta e a quinta colunas indicam a data e a hora da última modificação de cada arquivo ou diretório.

• **MD** - Cria um novo subdiretório.

Exemplo: `MD DIR9` - quando este comando é digitado, o DOS cria um diretório denominado `DIR9` subordinado ao diretório corrente.

• **RD** - Remove um diretório *vazio*. (Se o diretório que se deseja remover não estiver vazio, ele deve ser esvaziado antes de este comando poder ser utilizado.)

Exemplo: `RD DIR9` - quando este comando é digitado, o DOS remove o diretório denominado `DIR9` subordinado ao diretório corrente (supondo que o mesmo está vazio).

Manipulação de Arquivos

Todos os comandos para manipulação de arquivos vistos a seguir aceitam o uso de **caracteres curinga**. Estes caracteres, representados por "*" (asterisco) e "?" (ponto de interrogação) podem substituir quaisquer caracteres e são úteis, por exemplo, quando se deseja realizar uma operação sobre vários arquivos com denominações semelhantes ou quando se conhece apenas parte do nome de um arquivo. Os caracteres "*" e "?" têm interpretações diferentes: o caractere "*" pode substituir qualquer número de caracteres (inclusive 0), enquanto que o caractere "?" pode substituir apenas um caractere. Por exemplo, uma especificação de arquivo como: `arq*` representa qualquer arquivo que começa com as letras "`arq`" (por exemplo, `arq`, `arq15`, `arquivo1.txt`). Entretanto, a especificação: `arq?` representa apenas aqueles arquivos que começam por "`arq`", mas seguidos apenas por mais um caractere (por exemplo, `arq1`, `arq5`).

• **COPY** - Copia um ou mais arquivos. Se a cópia for feita para o mesmo diretório, deve-se especificar um nome diferente para a cópia.

Exemplo: `COPY ARQ1 A:` - quando este comando é digitado, o DOS copia o arquivo `ARQ1` no diretório corrente para o disco `A`.

• **DELETE** - Apaga um arquivo. (Pode-se utilizar ainda **DEL** ou **ERASE** para o mesmo propósito.)

Exemplo: `DELETE ARQ1` - quando este comando é digitado, o DOS apaga o arquivo `ARQ1` no diretório corrente para o disco `A`.

• **RENAME** - Renomeia um arquivo. (**REN** pode também ser utilizado.)

Exemplo: `RENAME ARQ1 MEUARQ` - quando este comando é digitado, o DOS renomeia o arquivo denominado `ARQ1` no diretório corrente para `MEUARQ`.

Manipulação de Entrada e Saída

• **CLS** - Limpa a tela do computador.

• **MORE** - Apresenta o conteúdo de um arquivo, uma tela de cada vez.

Exemplo: `MORE ARQ1` - quando este comando é digitado, o DOS apresenta, na tela, o conteúdo do arquivo-texto denominado `ARQ1` no diretório corrente. Se este conteúdo ocupar mais de uma tela, a apresentação será interrompida a cada tela preenchida à espera que o usuário comande uma nova página (i.e., tela).

• **PRINT** - Imprime um arquivo-texto.

• **TYPE** - Envia um arquivo para a saída padrão. Idêntico a **MORE** quando a saída padrão é a tela, com a exceção de que se o conteúdo do arquivo-texto ocupar mais de uma tela, a apresentação *não* será interrompida.

Apêndice B

USANDO TURBO PASCAL

B.1 O Ambiente Turbo Pascal

Para que um programa escrito em linguagem de alto nível, como Pascal, possa ser executado é necessário que este programa seja antes traduzido para linguagem de máquina por meio de um interpretador ou compilador. O Turbo Pascal possui esta capacidade de tradução, mas ele oferece mais do que um simples compilador ou interpretador. Na realidade, Turbo Pascal é um **ambiente de programação** ou **ambiente de desenvolvimento** (**de software**). Como tal, além de um compilador Pascal, o Turbo Pascal oferece as seguintes facilidades adicionais que ajudam o programador a desenvolver seus programas em Pascal:

- **Editor Orientado por Sintaxe**, que, diferentemente de outros editores ou processadores de texto convencionais, é dotado de conhecimento sobre a sintaxe da linguagem Pascal e utiliza este conhecimento para, entre outras coisas, fazer uma verificação prévia de sintaxe do programa digitado pelo programador, endentar (semi-) automaticamente o programa, realçar palavras reservadas de Pascal, etc. Deve-se notar que este editor não é um editor comum: é um editor *especializado em Pascal*. Por isso, ele facilita a *escrita de programas em Pascal*; por outro lado, este editor não é prático, por exemplo, para a escrita de textos comuns ou programas em *C*.

- **Carregador de Programas**, que permite que um programa recentemente digitado seja compilado e executado sem que o programador saia do ambiente Turbo Pascal.

- **Facilidades para Depuração**, que auxiliam o programador a testar e depurar erros em seus programas.

- **Editor de Ligações** para fazer as ligações entre as diferentes partes de programas grandes contidas, usualmente, em vários arquivos componentes.

- **Ajuda On-line**, com a qual o programador pode tirar dúvidas a respeito da sintaxe de instruções Pascal ou sobre o próprio ambiente Turbo Pascal.

Além das facilidades apresentadas acima, existem ainda outras facilidades menores que não serão abordadas aqui. Este apêndice apresenta o mínimo de conhecimento necessário para executar-se um programa em Turbo Pascal. O conhecimento de características mais avançadas do ambiente de programação poderá ser adquirido consultando o manual de Turbo Pascal[42].

Quando você inicia uma sessão com o Turbo Pascal, o editor é acionado e um novo documento é iniciado (com um nome inicial, como "`NONAME00.PAS`", atribuído pelo Turbo Pascal). Você pode então começar a digitar seu programa (que, idealmente, já foi escrito à mão após todas aquelas etapas descritas na *Seção 2.4*).

O Turbo Pascal apresenta uma barra de menus na parte superior da tela com o formato mostrado abaixo:

File	**Edit**	**Search**	**Run**	**Compile**	**Debug**	**Tools**	**Options**	**Window**	**Help**

Cada um destes menus agrupa um conjunto de ações com características semelhantes (por exemplo, o menu **File** agrupa ações, tais como abrir e imprimir, relacionadas a tratamento de arquivos). Para escolher uma opção faça o seguinte: (1) aponte o ponteiro do *mouse* para o menu contendo a opção desejada; (2) clique o mouse para abrir o menu; (3) mova ponteiro do mouse para cima ou para baixo até atingir a opção desejada; e, finalmente, (4) clique sobre a opção desejada. Estes menus, bem como suas opções mais importantes para o desenvolvimento de programas simples, serão brevemente descritos a seguir.

[42] A versão de Turbo Pascal descrita aqui é a 7.0 sendo executado sob *DOS* (e não com *Windows*). Outras versões podem oferecer outras facilidades ou modificações que não são apresentadas aqui, mas o funcionamento básico do ambiente não deve sofrer mudança substancial de uma versão para outra. Entretanto, se estudante estiver utilizando outra versão de Turbo Pascal, ele deve consultar o manual para verificar as diferenças entre sua versão e aquela apresentada aqui.

B.2 Menu File

O menu **File** contém ações relacionadas ao tratamento de arquivos. As principais opções deste menu são:

New. Quando escolhida, esta opção cria um novo arquivo no qual você pode editar um novo programa.

Open. Esta opção permite-o abrir um arquivo-fonte existente em disco para que você possa modificá-lo ou simplesmente examiná-lo. Quando você escolhe esta opção, o Turbo Pascal apresenta uma espécie de "*mapa de navegação*" para facilitar a localização do arquivo que você pretende abrir.

Save. Esta opção deve ser utilizada para salvar (i.e., guardar permanentemente) um arquivo em disco. Na primeira vez que você utiliza esta opção com um arquivo novo, o Turbo Pascal solicita que você informe o nome do arquivo. Você precisa apenas especificar o nome principal do arquivo, pois a extensão ("`.PAS`") é acrescentada automaticamente[43].

Print. Utilizando esta opção você pode imprimir o arquivo (programa-fonte) corrente na impressora.

Exit. Utilize esta opção para sair do Turbo Pascal.

B.3 Menu Edit

O menu **Edit** contém ações que visam facilitar as tarefas de edição de programas. As opções disponíveis neste menu são:

Undo. Desfaz a última ação de edição executada. Por exemplo, se você apagou uma palavra do programa por descuido, você pode desfazer esta ação (i.e., recuperar a palavra apagada) utilizando esta opção.

Redo. Refaz a última ação de edição que foi desfeita. Por exemplo, se você *desfez* a ação de apagar uma palavra como no exemplo anterior, você pode

[43] Utilize nomes de arquivos que sejam significativos, i.e., que sugiram os seus conteúdos; esta estratégia o fará ganhar tempo quando estiver buscando um determinado arquivo escrito algum tempo atrás.

refazer esta ação (i.e., apagar novamente a palavra apagada antes de usar **Undo**) utilizando esta opção.

Cut. Utilizando esta opção você pode *cortar* um bloco de texto com o objetivo de colá-lo posteriormente em outro local (no mesmo arquivo ou em outro arquivo). Antes de cortar um bloco é preciso primeiro delimitar o início e o final do bloco desejado. O início de um bloco é marcado utilizando a seqüência de teclas `CTRL + K + B` (isto significa: mantenha a tecla `CTRL` pressionada enquanto digita as teclas `K` e `B`); o final de um bloco é marcado através da seqüência de teclas `CTRL + K + K`[44].

Copy. Esta opção também copia um bloco para colá-lo posteriormente em outro local. A diferença entre **Copy** e **Cut** é que **Cut** apaga o bloco marcado, enquanto **Copy** não apaga. Em outras palavras, **Copy** é utilizado normalmente para *duplicar* um bloco, enquanto que **Cut** é utilizado para *mover* um bloco.

Paste. Esta ação serve para colar um bloco anteriormente copiado ou cortado (utilizando **Copy** ou **Cut**, respectivamente) para o local aonde se encontra o cursor.

Clear. Serve para apagar um bloco inteiro. (**Cuidado**: Não confunda **Clear** com **Cut**: se você utilizar **Clear**, você não poderá *colar* o bloco, embora você possa desfazer a ação utilizando **Undo**.)

B.4 Menu Search

O menu **Search** contém ações que facilitam o gerenciamento de programas grandes. Não são usualmente necessários para programas pequenos requeridos num curso introdutório como este.

[44] A versão 7.0 de Turbo Pascal também permite que você faça esta marcação *arrastando* uma seleção com o mouse pressionado.

B.5 Menu Run

O menu **Run** contém ações que controlam a execução do programa. Principais opções:

Run. Esta opção executa o programa. Note entretanto que, como o programa é executado dentro do Turbo Pascal, se houver saída para a tela do computador, esta saída poderá ser tão rápida que se tornará muito difícil de ser lida. Neste caso, utilize a opção **Output** ou **User Screen** do menu **Debug** (v. mais adiante).

Step Over. Esta instrução executa a instrução corrente. A cada instrução executada, o programa pára e espera que o usuário comande o recomeço da execução. Uma chamada de procedimento ou função é considerada como uma instrução qualquer da linguagem; isto significa que o programa *não entra* e pára a cada execução de uma instrução que esteja dentro de um procedimento ou função. Esta opção é útil quando se deseja examinar cuidadosamente o funcionamento do programa como um todo (i.e., quando você tem certeza que alguns procedimentos e funções utilizados pelo programa funcionam corretamente).

Trace Into. Esta opção é idêntica à opção anterior, mas se a instrução corrente for uma chamada de procedimento ou envolver uma chamada de função, a próxima instrução a ser executada será a primeira instrução dentro do procedimento ou função. Esta opção é útil quando se deseja examinar cuidadosamente o funcionamento do programa inclusive os procedimentos e funções utilizados pelo programa.

Go To Cursor. Esta opção executa o programa até o ponto delimitado pelo cursor na tela, e é útil para depuração e teste de programas. Por exemplo, suponha que você tenha certeza que até um certo ponto seu programa está *perfeitamente correto*. Você pode então utilizar esta opção para executar *normalmente* o trecho inicial de programa que você tem certeza que está funcionando bem e, a partir do ponto aonde pairam dúvidas de funcionamento, você pode utilizar as opções **Step Over** ou **Trace Into** para examinar mais detidamente o funcionamento das partes seguintes do programa.

B.6 Menu Compile

O menu **Compile** contém as seguintes opções:

- **Compile**. Serve justamente para compilar (mas não executar) o arquivo corrente. Se você quiser executar um programa logo após editá-lo, você precisa apenas escolher **Run**, pois esta opção automaticamente compila o programa antes de executá-lo. Se o seu programa for constituído por mais de um arquivo e você utilizar esta opção para compilar um destes arquivos, será gerado um arquivo-objeto com a extensão "`.OBJ`" e este arquivo ainda não será executável; se seu programa for constituído por apenas um arquivo, o arquivo-objeto gerado será executável e terá a extensão "`.EXE`".

- **Make**. Esta opção compila o programa juntamente com todos seus arquivos-fonte, e cria um arquivo-objeto contendo o programa em linguagem de máquina. O nome principal do arquivo-objeto resultante é o mesmo nome que você forneceu para o arquivo-fonte e a extensão deste novo arquivo é "`.EXE`". Se houver apenas um arquivo em seu programa, esta opção é equivalente à anterior.

- **Build**. Esta opção é idêntica à opção **Make**; a diferença entre as duas opções não é importante aqui.

- **Information**. Se você quiser obter informações (por exemplo, número de linhas, tamanho do código objeto gerado em bytes, etc.) sobre o programa corrente você pode utilizar esta opção.

B.7 Menu Debug

Este menu contém opções de depuração do depurador (**debugger**) do Turbo Pascal. Examinar os detalhes de funcionamento deste depurador está além do escopo deste apêndice. Um programador consciente, entretanto, deve considerar seriamente a aprendizagem e uso desta ferramenta pois a mesma é indispensável para depuração e teste de programas não-triviais. Duas opções que podem ser utilizadas pelo programador novato são:

Output. Esta opção apresenta uma pequena janela na parte inferior da tela com saída de informação do programa sendo executado. Esta janela apresenta apenas informações textuais; gráficos não aparecem nela.

User Screen. Esta opção apresenta uma janela que ocupa todo o espaço da tela com saída de informação do programa que foi recentemente executado. Esta opção é parecida com a anterior mas é mais vantajosa quando o programa produz saídas longas ou gráficas. Esta janela pode ser dispensada digitando-se qualquer tecla ou clicando-se o mouse.

B.8 Menu Tools

O menu **Tools** não apresenta nenhuma opção essencial para um principiante.

B.9 Menu Options

O menu **Options** informa ao Turbo Pascal como ele deve compilar seu programa e não apresenta nenhuma opção essencial para um principiante.

B.10 Menu Windows

O menu **Windows** é importante para a manipulação de múltiplas janelas quando se está trabalhando com vários arquivos ao mesmo tempo; caso contrário, raramente precisa ser utilizado.

B.11 Menu Help

O menu **Help é** muito importante para auxílio ao programador iniciante ou usuários novos ao sistema (ou mesmo para aqueles que odeiam ler manuais tradicionais!).

Portanto, aprender a utilizar a ajuda on-line é essencial. As opções contidas neste menu são:

- **Contents.** Apresenta todo o conteúdo do Help On-line por tópicos, inclusive como usar o próprio Help.

- **Index.** Apresenta todo o conteúdo do Help On-line em ordem alfabética. Você pode digitar os caracteres iniciais da palavra que você está procurando e o Turbo Pascal localiza as palavras que começam com aqueles caracteres.

- **Topic Search.** Exatamente a mesma coisa da opção **Index.** (Talvez mantido aqui por questões de compatibilidade com versões anteriores?)

- **Previous Topic.** Apresenta a janela contendo um tópico de ajuda que você pesquisou anteriormente de maneira ordenada. Você pode voltar, em ordem, até 20 tópicos anteriores.

- **Using Help.** Ensina a utilizar o próprio Help On-line.

- **Files.** Apresenta ajuda específica sobre o tópico de manipulação de arquivos.

- **Compiler Directives.** Apresenta ajuda específica sobre diretivas do compilador Turbo Pascal.

- **Procedures and Functions.** Apresenta ajuda específica sobre procedimentos e funções (por exemplo, sqrt) embutidos em Turbo Pascal.

- **Reserved Words.** Apresenta ajuda específica sobre palavras reservadas em Turbo Pascal.

- **Standard Units.** *Unidades (Units)* são conjuntos de procedimentos e funções predefinidos, mas que não são embutidos em Turbo Pascal (i.e. não fazem, em si, parte da linguagem). Para serem utilizadas num programa, estas unidades precisam ser *importadas* (i.e., incluídas no programa). Esta opção apresenta ajuda específica sobre as unidades encontradas em Turbo Pascal.

- **Borland Pascal Language.** Apresenta ajuda sobre a implementação específica de Pascal da Borland (empresa proprietária de Turbo Pascal).

- **Error Messages.** Apresenta uma interpretação para cada mensagem de erro gerada por Turbo Pascal.

- **About.** Apresenta informação sobre a versão corrente de Turbo Pascal.

Apêndice C

LISTA DE VERIFICAÇÃO PARA PROGRAMAS EM PASCAL

Este apêndice contém uma lista de verificação contendo questões referentes a causas de problemas freqüentes em programas em Pascal. Aqui, não são apresentados itens para verificação de sintaxe, pois o compilador Pascal encarrega-se de verificar automaticamente a sintaxe dos programas automaticamente. Também, questões de estilo não são abordadas, embora apesar do reconhecido valor deste item para a construção de programas mais fáceis de entender e manter. A lista de itens de verificação é derivada da experiência do autor no ensino de Pascal como uma linguagem de introdução à programação, bem como de listas de verificação para outras linguagem. Evidentemente, esta lista é incompleta.

C.1 Variáveis e Constantes

- Verifique se alguma variável inteira ou real assume valores abaixo do mínimo ou acima do máximo permitido para o respectivo tipo na implementação de Pascal utilizada.

- Em alguns casos, é melhor utilizar um intervalo como tipo de uma variável que assume valores limitados do que o tipo inteiro. Por exemplo, uma variável representando a idade de uma pessoa é melhor ser declarada como:

```
var idade : [0..80];
```

do que como:

```
var idade : integer;
```

- Se o compilador utilizado não tem como padrão verificar se valor de uma variável do tipo intervalo está realmente dentro do intervalo, utilize uma diretiva de compilador (por exemplo, `{$R+}` em Turbo Pascal) para instruí-lo a fazer esta verificação.

• Antes de usar qualquer variável, certifique-se de que a mesma possui um valor *explicitamente* atribuído.

C.2 Expressões Aritméticas

• Verifique se a ordem de execução de cada expressão aritmética corresponde àquilo que você espera. Em outras palavras, convença-se de que entende a ordem de precedência dos operadores envolvidos em cada expressão aritmética. Em caso de dúvida, utilize parênteses.

• Tenha cuidado com expressões contendo conversão implícita de tipos. Lembre-se que se você utilizar "/" para dividir dois inteiros, o resultado será real. Se você pretende que o resultado seja inteiro, utilize **div**.

• Antes de efetuar uma divisão, verifique se o divisor é sempre diferente de zero.

• Certifique-se de que os argumentos passados para funções aritméticas são corretos (por exemplo, a função `sqrt(x)` apresenta um erro se o argumento passado for menor do zero).

C.3 Expressões Booleanas e Condicionais

• Condicionais e laços de repetição contendo expressões booleanas complexas são uma fonte de erro muito comum. Verifique cuidadosamente todas as ramificações possíveis.

• Certifique-se de que entende a forma como o compilador utilizado avalia expressões booleanas. Por exemplo, uma instrução **if** do tipo:

```
if (x <> 0) and (10/x > 0.5) then
```

resulta em erro quando x é igual a zero em compiladores que sempre avaliam toda a expressão booleana, mesmo quando se pode concluir o valor da mesma sem que seja necessária a avaliação completa da expressão. Com outros compiladores mais inteligentes, isto não ocorre, pois, quando x é

zero, a expressão `(10/x > 0.5)` não precisa ser avaliada para determinar que toda a expressão booleana resulta em **false**. (Isto é, sabe-se que A **and** B é verdadeiro apenas quando A é verdadeiro e B é verdadeiro; portanto, quando se sabe que A é falso, não é necessário avaliar B para concluir que A **and** B é falso.)

• Verifique se existe alguma expressão booleana numa condicional que sempre *resulta* em **true** ou *sempre* resulta em **false**. Condicionais desse tipo não fazem sentido.

• Cuidado com expressões **if** aninhadas cujas expressões booleanas não são independentes entre si. A ordem de colocação dos testes condicionais pode ser importante nestes casos (v. exemplo na *Seção 3.12.1*).

• É sempre recomendável utilizar **else** em instruções **case**, mesmo quando aparentemente não existe chance de esta parte da instrução ser executada.

C.4 Entrada e Saída

• Antes de uma instrução de entrada (**read** ou **readln**), utilize sempre um aviso (*prompt*) que informe ao usuário qual é a informação que o programa espera que seja introduzida.

• Em instruções **write** ou **writeln**, verifique se o dimensionamento (descrição de formato) de cada variável numérica pode acomodar o menor e o maior valores possíveis da variável. Lembre-se que o sinal e o ponto decimal devem ser levados em consideração nestes dimensionamentos.

C.5 Laços de Repetição

• Certifique-se de que a execução de cada laço de repetição sempre termina.

• Verifique se a variável de controle de cada instrução **for** é modificada dentro do corpo do laço. Isto não é recomendável e é uma fonte de erros.

• Verifique se o corpo de cada instrução **for** é realmente executado o número de vezes desejado.

• Lembre-se de que o final de um laço **while** não ocorre no instante em que a condição deixa de ser satisfeita, mas sim no momento em que ela é *avaliada e não satisfeita* (v. *Seção 3.12.3*).

• Uma instrução **repeat** deve ser utilizada quando se tem certeza de que o corpo do laço deve ser executado pelo menos uma vez. Verifique se este é o caso para cada instrução **repeat** do programa.

C.6 Procedimentos e Funções

• Certifique-se de que parâmetros *exclusivamente* de entrada não são modificados por um procedimento ou função.

• Verifique se qualquer comunicação de dados entre um procedimento e o programa principal, ou entre dois procedimentos, é realizada por meio de *passagem de parâmetros*. Evite utilizar variáveis globais dentro de procedimentos. (O mesmo comentário aplica-se a funções.)

• Variáveis locais a um procedimento (ou função) não podem ser utilizadas fora do mesmo.

• Considere a adoção de parâmetros variáveis para parâmetros que demandam muito armazenamento.

• Para cada função definida no programa, verifique se o valor correto é retornado (i.e., se um valor correto é atribuído ao nome da função pelo menos uma vez).

• Para cada procedimento ou função recursivo, certifique-se que a condição de parada é sempre atingida.

C.7 Arranjos, Cadeias de Caracteres e Registros

• Verifique se a dimensão de cada arranjo ou string é suficiente para conter o número máximo possível de elementos.

• Verifique se cada referência a elementos de um arranjo está dentro dos limites do índice do arranjo. (Utilize a diretiva `{$R+}` para que o Turbo Pascal faça a verificação de intervalos.)

• Utilize constantes simbólicas ao invés de valores inteiros nas definições de índices de arranjos e strings.

• Comparações entre strings numéricos podem não produzir efeitos desejados. Por exemplo, `'10' < '8'` é verdadeiro, apesar de `10 < 8` ser falso.

• Evite o uso de instruções **with** (principalmente *aninhadas*) que possam comprometer a claridade do programa.

• Utilize sempre um campo indicador na definição de um registro variante. Utilize o campo indicador para determinar quais são os campos variantes válidos em cada instante.

C.8 Arquivos

• Certifique-se de que arquivos de entrada realmente existem.

• Antes de ler dados pela primeira vez em cada arquivo de entrada, verifique se a instrução **reset** foi utilizada.

• Antes de gravar dados pela primeira vez em cada arquivo de saída, verifique se a instrução **rewrite** foi utilizada.

• Não esqueça de fechar um arquivo quando não mais precisar dele.

• Cuidado para não tentar escrever para um arquivo que já foi fechado.

• Sempre teste se o final do arquivo foi atingido antes de ler dados de um arquivo de entrada.

• Para cada instrução de entrada/saída envolvendo um arquivo binário, verifique se as variáveis utilizadas na instrução são do mesmo tipo base do arquivo.

C.9 Ponteiros e Alocação Dinâmica

- Para cada referência feita a um ponteiro, verifique se o mesmo aponta para um endereço válido.

- Utilize **dispose** para liberar um espaço alocado quando este não for mais necessário.

- Utilize **nil** após cada instrução **dispose** para evitar que o espaço liberado seja inadvertidamente referenciado.

- Lembre-se que o espaço alocado dinamicamente dentro de um procedimento ou função não é liberado automaticamente quando encerra a execução do procedimento (ou função). Isso deve ser feito explicitamente no final do procedimento através de instruções **dispose**.

- Em funções que retornam ponteiros, normalmente aloca-se espaço para estes ponteiros dentro da função. Não libere este espaço no final da função.

C.10 Comentários

- Verifique se cada comentário é pertinente.

- Existe algum comentário desnecessário? Um comentário desnecessário pode distrair a atenção de quem lê o programa.

- Porções do programa que contêm truques e sutilezas merecem comentários mais elaborados.

Evidentemente, falhas em comentários não resultam em programas defeituosos (do ponto-de-vista do usuário final). Entretanto, programas bem comentados são bem mais fáceis de serem mantidos. Também, aprenda a escrever comentários como se estivesse explicando seu programa para outra pessoa; isto ajuda a descobrir erros logo cedo no programa.

Bibliografia

1. Koffman, Elliot B., 1986. *Turbo Pascal: A Problem Solving Approach*. Addison-Wesley Publishing.

2. Deitel, Harvey M., 1990. *An Introduction to Operating Systems*. Second Edition. Addison-Wesley Publishing.

3. Mayers, J. M. 1979. *The Art of Software Testing*. John Wiley & Sons.

4. Tenenbaum, A. M. e Augenstein, M. J., 1981. *Data Structures Using Pascal*. Prentice-Hall.